



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΪΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

**FPGA Architectures of
Deep Convolutional Neural Networks
for Satellite Image Classification**

*Αρχιτεκτονικές FPGA για
Βαθιά Συνελικτικά Νευρωνικά Δίκτυα
Ταξινόμησης Δορυφορικών Εικόνων*

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΡΕΠΠΑ - ΧΡΥΣΟΒΙΤΣΙΝΟΥ ΔΗΜΗΤΡΙΟΥ

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αν. Καθηγητής ΗΜΜΥ

Αθήνα, Σεπτέμβριος 2018



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

FPGA Architectures of Deep Convolutional Neural Networks for Satellite Image Classification

*Αρχιτεκτονικές FPGA για
Βαθιά Συνελικτικά Νευρωνικά Δίκτυα
Ταξινόμησης Δορυφορικών Εικόνων*

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΡΕΠΠΑ - ΧΡΥΣΟΒΙΤΣΙΝΟΥ ΔΗΜΗΤΡΙΟΥ

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αν. Καθηγητής ΗΜΜΥ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Σεπτεμβρίου 2018.

(Υπογραφή)

(Υπογραφή)

(Υπογραφή)

.....
Δημήτριος Ι. Σούντρης
Αν. Καθηγητής ΗΜΜΥ

.....
Κιαμάλ Ζ. Πεσχυμεστζή
Καθηγητής ΗΜΜΥ

.....
Κωνσταντίνος Καράντζαλος
Επίκουρος Καθηγητής ΣΑΤΜ

Αθήνα, Σεπτέμβριος 2018



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Copyright ©–All rights reserved Ρέππας - Χρυσοβιτισινός Δημήτριος, 2018.

Με την επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Υπεύθυνη Δήλωση

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Διπλωματικής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Δηλώνω, συνεπώς, ότι αυτή η Διπλωματική Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

.....

Ρέππας - Χρυσοβιτισινός Δημήτριος

Περίληψη

Την τελευταία δεκαετία τα Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ) αναδείχθηκαν ως μία από τις καλύτερες προσεγγίσεις για την αντιμετώπιση ορισμένων προκλήσεων της Όρασης Υπολογιστών, όπως η ταξινόμηση εικόνων και ο εντοπισμός αντικειμένων σε αυτές. Για την αξιοποίηση των ΣΝΔ σε ένα πλήθος εφαρμογών απαιτούνται υψηλών επιδόσεων και χαμηλής κατανάλωσης ενέργειας υπολογιστικά συστήματα. Στην κατεύθυνση αυτή, η τεχνολογία των FPGA αποτελεί έναν εξαιρετικό υποψήφιο για την υλοποίηση ΣΝΔ σε ενσωματωμένα συστήματα. Αποφασίσαμε να διερευνήσουμε FPGA αρχιτεκτονικές για Συνελικτικά Νευρωνικά Δίκτυα ταξινόμησης δορυφορικών εικόνων, έχοντας κατά νου τις ανάγκες της επιστημονικής κοινότητας της Τηλεπισκόπησης. Η παρούσα διπλωματική εργασία παραδίδει (i) FPGA αρχιτεκτονικές σχεδιασμένες σε VHDL για την υλοποίηση Συνελικτικών Νευρωνικών Δικτύων χρησιμοποιώντας αποκλειστικά την μνήμη της προγραμματιζόμενης λογικής και (ii) το "Modified Cifar-10 Full", ένα βαθύ ΣΝΔ λίγων bit, κατασκευασμένο και εκπαιδευμένο με το Caffe framework σε εικόνες του SAT-6 airborne dataset. Το συγκεκριμένο σύνολο δεδομένων αποτελείται από εικόνες των 28×28 εικονοστοιχείων, χωρικής ακρίβειας του ενός μέτρου, οι οποίες ανήκουν σε 6 κατηγορίες: άγρονη γη, δέντρα, λιβάδια, δρόμοι, κτήρια και υδάτινες μάζες. Το Modified Cifar-10 Full ΣΝΔ που καταθέτουμε, επιτυγχάνει να ταξινομήσει σωστά τις εικόνες του SAT-6 airborne dataset με top-1 ακρίβεια 94,89%, χρησιμοποιώντας 8-bit για τα βάρη του δικτύου και 4-bit για τα κανάλια εισόδων και εξόδων εντός του. Η πλήρης εκτέλεση του "Modified Cifar-10 Full" ΣΝΔ πραγματοποιείται αποκλειστικά στην προγραμματιζόμενη λογική του FPGA, χωρίς την χρήση εξωτερικής μνήμης ή την διαμεσολάβηση μιας CPU. Τοποθετημένη στη συσκευή Xilinx Zynq Z-7020 SoC, η αρχιτεκτονική που σχεδιάσαμε λειτουργεί στα 100 MHz, μπορεί να ταξινομήσει 4650 εικόνες το δευτερόλεπτο και καταναλώνει 1,76 Watt ενέργεια, επιτυγχάνοντας ×377 επιτάχυνση στο ρυθμό ταξινόμησης εικόνων σε σχέση με την εκτέλεση του ΣΝΔ αποκλειστικά στον ενσωματωμένο Cortex A9-Arm επεξεργαστή. Σε σύγκριση με HLS υλοποιήσεις επί της ίδιας συσκευής, η σχεδιάσή μας επιτυγχάνει ×2,4 επιτάχυνση. Τέλος, συγκρινόμενη με το Fathom Neural Compute Stick της Movidius που βασίζεται στο Myriad Visual Processing Unit, η σχεδιάσή μας επί του Xilinx Zynq Z-7020 SoC επιτυγχάνει ×16 υψηλότερο ρυθμό ταξινόμησης.

Λέξεις Κλειδιά

Συνελικτικά Νευρωνικά Δίκτυα, Μηχανική Μάθηση, Ταξινόμηση εικόνων, FPGA, VHDL, δορυφορικές εικόνες, επιταχυντές υλικού, ενσωματωμένα συστήματα, Caffe framework

Abstract

Over the past decade Convolutional Neural Networks (CNNs) emerged as the state-of-the-art approach to tackle certain Computer Vision problems such as image classification and object detection. Employing CNNs in a multitude of applications requires high-performance, low-power computing. Field Programmable Gate Array (FPGA) technologies have been identified as exceptional candidates for the implementation of CNNs' inference stage. We explored FPGA architectures for satellite image classification using CNNs, having in mind the needs of the remote sensing community. This thesis delivers a high-throughput, low-power FPGA design of a highly accurate deep Convolutional Neural Network, suitable for embedded systems placed on UAVs-drones/satellites, classifying images at the edge of the computing cloud. More specifically, we deliver i) FPGA architectures designed in VHDL for the inference of deep CNNs, using only the on-chip memory of the Programmable Logic and, ii) a low bit-width customized CNN model (the "Modified Cifar-10 Full"), created and trained with the Caffe framework on the SAT-6 airborne dataset. The SAT-6 airborne dataset consists of 28×28 pixel images of one meter spatial resolution, covering six land cover classes (barren land, trees, grassland, roads, buildings and water bodies). Using 8-bit weights and 4-bit feature maps, Modified Cifar-10 Full achieves 94.89% top-1 accuracy on the SAT-6 airborne dataset. The whole processing of Modified Cifar-10 Full is performed on the Programmable Logic of the FPGA chip without a need for an external memory or a CPU to coordinate and monitor the algorithm's execution. When mapped on the Xilinx Zynq Z-7020 SoC our design operates at 100MHz consuming 1.76 Watt and can classify 4650 images per second. Compared against other implementations, our design on the Xilinx Zynq Z-7020 SoC achieves throughput speedups of $\times 377$ against an implementation solely on the embedded Cortex-A9 Arm processor, $\times 2.4$ against High-Level Synthesis implementations on the same device and, $\times 16.2$ against the Fathom Neural Compute Stick by Movidius featuring the Myriad Visual Processing Unit.

Key words: Convolutional Neural Networks, FPGA, VHDL, Machine Learning, Deep Learning, Image Classification, Satellite Images, Caffe framework, Hardware Acceleration, Zynq SoC, Embedded systems, CNN, Remote Sensing

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή κ. Δημήτρη Σούντρη για την ευκαιρία που μου έδωσε να εκπονήσω τη συγκεκριμένη διπλωματική εργασία στο Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων.

Η εργασία αυτή δεν θα ήταν εφικτή χωρίς την καθοδήγηση και υποστήριξη από τον μεταδιδακτορικό ερευνητή Γιώργο Λεντάρη. Οι συζητήσεις μαζί του είναι ιδιαίτερα εκπαιδευτικές και οι γρήγορες και εύστοχες παρατηρήσεις του καθοριστικές για την πορεία της εργασίας. Για τα παραπάνω, αλλά και για την πάντα καλή του διάθεση, τον ευχαριστώ ιδιαίτερα.

Εισαγωγικό Σημείωμα - Δομή Εργασίας

Το πρώτο κεφάλαιο προσφέρει μια πλήρη επισκόπηση της εργασίας στα ελληνικά. Τα υπόλοιπα κεφάλαια είναι γραμμένα στα αγγλικά. Αναλυτικότερα, το δεύτερο κεφάλαιο κάνει μια εισαγωγή στο θέμα, το τρίτο παρουσιάζει ευσύννοπτα το θεωρητικό υπόβαθρο της εργασίας, τα κεφάλαια 4 & 5 παρουσιάζουν τη δική μας πραγμάτευση επί του ζητήματος. Τέλος στο κεφάλαιο 6 συνοψίζουμε τα βασικά αποτελέσματα της εργασίας μας και προτείνουμε κατευθύνσεις για μελλοντική έρευνα. Ο κώδικας ο οποίος χρησιμοποιήθηκε στην παρούσα εργασία είναι διαθέσιμος κατόπιν επικοινωνίας με το εργαστήριο μικροϋπολογιστών & ψηφιακών συστημάτων.

Περιεχόμενα

Περίληψη	i
Ευχαριστίες	iii
Εισαγωγικό Σημείωμα - Δομή Εργασίας	v
Περιεχόμενα	vii
A' Αρχιτεκτονικές FPGA για Βαθιά Συνελικτικά Νευρωνικά Δίκτυα Ταξινόμησης Δορυφορικών Εικόνων	1
A'.1 Εισαγωγή	1
A'.1.1 Στόχοι της Διπλωματικής Εργασίας	4
A'.2 Τεχνητά Νευρωνικά Δίκτυα	5
A'.2.1 Συνελικτικά Νευρωνικά Δίκτυα	7
A'.3 Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών (FPGA)	11
A'.3.1 Συνελικτικά Νευρωνικά Δίκτυα σε FPGA	11
A'.4 Εκπαίδευση ΣΝΔ και Βελτιστοποίηση	14
A'.4.1 Προσαρμογή του ΣΝΔ μοντέλου στην περίπτωση μας	15
A'.4.2 Αποτελέσματα εκπαίδευσης του Modified Cifar-10 Full ΣΝΔ	18
A'.4.3 Συμπιέζοντας το ΣΝΔ: Βελτιστοποίηση μήκους λέξης	20
A'.5 Υλοποίηση σε FPGA	21
A'.5.1 Προσέγγιση	21
A'.5.2 Αρχιτεκτονική FPGA - Τα βασικά στοιχεία	23
A'.6 Οργάνωση της on-chip μνήμης	28
A'.7 Εξερεύνηση του χώρου σχεδίασης	29
A'.7.1 Αποτελέσματα εξερεύνησης χώρου σχεδίασης	32
A'.7.2 Συγκριτική εκτίμηση της υλοποίησης	34

Contents

1	Executive Summary in Greek	1
2	Introduction	45
2.1	Motivation	45
2.2	Thesis Goals	48
3	Background and concepts	51
3.1	Convolutional Neural Networks	51
3.1.1	A step back: The bigger picture of Machine Learning	51
	Types of Machine Learning	52
	A bit of Supervised Learning	53
3.1.2	The problem of Image Classification	56
3.1.3	Introduction to Artificial Neural Networks	58
3.1.4	Introduction to Convolutional Neural Networks	61
3.1.5	Common layers used to build CNNs	63
3.1.6	Convolutional Neural Network Architectures	69
3.2	Field-Programmable Gate Arrays	72
3.2.1	FPGA Programming	74
3.2.2	FPGA Fabric	76
3.3	Convolutional Neural Networks in FPGAs	80
4	CNN training and Optimization	83
4.1	The Caffe Framework	83
4.2	The SAT-4 & SAT-6 Airborne datasets	84
4.3	CNN models in this Thesis	85
4.3.1	The original "Cifar-10 Full" CNN model	85
4.3.2	Why use the "Cifar-10 Full" model?	87
4.3.3	Customizing the "Cifar-10 Full" model to our needs	88
4.3.4	The "Modified Cifar-10 Full" CNN model	90
4.3.5	The "Modified Cifar-10 Full" CNN model's Training & Results	93
4.4	Compressing the network: Word-length Optimization	95

5	FPGA Implementation	101
5.1	Design Approach	101
5.2	The design's key components	103
5.2.1	The "Expander" component	105
5.2.2	The "2D Convolution Engine" component	105
5.2.3	The "SP" component	107
5.2.4	The "Window-Gen" component	107
5.2.5	The "Pooling Layer" component	108
5.2.6	The "Fully Connected Layer" component	110
5.3	Bit-width Calculations	112
5.4	On-Chip Memory Organization	113
5.5	Intra-Layer Control	115
5.6	Inter-Layer Control	118
5.7	Design Space Exploration	122
5.7.1	Approach	122
5.7.2	Results	127
5.8	Final Configuration and FPGA Implementation Results	132
5.8.1	Comparison to relevant works	137
6	Conclusions	145
	Bibliography	149

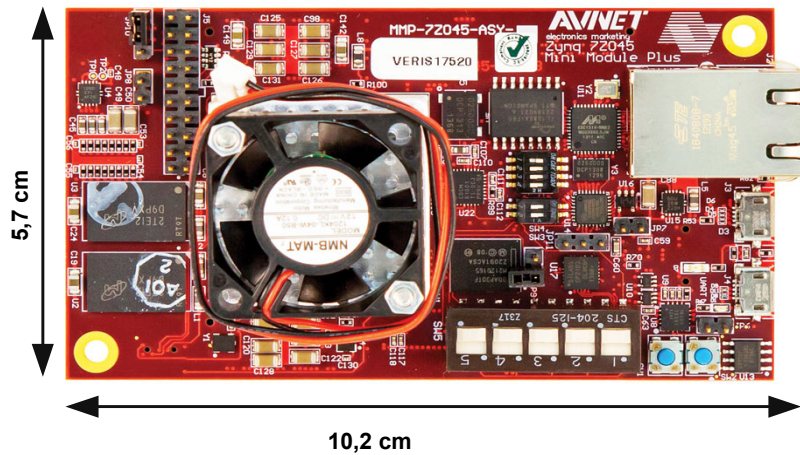
CHAPTER 1

Executive Summary in Greek

Α' Αρχιτεκτονικές FPGA για Βαθιά Συνελικτικά Νευρωνικά Δίκτυα Ταξινόμησης Δορυφορικών Εικόνων

Α'.1 Εισαγωγή

Την τελευταία δεκαετία, τα Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ) αναδείχθηκαν ως ένας από τους καλύτερους τρόπους για την επίλυση ορισμένων προβλημάτων από το πεδίο της Όρασης Υπολογιστών. Η υψηλή ακρίβεια με την οποία ανταποκρίνονται τα Συνελικτικά Νευρωνικά Δίκτυα σε προβλήματα όπως η αναγνώριση εικόνων και ο εντοπισμός αντικειμένων σε αυτές, ταυτόχρονα με την ραγδαία αύξηση των διαθέσιμων ψηφιακών δεδομένων και την εμφάνιση του Διαδικτύου των Πραγμάτων (IoT), έχουν αυξήσει το ενδιαφέρον για την αξιοποίησή τους σε πλήθος ερευνητικών, βιομηχανικών και εμπορικών εφαρμογών. Σε ένα σημαντικό πλήθος εφαρμογών, όπως για παράδειγμα τα αυτόνομα οχήματα και ο έλεγχος βιομηχανικών διαδικασιών, τα εμπλεκόμενα υπολογιστικά συστήματα απαιτείται να είναι μικρών φυσικών διαστάσεων, να λειτουργούν με χαμηλή κατανάλωση ενέργειας και ταυτόχρονα να μπορούν να λειτουργούν εντός αυστηρών προδιαγραφών στους χρόνους εκτέλεσης. Τα ΣΝΔ παρουσιάζουν υψηλή υπολογιστική πολυπλοκότητα και προκειμένου να εφαρμοστούν επιτυχώς σε προβλήματα που βασίζονται στην αναγνώριση εικόνας, χρειάζεται να υλοποιηθούν σε ενσωματωμένα υπολογιστικά συστήματα χαμηλής κατανάλωσης ενέργειας και υψηλών επιδόσεων. Σε αυτή την κατεύθυνση, η τεχνολογία των FPGA -Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών- έχει προσδιοριστεί ως μια εξαιρετική λύση για την υλοποίηση Τεχνητών Νευρωνικών Δικτύων. Τα FPGA αποτελούν προγραμματιζόμενα ψηφιακά κυκλώματα, επί των οποίων μπορούν να σχεδιαστούν εξειδικευμένες αρχιτεκτονικές παράλληλης επεξεργασίας. Η ιδιότητά τους αυτή δένει αρμονικά με το γεγονός ότι τα Τεχνητά Νευρωνικά Δίκτυα είναι εγγενώς σχεδιασμένα ως εξαιρετικά μεγάλης παραλληλίας υπολογιστικά μοντέλα. Επιπρόσθετα, τα FPGA έχουν ντετερμινιστικό χρόνο εκτέλεσης και χαμηλή κατανάλωση ενέργειας. Σε σχέση με τις υψηλών επιδόσεων κάρτες γραφικών, τα FPGA προσφέρουν μέχρι και δύο τάξεις μεγέθους λιγότερη κατανάλωση ενέργειας. Ενδεικτικά αναφέρουμε πως η NVIDIA Tesla K40 GPU καταναλώνει 235 Watt, ενώ η ετερογενής πλατφόρμα της Xilinx Zynq Z-7020 SoC, η οποία περιλαμβάνει μικροεπεξεργαστή της ARM και FPGA τεχνολογία στο ίδιο chip, καταναλώνει 2,5 Watt. Η χαμηλή κατανάλωση ενέργειας, η υψηλή επίδοση και η δυνατότητα επαναπρογραμματισμού των FPGA, έχουν οδηγήσει εταιρείες όπως η Microsoft και η Amazon να χρησιμοποιήσουν FPGA στα κέντρα δεδομένων τους για να μειώσουν το κόστος ενέργειας, να επιταχύνουν λειτουργίες

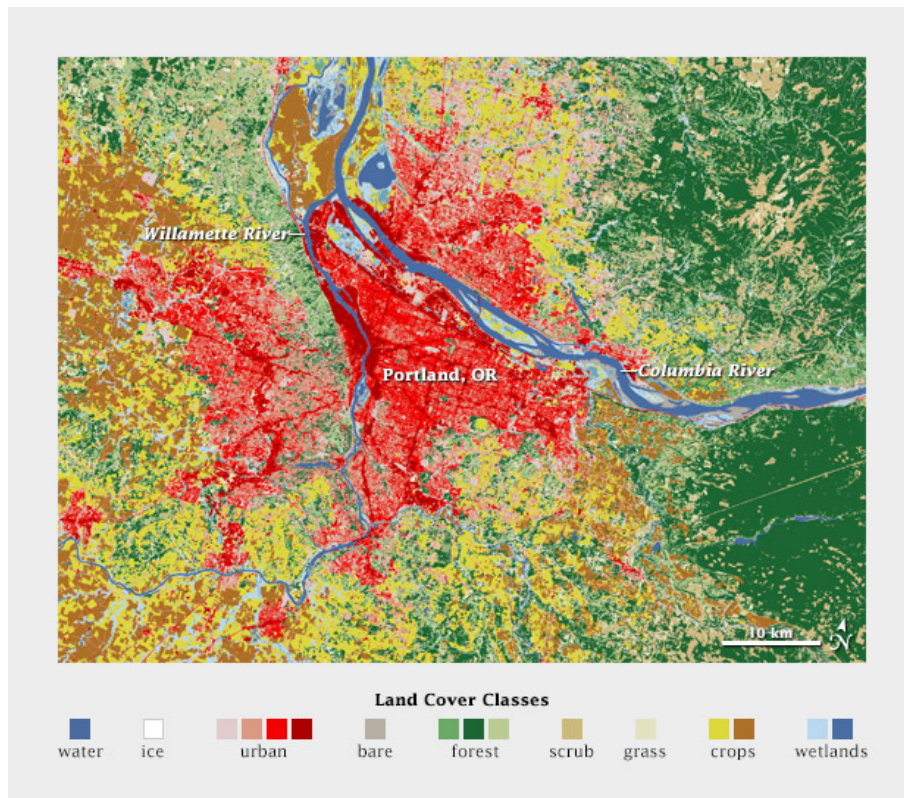


Σχήμα Α'.1: Παράδειγμα ενός ενσωματωμένου υπολογιστικού συστήματος. Το Xilinx Zynq MMP, το οποίο περιέχει ένα Zynq-Z7045 SoC. Η συγκεκριμένη πλατφόρμα κοστίζει περίπου 3000€. Πιο οικονομικές λύσεις υπάρχουν, όπως για παράδειγμα το Zybo Zynq-7000 το οποίο περιέχει ένα System on Chip με συγκριτικά λιγότερους FPGA πόρους και κοστίζει περίπου 300€.

τους και για να προσφέρουν δυνατότητες Τεχνητής Νοημοσύνης σε πόρους του υπολογιστικού νέφους. Παρότι υπάρχουν λύσεις για την εκτέλεση των Συνελικτικών Νευρωνικών Δικτύων στο υπολογιστικό νέφος, σε πλήθος περιπτώσεων υπάρχει ανάγκη οι υπολογισμοί να εκτελούνται τοπικά. Παραδείγματος χάρη, σε περιπτώσεις που οι χρόνοι απόκρισης είναι κρίσιμοι, σε περιπτώσεις που δεν μπορεί να εξασφαλιστεί σταθερά γρήγορη σύνδεση στο ίντερνετ, σε περιπτώσεις που υπάρχουν ζητήματα ασφαλείας, οι λύσεις τους υπολογιστικού νέφους υστερούν. Τα τελευταία χρόνια αρκετοί επιταχυντές υλικού για ΤΝΔ έχουν προταθεί.

Η υψηλής ακρίβειας ταξινόμηση εικόνων και ο εντοπισμός αντικειμένων είναι ιδιαίτερα σημαντική σε ένα πλήθος εφαρμογών που βασίζονται στις αεροφωτογραφίες και τα δορυφορικά δεδομένα. Παραδοσιακά, σε αυτές τις εφαρμογές τα δεδομένα δεν αξιολογούνται την ώρα που δημιουργούνται, στα τοπικά υπολογιστικά συστήματα, αλλά αποθηκεύονται ή αποστέλλονται για να υποβληθούν σε επεξεργασία σε απομακρυσμένα από το σημείο συλλογής τους υπολογιστικά συστήματα. Η εγκατάσταση ενσωματωμένων συστημάτων ικανών να εκτελούν Συνελικτικά Νευρωνικά Δίκτυα πάνω σε μη-επανδρωμένα αεροσκάφη και δορυφόρους μπορεί να προσφέρει την αυτοματοποιημένη, υψηλής ακρίβειας και σε πραγματικό χρόνο ταξινόμηση εικόνων, χωρίς την ανάγκη της αποστολής των δεδομένων σε απομακρυσμένα υπολογιστικά συστήματα. Αυτό μπορεί να είναι ιδιαίτερα χρήσιμο σε ένα πλήθος εφαρμογών όπως σε συστήματα έγκαιρης προειδοποίησης τοποθετημένα σε δορυφόρους και η δημιουργία αυτόνομων μη-επανδρωμένων αεροσκαφών για καινοτόμες τεχνολογικές λύσεις πχ στην διαχείριση φυσικών καταστροφών, σε αυτοματοποιημένη γεωργική παραγωγή ακριβείας και σε υπηρεσίες αυτοματοποιημένης ταχυμεταφοράς προϊόντων. Σε τέτοιου τύπου εφαρμογές, η κατανάλωση ενέργειας και το φυσικό μέγεθος του ενσωματωμένου υπολογιστικού συστήματος αποτελούν κρίσιμους παράγοντες και συνεπώς η επιλογή μιας κατάλληλης ενσωματωμένης πλατφόρμας είναι ιδιαίτερα σημαντική. Σε αρκετές εφαρμογές ενδιαφερόμαστε για την ταξινόμηση αερο-

φωτογραφιών/ δορυφορικών εικόνων σε σχέση με τη κάλυψη γης. Ως κάλυψη γης ορίζεται η φυσική και βιολογική κάλυψη της επιφάνειας της γης. Παραδείγματα κλάσεων στην κάλυψη γης είναι για παράδειγμα τα δάση, οι υδάτινοι σχηματισμοί, οι δομημένες περιοχές και οι αγροτικές περιοχές. Οι χάρτες κάλυψης γης συνεισφέρουν στην παρακολούθηση και τον εντοπισμό ζητημάτων όπως η αποψίλωση, η αναδάσωση, η μείωση των υδάτινων πόρων και η επέκταση των αστικών περιοχών.



Σχήμα Α'2: Παράδειγμα ενός χάρτη κάλυψης γης από την περιοχή του Portland, Oregon USA (NASA Earth Observatory).

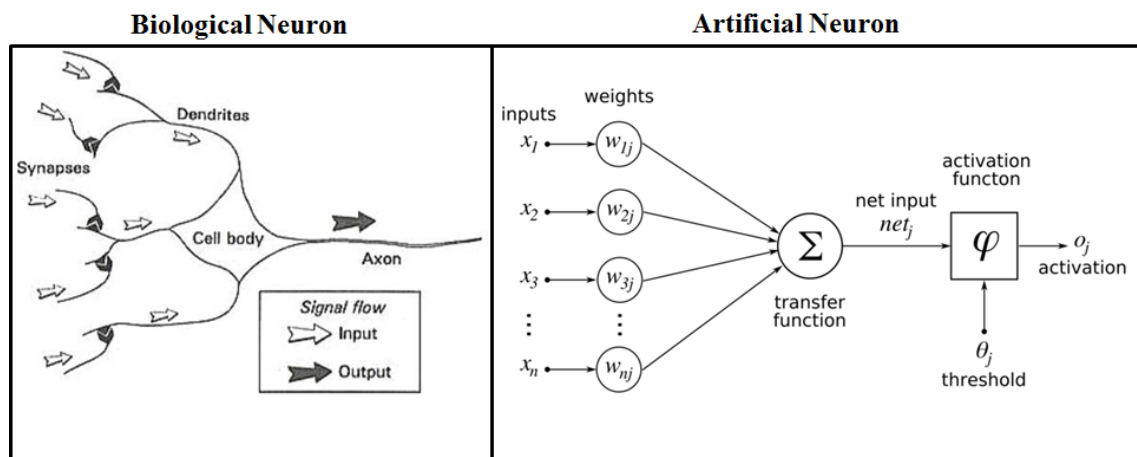
A'.1.1 Στόχοι της Διπλωματικής Εργασίας

Η παρούσα διπλωματική εργασία έχει τους κάτωθι στόχους:

- Την ανασκόπηση των πρόσφατων εξελίξεων στο αντικείμενο της Μηχανικής Μάθησης σε σχέση με τα Συνελικτικά Νευρωνικά Δίκτυα.
- Την ανασκόπηση της υπάρχουσας βιβλιογραφίας σε σχέση με τις προσεγγίσεις στη σχεδίαση υλοποιήσεων των ΣΝΔ σε υλικό.
- Την επιλογή και εκπαίδευση μιας αρχιτεκτονικής ΣΝΔ κατάλληλης για την ταξινόμηση δορυφορικών εικόνων.
- Την εφαρμογή βελτιστοποιήσεων στην αρχιτεκτονική του επιλεγμένου ΣΝΔ με σκοπό την υλοποίησή του σε FPGA.
- Την σχεδίαση αρχιτεκτονικών FPGA για την υλοποίηση Συνελικτικών Νευρωνικών Δικτύων.
- Την βελτιστοποίηση και αξιολόγηση της FPGA αρχιτεκτονικής για το συγκεκριμένο Συνελικτικό Νευρωνικό Δίκτυο σε σχέση με την ρυθμό ταξινόμησης εικόνων και την χρήση πόρων της συσκευής, για αρκετές συσκευές της Xilinx Zynq 7000 SoC οικογένειας.

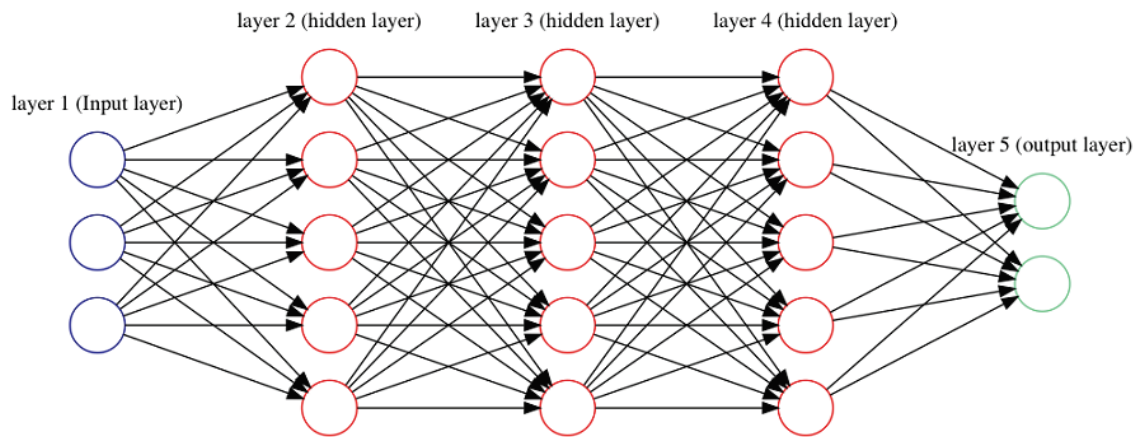
Α'.2 Τεχνητά Νευρωνικά Δίκτυα

Τα Τεχνητά Νευρωνικά Δίκτυα είναι υπολογιστικά συστήματα και αλγόριθμοι εμπνευσμένα από τον ανθρώπινο εγκέφαλο και την παρατήρηση ότι ο ανθρώπινος εγκέφαλος λειτουργεί με πολύ διαφορετικό τρόπο από τους συμβατικούς, ψηφιακούς υπολογιστές. Το βασικό δομικό στοιχείο των ΤΝΔ είναι ο τεχνητός νευρώνας. Πλήθος από τεχνητούς νευρώνες διασυνδέονται για να σχηματίσουν ένα δίκτυο, η συμπεριφορά του οποίου εξαρτάται άμεσα από το πλήθος των νευρώνων που περιέχει και από την τοπολογία με την οποία αυτοί διασυνδέονται. Υλοποιήσεις Τεχνητών Νευρωνικών Δικτύων τόσο σε υλικό, όσο και λογισμικό έχουν εφαρμοστεί σε πλήθος εφαρμογών όπως η αναγνώριση χαρακτήρων (πχ OCR), η πρόβλεψη χρονοσειρών και ο έλεγχος διεργασιών και συστημάτων [Wid94].



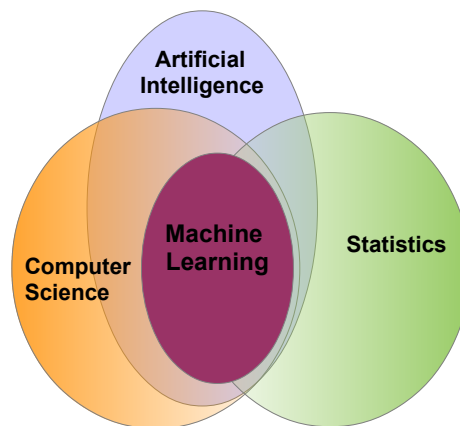
Σχήμα Α'.3: Το απλοποιημένο μοντέλο ενός βιολογικού νευρώνα (αριστερά) και το μαθηματικό μοντέλο ενός τεχνητού νευρώνα (δεξιά). Σε κάθε σήμα εισόδου του τεχνητού νευρώνα αντιστοιχεί ένα βάρος με το οποίο πολλαπλασιάζεται και τα αποτελέσματα αθροίζονται. Για να παραχθεί η έξοδος του νευρώνα, στο σταθμισμένο άθροισμα εφαρμόζεται μια μη-γραμμική συνάρτηση ενεργοποίησης $y = \varphi(\sum [x_i \cdot w_i] + w_b)$.

Για την εκπαίδευση των Τεχνητών Νευρωνικών Δικτύων επιστρατεύονται αλγόριθμοι από το πεδίο της Μηχανικής Μάθησης (Machine Learning). Η μηχανική μάθηση χρησιμοποιεί τεχνικές από τη στατιστική και τη μαθηματική βελτιστοποίηση και βασίζεται στην υπόθεση ότι η γνώση μπορεί να εξαχθεί μέσα από πλήθος παραδειγμάτων. Επί του παρόντος υπάρχουν δύο βασικές προσεγγίσεις στη μηχανική μάθηση: α) η επιβλεπόμενη μάθηση και β) η μη-επιβλεπόμενη μάθηση. Στα προβλήματα της όρασης υπολογιστών ως επί το πλείστον χρησιμοποιούνται επιβλεπόμενες τεχνικές μάθησης. Στην περίπτωση του προβλήματος της ταξινόμησης, στην επιβλεπόμενη μάθηση έχουμε ένα πλήθος από προ-ταξινομημένα παραδείγματα, που ονομάζονται δεδομένα εκπαίδευσης (training data). Διαθέτοντας τις μεταβλητές εισόδου X του αλγορίθμου για κάθε παράδειγμα, και γνωρίζοντας τις αντίστοιχες εξόδους Y , προσπαθούμε με επαναληπτικές μεθόδους να προσεγγίσουμε τη συνάρτηση f , όπου $Y = f(X)$. Σε αντίθεση με τη μαθηματική βελτιστοποίηση, στη μηχανική μάθηση δεν ενδιαφερόμαστε μόνο να προσεγγίσουμε ικανοποιητικά τη συνάρτηση f , αλλά επιθυμούμε το μοντέλο που κατασκευ-



Σχήμα Α'.4: Παράδειγμα ενός πλήρως διασυνδεδεμένου τεχνητού νευρωνικού δικτύου. Όλοι οι νευρώνες του ενός επιπέδου συνδέονται με όλους τους νευρώνες των γειτονικών επιπέδων.

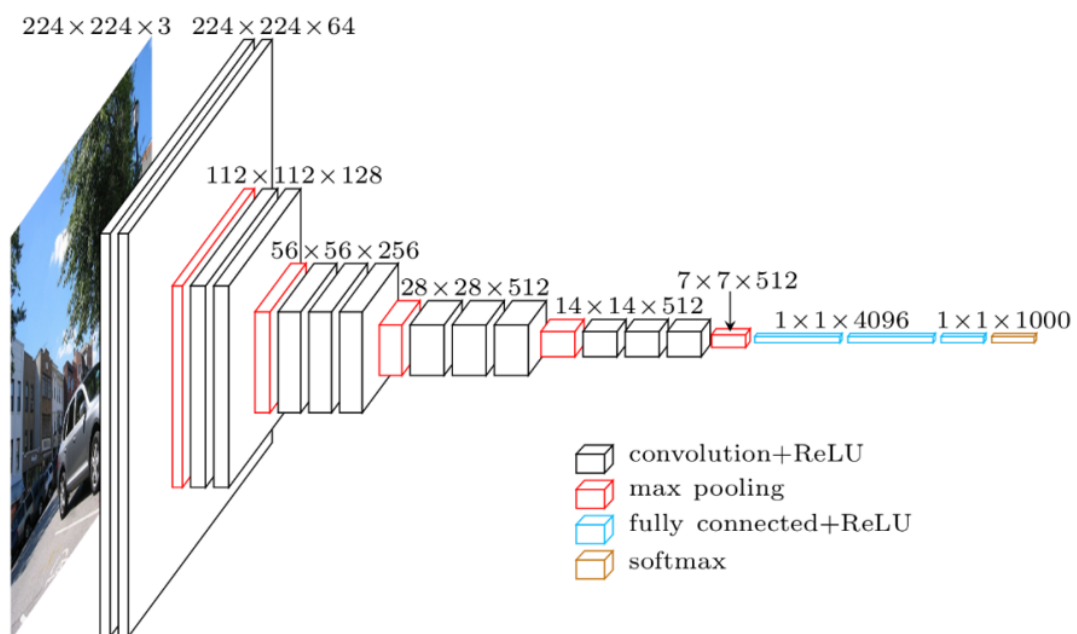
άζουμε να μπορεί να γενικεύσει, ταξινομώντας σωστά και δεδομένα τα οποία δεν έχει ξαναδεί. Για το σκοπό αυτό αξιολογούμε την επίδοση του μοντέλου χρησιμοποιώντας ένα αυστηρά ξεχωριστό σετ δεδομένων ελέγχου (test data), με τα οποία το μοντέλο δεν έχει εκπαιδευτεί. Ο πιο συχνά χρησιμοποιούμενη μέθοδος εκπαίδευσης Τεχνητών Νευρωνικών Δικτύων ονομάζεται "Backpropagation".



Σχήμα Α'.5: Η σχέση της μηχανικής μάθησης με άλλα επιστημονικά πεδία.

A'.2.1 Συνελικτικά Νευρωνικά Δίκτυα

Τα Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ) είναι μια κατηγορία Τεχνητών Νευρωνικών Δικτύων (ΤΝΔ) που πρωτοεμφανίστηκαν στις αρχές του 1990 και τα οποία έχουν σχεδιαστεί ειδικά για την αναγνώριση δισδιάστατων αντικειμένων, παρουσιάζοντας υψηλό βαθμό αναλλοίωτης συμπεριφοράς κατά την μετάθεση, κλιμάκωση, στρέβλωση και άλλες παραμορφώσεις της εισόδου. Τα Συνελικτικά Νευρωνικά Δίκτυα οργανώνονται σε επίπεδα, κάθε ένα εκ των οποίων εκτελεί μια μαθηματική πράξη. Πολλά επίπεδα διασυνδέονται σχηματίζοντας έναν κατευθυνόμενο άκυκλο γράφο. Όταν χρησιμοποιούνται για την ταξινόμηση εικόνων, τα ΣΝΔ παίρνουν ως είσοδο την εικόνα και παράγουν ένα διάλυμα από σκορ που υποδεικνύουν την κλάση στην οποία ανήκει η εικόνα. Σε σχέση με τα πλήρως διασυνδεδεμένα ΤΝΔ, τα ΣΝΔ διαθέτουν τα εξής διαφορετικά ποιοτικά χαρακτηριστικά: α) οι νευρώνες κάθε επιπέδου δεν είναι συνδεδεμένοι με όλους τους νευρώνες των γειτονικών επιπέδων, αλλά μόνο με ένα μικρό πλήθος από τους νευρώνες του προηγούμενου επιπέδου ο καθένας, β) οι νευρώνες κάθε επιπέδου μοιράζονται κοινά βάρη μεταξύ τους, σε αντίθεση με τα παραδοσιακά ΤΝΔ όπου κάθε νευρώνας έχει το δικό του βάρος και γ) ανάμεσα στα επιμέρους επίπεδα νευρώνων μεσολαβούν διατάξεις που πραγματοποιούν χωρική υποδειγματοληψία. Με τα ποιοτικά αυτά διαφορετικά χαρακτηριστικά, οι ερευνητές που πρωτο-σχεδίασαν τα ΣΝΔ επιχείρησαν να ενσωματώσουν πρότερη γνώση στη δομή του δικτύου, όπως για παράδειγμα ότι οι εικόνες έχουν εγγενώς δισδιάστατη δομή [Yan98].



Σχήμα A'.6: Παράδειγμα ενός Συνελικτικού Νευρωνικού Δικτύου: Πολλά επίπεδα τοποθετούνται το ένα μετά το άλλο και τα δεδομένα ρέουν προς μια μόνο κατεύθυνση (προς τα δεξιά). Το δίκτυο 'VGG-16' που παρουσιάστηκε το 2014. Το μοντέλο αυτό πέτυχε 92,7% top-5 ακρίβεια στον διαγωνισμό ταξινόμησης εικόνων Imagenet.

Στα Συνελικτικά Νευρωνικά Δίκτυα σκεφτόμαστε τους νευρώνες του κάθε επιπέδου ως οργανωμένους σε κυβική δομή και τα δεδομένα που εισέρχονται και εξέρχονται σε κάθε επίπεδο ως ένα σύνολο από διδιάστατους πίνακες. Οι διδιάστατοι πίνακες εισόδων και εξόδων ονομάζονται κανάλια ή χάρτες χαρακτηριστικών. Τα πιο βασικά είδη επιπέδων με τα οποία συντίθενται ΣΝΔ είναι: α) Το συνελικτικό επίπεδο, β) το επίπεδο χωρικής υποδειγματοληψίας και γ) το πλήρως διασυνδεδεμένο επίπεδο

Το συνελικτικό επίπεδο

Το συνελικτικό επίπεδο αποτελεί το σημαντικότερο δομικό συστατικό των ΣΝΔ, όπως άλλωστε το όνομά του υπονοεί.

Το συνελικτικό δίκτυο εξάγει N κανάλια εξόδου, από M κανάλια εισόδου, εκτελώντας την πράξη της συνέλιξης μεταξύ κάθε ενός από τα M κανάλια εισόδου με N φίλτρα. Κάθε φίλτρο έχει διαστάσεις $K \times K \times M$. Κάθε ένας από τους $K \times K$ διδιάστατους πίνακες ονομάζεται πυρήνας. Κάθε μια από τις $N \cdot K \cdot K \cdot M$ παραμέτρους των φίλτρων ονομάζεται βάρος. Κάθε νευρώνας του επιπέδου συνδέεται σε ένα κύβο νευρώνων από το προηγούμενο επίπεδο, μεγέθους $K \times K \times M$. Αν τα N κανάλια εξόδου έχουν μέγεθος $H \cdot H$ το καθένα, τότε για την εκτέλεση του συνελικτικού επιπέδου απαιτούνται $H \cdot H \cdot N \cdot K \cdot K \cdot M$ πολλαπλασιασμοί. Υποθέτοντας ότι τα κανάλια εισόδου έχουν τις ίδιες διαστάσεις με τα κανάλια εξόδου, η εμπρόσθια μεταφορά δεδομένων σε αυτά τα επίπεδα υπολογίζεται ως:

$$\begin{aligned} \forall n = 1 : N & \text{ (Πλήθος των καναλιών εξόδου)} \\ \forall i = 1 : R & \text{ (Πλήθος γραμμών ανά κανάλι)} \\ \forall j = 1 : C & \text{ (Πλήθος στηλών ανά κανάλι)} \\ F[n, i, j] &= b[n] + \sum_{m=1}^M \sum_{x=0}^{K-1} \sum_{y=0}^{K-1} \Phi[m, i+x, j+y] \cdot w[n, m, x, y] \end{aligned} \quad (A'.1)$$

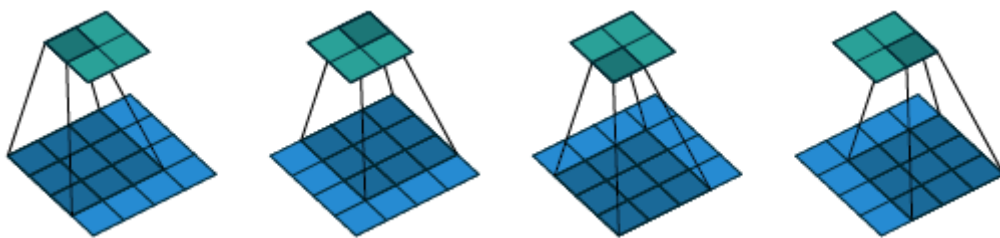
όπου

- F είναι ένας τανυστής που περιλαμβάνει τα κανάλια της εξόδου
- $b[n]$ είναι μια μεταβλητή πόλωσης που προστίθεται σε κάθε στοιχείο των καναλιών εξόδου n
- Φ είναι ένας τανυστής των καναλιών εισόδου
- w ένας τανυστής από φίλτρα που έμαθε το δίκτυο στο στάδιο της εκπαίδευσης

Το μέγεθος κάθε διαστάτου καναλιού εξόδου στη μια διάσταση μπορεί να υπολογιστεί ως:

$$H_{out} = \frac{H_{in} - K + 2 \cdot P}{S} + 1 \quad (A'.2)$$

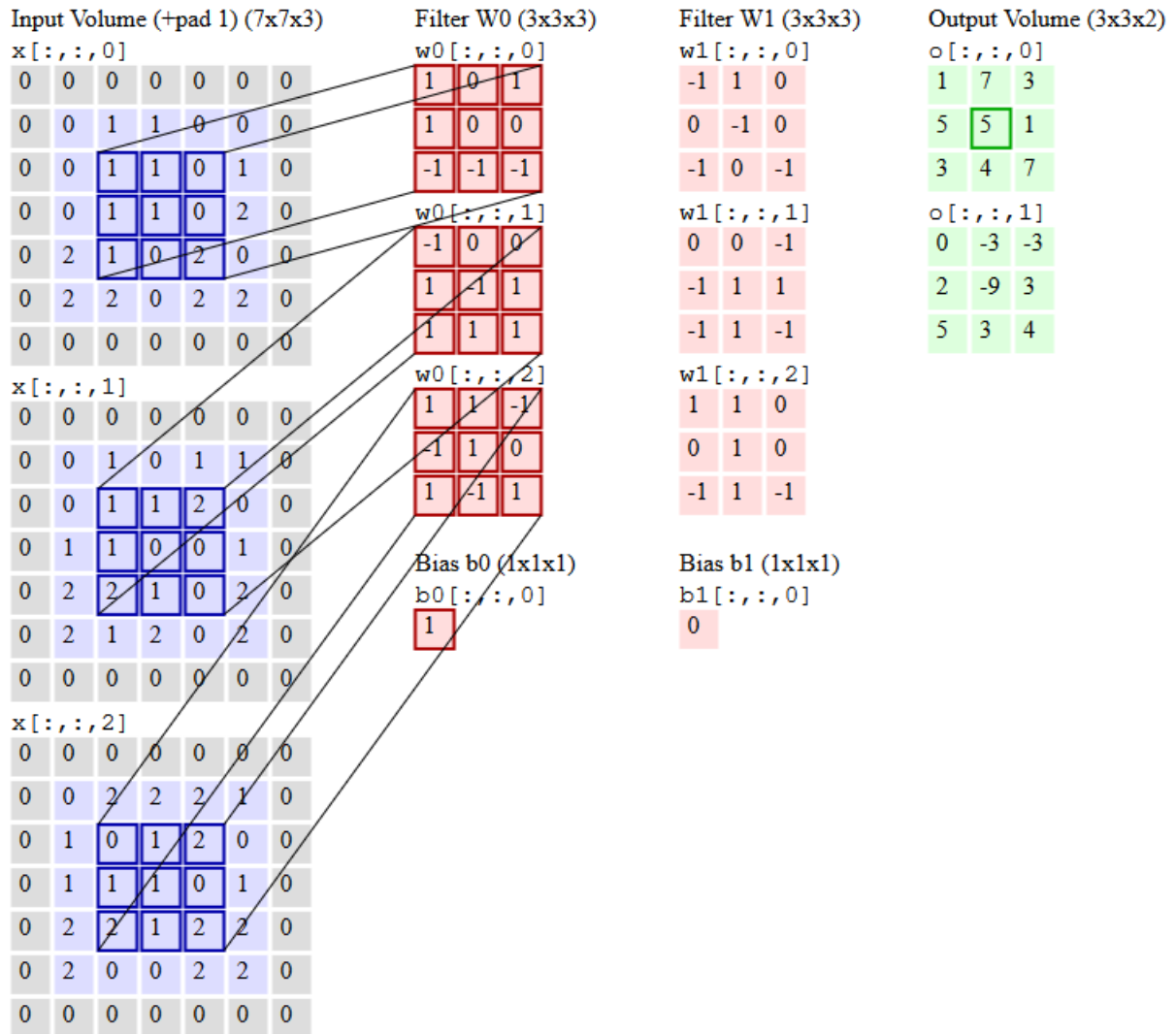
όπου H_{in} , το μέγεθος της μιας πλευράς του διαστάτου πίνακα εισόδου, P το πλήθος των μηδενικών στοιχείων με τα οποία θα μεγαλώσει το κανάλι εισόδου και χρησιμοποιείται για να ρυθμίσουμε το μέγεθος της εξόδου και S το βήμα με το οποίο ο διαστάτος $K \times K$ πυρήνας 'γλιστράει' επάνω στον πίνακα του καναλιού εισόδου.



Σχήμα A'.7: Η βασική πράξη του συνελκτικού επιπέδου. Ονομάζεται συνέλιξη, αλλά κατ' ουσίαν πρόκειται για διαδιάσταση ετεροσυσχέτιση. Ένας 3×3 πυρήνας γλιστράει πάνω από ένα 4×4 κανάλι εισόδου. $H_{in} = 4$, $K = 3$, $S = 1$, $P = 0$. Το κανάλι της εξόδου θα έχει διαστάσεις 2×2 . Πηγή: [Dum16]

Original	Gaussian Blur	Sharpen	Edge Detection
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

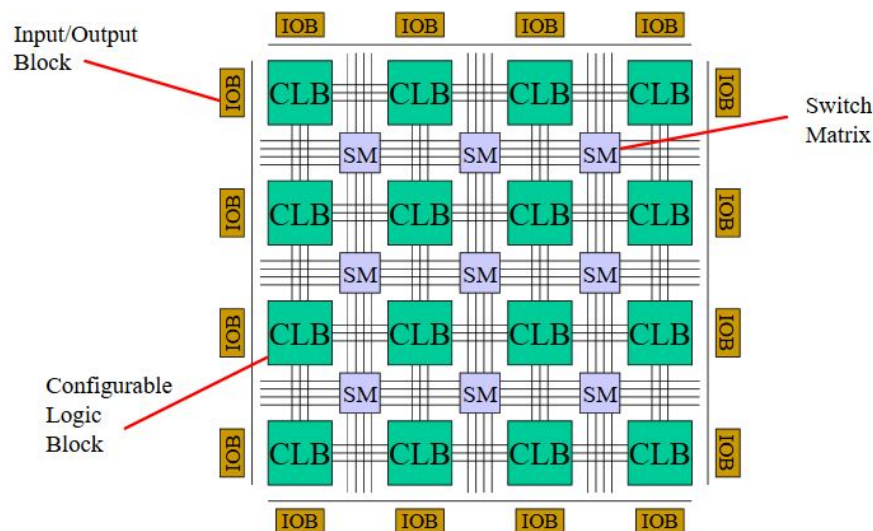
Σχήμα A'.8: Συνέλιξη μιας εικόνας με γνωστούς πυρήνες από το πεδίο της Όρασης Υπολογιστών: Blur, sharpen, edge detection. Στα Συνελκτικά Νευρωνικά Δίκτυα αντίθετα, οι πυρήνες είναι οι μεταβλητές που θα ρυθμιστούν για να προσεγγίσουμε τη συνάρτηση που αποτυπώνει την εικόνα εισόδου στην κατάλληλη κλάση εξόδου.



Σχήμα Α'9: Παράδειγμα της τρισδιάστατης συνέλιξης: Τρία κανάλια εισόδου συνελίσσονται με δύο φίλτρα, παράγοντας δύο κανάλια εξόδου. Πηγή: [Karb]

Α'.3 Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών (FPGA)

Τα FPGA είναι μια τεχνολογία επαναδιαμορφούμενων ολοκληρωμένων κυκλωμάτων. Το πλεονέκτημα αυτών των μονάδων είναι η δυνατότητα υλοποίησης εξειδικευμένων συστημάτων πολύ υψηλής παραλληλίας, επιτυγχάνοντας με αυτόν τον τρόπο επιδόσεις πολύ υψηλότερες από κοινές επεξεργαστικές μονάδες. Για τον προγραμματισμό τους χρησιμοποιούνται γλώσσες περιγραφής υλικού όπως η VHDL, καθώς επίσης και τεχνικές υψηλότερου επιπέδου αφαίρεσης από το υλικό (High Level Synthesis). Τα σύγχρονα FPGA περιλαμβάνουν πληθώρα από διαμορφούμενες λογικές μονάδες, καθώς επίσης και μονάδες εξειδικευμένων λειτουργιών όπως μνήμες SRAM, DSPs, Ethernet πομποδέκτες. Επιπρόσθετα, πλέον τοποθετούνται μαζί με μικροεπεξεργαστές όπως αυτοί της ARM, σε μια κοινή ψηφίδα, σχηματίζοντας ετερογενή Συστήματα σε Ψηφίδα (SoC) για να υποστηρίξουν και δυνατότητες λογισμικών υλοποιήσεων ταυτόχρονα με τους επιταχυντές υλικού.



Σχήμα Α'.10: Η προγραμματιζόμενη λογική των FPGA

Για μια ιστορική ανασκόπηση των FPGA και τη θέση τους στο σύμπαν των ημιαγωγών και των ψηφιακών συστημάτων παραπέμπουμε στο [Woo17]. Για μια σύγχρονη σύγκρισή τους με την τεχνολογία των GPU στο [Berb].

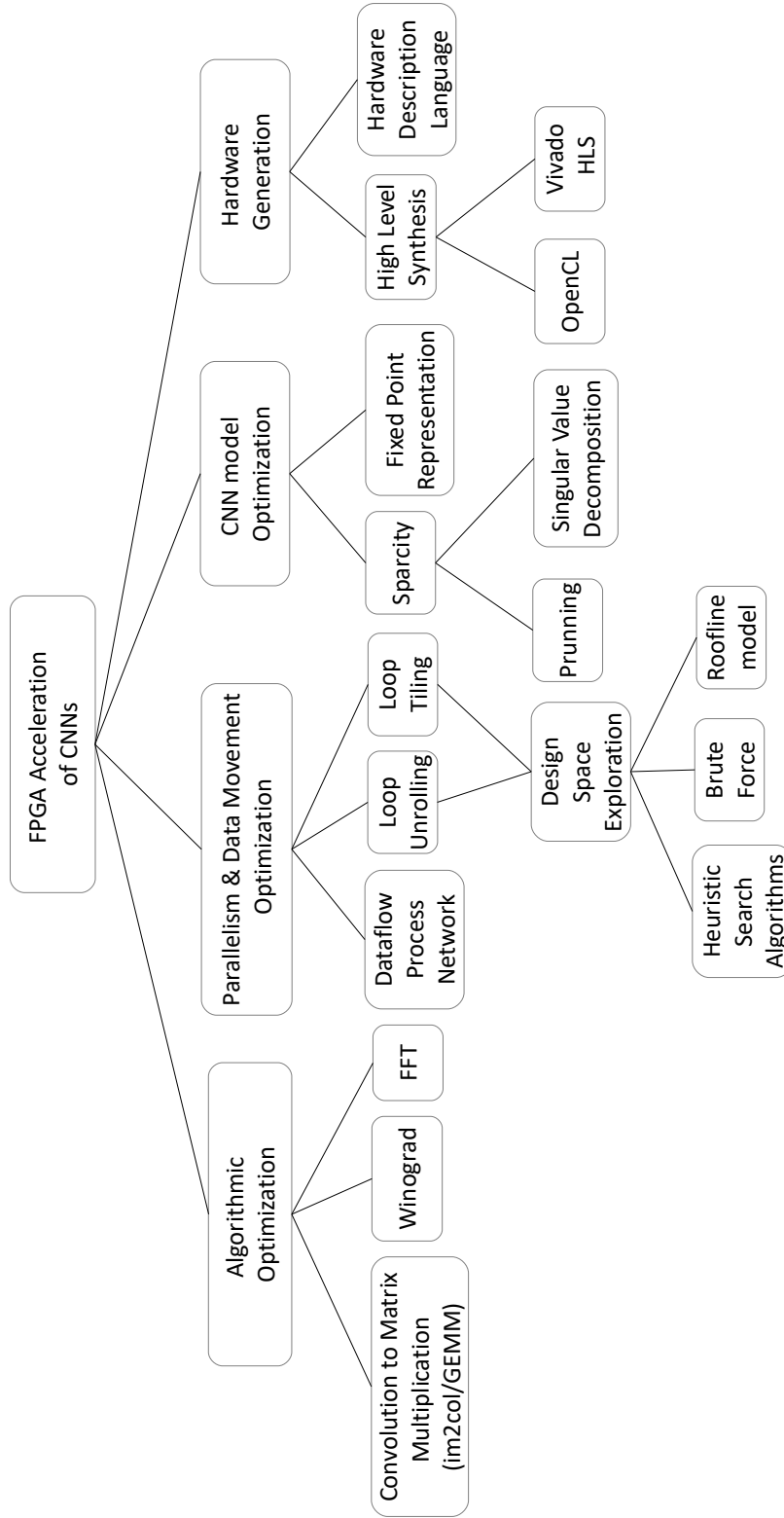
Α'.3.1 Συνελικτικά Νευρωνικά Δίκτυα σε FPGA

Επί του παρόντος, στις περισσότερες περιπτώσεις η εκπαίδευση των Τεχνητών Νευρωνικών Δικτύων πραγματοποιείται σε κάρτες γραφικών (GPU) και σε πολυύρηνους επεξεργαστές (CPU). Όσον αφορά την εμπρόσθια διάδοση των ΣΝΔ για την εκτέλεση της ταξινόμησης, αρκετές υλοποιήσεις και τρόποι σχεδίασης στα FPGA έχουν προταθεί τα τελευταία χρόνια. Η τεχνολογία των FPGA έχει αναδειχθεί ως μια εξαιρετική λύση για την υλοποίηση ΣΝΔ σε

ενσωματωμένα συστήματα. Ορισμένοι για αυτό λόγοι είναι:

- Τα Τεχνητά Νευρωνικά Δίκτυα είναι εγγενώς πολύ μεγάλης παραλληλίας υπολογιστικά συστήματα και αλγόριθμοι, χωρίς υπό συνθήκες διακλαδώσεις. Οι αρχιτεκτονικές FPGA σχεδιάζονται ως εξειδικευμένου τύπου υπολογιστικά συστήματα μεγάλης παραλληλίας. Από την άποψη της παραλληλίας τα χαρακτηριστικά των ΤΝΔ βρίσκονται σε σύμπτωση με τις δυνατότητες που προσφέρουν τα FPGA.
- Σε πολλές εφαρμογές των ΣΝΔ απαιτείται τα εμπλεκόμενα συστήματα να είναι μικρά σε μέγεθος, να έχουν χαμηλή κατανάλωση ενέργειας και γρήγορους χρόνους απόκρισης. Τα FPGA προσφέρουν ντετερμινιστικό υπολογιστικό χρόνο και χαμηλή κατανάλωση ενέργειας.
- Στις μέρες μας υπάρχει μεγάλο πεδίο έρευνας γύρω από τα ΣΝΔ και νέες αρχιτεκτονικές προτείνονται συνεχώς. Οι αρχιτεκτονικές FPGA μπορούν να επαναδιαμορφωθούν ώστε να συμπεριλαμβάνουν στη λειτουργία τους τις νέες τεχνικές των εκάστοτε καλύτερων ΣΝΔ.
- Τα ΣΝΔ έχει αποδειχθεί ότι μπορούν να δουλέψουν καλά με χαμηλής αριθμητικής ακρίβειας πράξεις και δεδομένα. Οι αρχιτεκτονικές FPGA σχεδιάζονται με προσαρμοσμένο μέγεθος datapath για την κάθε εφαρμογή. Με αυτό τον τρόπο απαιτείται λιγότερη μνήμη και μπορεί να επιτευχθεί μεγαλύτερης πυκνότητας λογική στο τσιπ, το οποίο οδηγεί σε αύξηση των παράλληλων υπολογιστικών μονάδων και συνεπώς σε γρηγορότερη εκτέλεση του ΣΝΔ, χωρίς σημαντικό κόστος στην ακρίβεια ταξινόμησης που αυτό επιτυγχάνει.

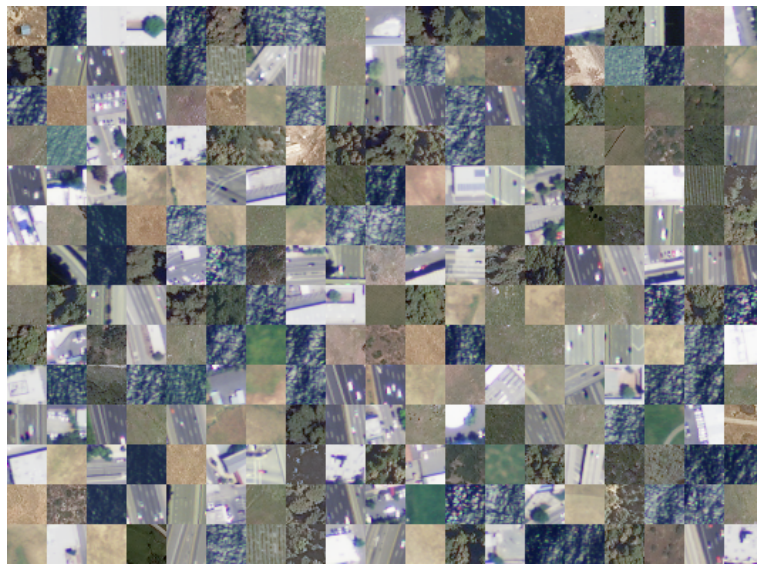
Για το πρόβλημα της αποτύπωσης αρχιτεκτονικών ΣΝΔ σε αρχιτεκτονικές FPGA έχουν προταθεί πλήθος προσεγγίσεων. Πρόσφατα μερικές ανασκοπήσεις των προτεινόμενων προσεγγίσεων δημοσιεύτηκαν [Abd18a], [Guo17], [Zha18] . Συνοψίζουμε τα βασικά τους ευρήματα ως προς τις τεχνικές που εφαρμόζονται στην επόμενη εικόνα.



Σχήμα Α.11: Οι βασικές προσεγγίσεις για την επιτάχυνση των ΣΝΔ στα FPGA.

Α'.4 Εκπαίδευση ΣΝΔ και Βελτιστοποίηση

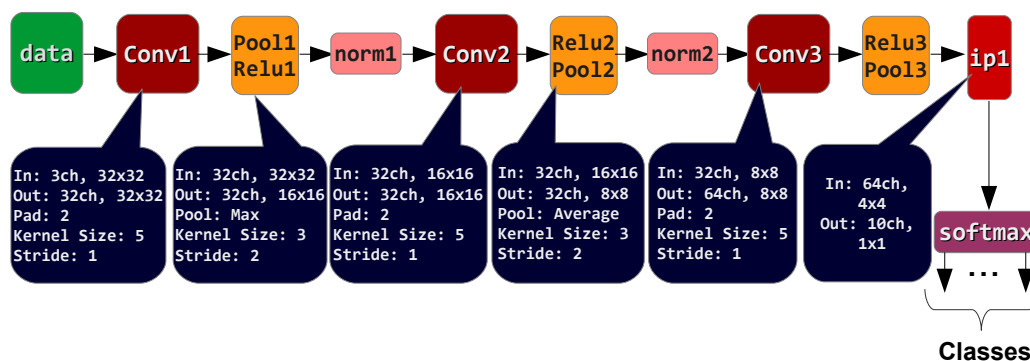
Στην συγκεκριμένη εργασία εκπαιδεύσαμε ένα ΣΝΔ για ταξινόμηση δορυφορικών εικόνων. Χρησιμοποιήσαμε το ελεύθερα προσβάσιμο σύνολο δεδομένων SAT-6 airborne dataset [Bas15], το οποίο αποτελείται από 324.000 εικόνες εκπαίδευσης και 81.000 εικόνες ελέγχου. Οι εικόνες αυτές είναι κομμάτια των 28×28 εικονοστοιχείων, με ένα μέτρο χωρική ακρίβεια ανά εικονοστοιχείο, που έχουν εξαχθεί από υψηλής ανάλυσης δορυφορικές εικόνες από το NAIP dataset. Περιλαμβάνουν έξι κατηγορίες κάλυψης γης: άγονη γη, δέντρα, λιβάδια, δρόμους, κτήρια και υδάτινες μάζες.



Σχήμα Α'.12: Παραδείγματα εικόνων από το SAT-6 airborne dataset.

Για την εκπαίδευση του ΣΝΔ χρησιμοποιήσαμε το Caffe framework. Το Caffe είναι ένα εργαλείο για βαθιά μάθηση που αναπτύχθηκε από το Berkeley AI Research και από συνεισφέροντες της κοινότητας της μηχανικής μάθησης.

Το ΣΝΔ που εκπαιδεύσαμε βασίστηκε στη δομή και στους κανόνες μάθησης του Cifar-10 Full CNN, σχεδιασμένο από τον Alex Krizhevsky. Το αρχικό δίκτυο στόχευε στην ταξινόμηση εικόνων 32×32 εικονοστοιχείων σε δέκα κλάσεις. Η δομή του δικτύου φαίνεται στην εικόνα Α'.13. Το συγκεκριμένο μοντέλο επιτυγχάνει υψηλή ακρίβεια ταξινόμησης στα δεδομένα μας, ενώ παράλληλα χωράει στην on-chip μνήμη του Xilinx Zynq Z-7020 SoC. Ταυτόχρονα έχει μικρότερες υπολογιστικές απαιτήσεις από ΣΝΔ περισσότερων επιπέδων. Τα χαρακτηριστικά αυτά το καθιστούν ιδιαίτερα ταιριαστό μοντέλο για την περίπτωσή μας.



Σχήμα Α'.13: Η δομή του Cifar-10 Full ΣΝΔ.

Α'.4.1 Προσαρμογή του ΣΝΔ μοντέλου στην περίπτωση μας

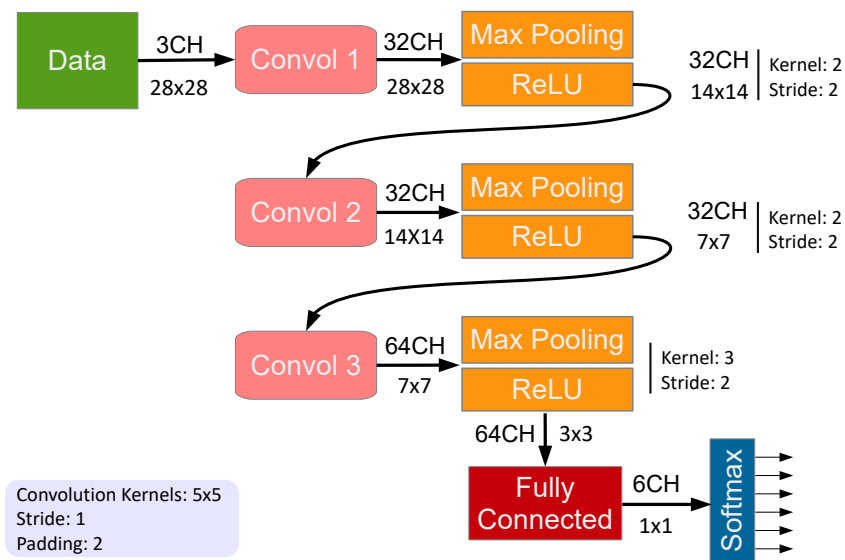
- **Αλλαγή των εισόδων και εξόδων σύμφωνα με το SAT-6 dataset** Αυτή είναι μια προφανής αλλαγή, το αρχικό μοντέλο στόχευε σε εικόνες 32×32 με δέκα κλάσεις, ενώ το δικό μας σε εικόνες 28×28 έξι κλάσεων.
- **Αφαίρεση του καναλιού NIR από τα δεδομένα εισόδου.** Τόσο σε προηγούμενες εργασίες από τη βιβλιογραφία ([Pap16]), όσο και μέσα από τη δικιά μας δοκιμή, επιβεβαιώσαμε ότι το συγκεκριμένο ΣΝΔ, στο συγκεκριμένο σύνολο δεδομένων λειτουργεί εξαιρετικά και χωρίς το συγκεκριμένο κανάλι εισόδου. Από την άποψη της σχεδίασης του υλικού αυτό είναι βολικό και σημαντικό: Λιγότερα δεδομένα στην είσοδο σημαίνουν λιγότερη απαιτούμενη μνήμη και μικρότερο πλήθος υπολογισμών. Ειδικά για τα ΣΝΔ τα οποία έχουν μεγάλη υπολογιστική πολυπλοκότητα η μείωση των διαστάσεων εισόδου είναι πολύ σημαντική.
- **Αφαίρεση του Normalization επιπέδου.** Τα τελευταία χρόνια η συνεισφορά αυτών των επιπέδων στην ακρίβεια του δικτύου αμφισβητείται και στα πιο πρόσφατα ΣΝΔ έχει πάψει η χρήση του ([Karc], [Che16a]). Στην δικιά μας περίπτωση πράγματι η αφαίρεσή του δεν είχε καμία επίπτωση στην επιτυγχανόμενη ακρίβεια. Συνεπώς το θεωρήσαμε περιττό να υλοποιηθεί το συγκεκριμένο είδος επιπέδων σε μια σχεδίαση υλικού. Επιπρόσθετα πρέπει να σημειώσουμε ότι τα Normalization επίπεδα έχουν λιγότερες Multiply-Accumulate πράξεις από τα συνελικτικά επίπεδα, πραγματοποιούν όμως διαιρέσεις με μη-σταθερούς αριθμούς, το οποίο είναι υπολογιστικά ακριβό.
- **Χρήση μόνο του max pooling για την χωρική υποδειγματοληψία των καναλιών εισόδου.** Στο αρχικό μοντέλο τα επίπεδα που εκτελούν την χωρική υποδειγματοληψία, χρησιμοποιούν εναλλάξ το μέγιστο του κάθε πυρήνα και το μέσο όρο του πυρήνα ως πράξεις. Για να έχουμε ένα περισσότερο ενιαίο δίκτυο αποφασίσαμε αυθαίρετα να χρησιμοποιήσουμε μόνο την πράξη του μεγίστου.

- **Αλλαγή των υπερ-παραμέτρων των επιπέδων υποδειγματοληψίας για την επίτευξη συμμετρικών διαστάσεων εξόδου.** Το Caffe framework δεν υπολογίζει τις διαστάσεις εξόδου των συνελκτικών επιπέδων και των επιπέδων υποδειγματοληψίας με τον ίδιο ακριβώς τρόπο. Στην περίπτωση του συνελκτικού επιπέδου στρογγυλοποιεί προς τα κάτω, ενώ στο επίπεδο υποδειγματοληψίας στρογγυλοποιεί προς τα πάνω. Η μια διάσταση κάθε διδιάστατου πίνακα εξόδου υπολογίζεται ως

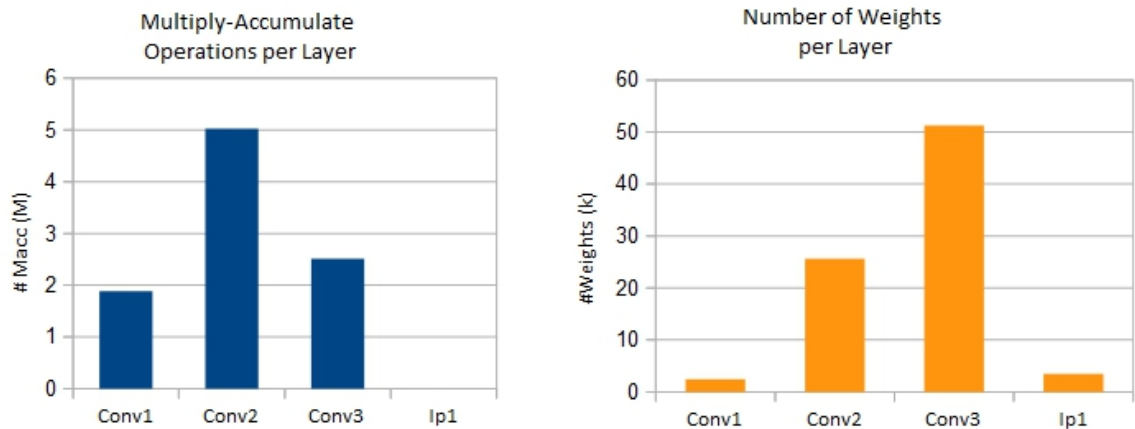
$$\text{Pooling Output_height} = \text{ceil}\left(\frac{\text{Input_height} - \text{Kernel_height}}{\text{Stride}} + 1\right)$$

Όταν το παραπάνω κλάσμα δεν είναι ακέραιος, το Caffe υλοποιεί την στρογγυλοποίηση των διαστάσεων προς τα πάνω προσθέτοντας μηδενικά στις μισές από τις ακμές κάθε καναλιού εισόδου.

Στις εικόνες που ακολουθούν αποτυπώνονται χαρακτηριστικά της αρχιτεκτονικής του "Modified Cifar-10 Full" ΣΝΔ που χρησιμοποιούμε.



Σχήμα Α.14: Το "Modified Cifar-10 Full" ΣΝΔ.



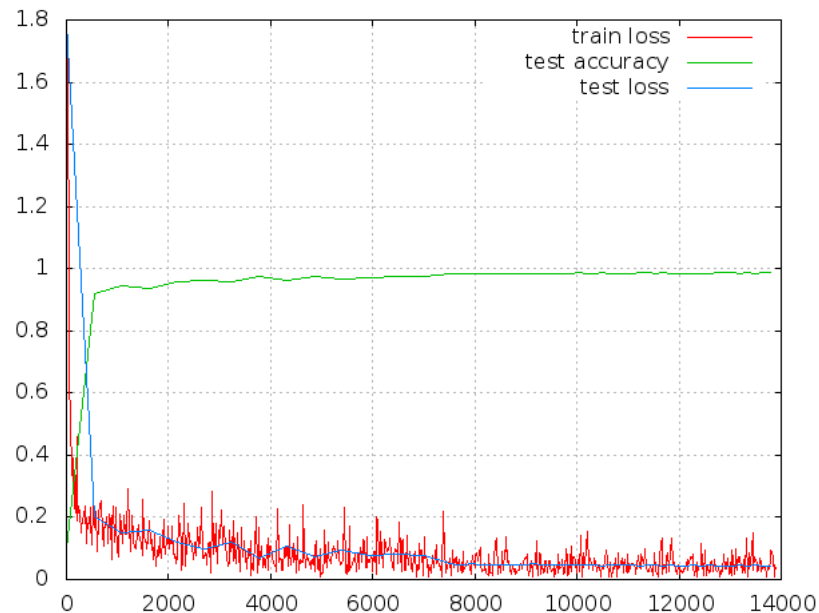
Σχήμα Α'.15: Το πλήθος των πράξεων και των βαρών ανά επίπεδο στο "Modified Cifar-10 Full".

Name	Type	Channels In	Dimension In	Channels Out	Dimensions Out	Operations	Memory
data	Input	3	28x28	3	28x28		activation: 2.35k
conv1	Convolution	3	28x28	32	28x28	Macc: 1.88M	activations: 25.09k parameters: 2.43k
pool1	Max Pooling	32	28x28	32	14x14	Comp: 25.09k	activations: 6.27k
relu1	ReLU	32	14x14	32	14x14	Comp: 6.27k	activations: 6.27k
conv2	Convolution	32	14x14	32	14x14	Macc: 5.02M	activations: 6.27k parameters: 25.63k
pool2	Max Pooling	32	14x14	32	7x7	Comp: 6.27k	activations: 1.57k
relu2	ReLU	32	7x7	32	7x7	Comp: 1.57k	activations: 1.57k
conv3	Convolution	32	7x7	64	7x7	Macc: 2.51M	activations: 3.14k parameters: 51.26k
pool3	Max Pooling	64	7x7	64	3x3	Comp: 5.18k	activations: 576
relu3	ReLU	64	3x3	64	3x3	Comp: 576	activations: 576
ip1	Inner Product	64	3x3	6	1x1	Macc: 3.46k	activations: 6 parameters: 3.46k
prob	Softmax	6	1x1	6	1x1	add: 6 div: 6 exp: 6	activation: 6
TOTAL						Macc: 9.41M Comp: 44.96k	activations: 53.69k parameters: 82.79k

Σχήμα Α'.16: Ανάλυση των επιπέδων του "Modified Cifar-10 Full" ΣΝΔ.

A'.4.2 Αποτελέσματα εκπαίδευσης του Modified Cifar-10 Full ΣΝΔ

Το SAT-6 σύνολο δεδομένων είναι χωρισμένο εκ των προτέρων σε σύνολο εκπαίδευσης και σύνολο ελέγχου. Έχοντας κατά νου ότι το συγκεκριμένο σύνολο δεδομένων δημοσιεύτηκε με σκοπό αλγόριθμους της μηχανικής μάθησης, δεν πραγματοποιήσαμε καθαρισμό, επαύξηση ή κάποιο μετασχηματισμό των δεδομένων. Το μόνο στάδιο προ-επεξεργασίας που παρεμβάλλεται ανάμεσα στα δεδομένα εικόνας και το ΣΝΔ είναι ο ανά κανάλι υπολογισμός του μέσου όρου όλων των εικόνων και η αφαίρεσή του. Όσον αφορά τον αλγόριθμο εκμάθησης και τις υπερ-παραμέτρους που πρέπει να ρυθμιστούν για την εκπαίδευση, ακολουθήσαμε το παράδειγμα του Cifar-10 Full ΣΝΔ. Εντός έξι εποχών εκπαίδευσης το δίκτυο πέτυχε top-1 ακρίβεια ταξινόμησης 98,8%. Στην επόμενη εικόνα αποτυπώνουμε τη διαδικασία της εκπαίδευσης. Συγκριτικά αποτελέσματα με άλλες μεθόδους από τη βιβλιογραφία, επί του ίδιου συνόλου δεδομένων παρουσιάζουμε στην εικόνα A'.18.



Σχήμα A'.17: Training loss and accuracy of the "Modified Cifar-10 Full" CNN model, for the SAT-6 dataset after 4 epochs of training.

Method	Overall Accuracy %	
	SAT-4	SAT-6
DBN [Bas15]	81.78	76.41
CNN [Bas15]	86.83	79.06
SDAE [Bas15]	79.98	78.43
Semi-Supervised [Bas15]	97.95	93.92
Pretrained-AlexNet [Vak15]	99.46	99.57
AlexNet [Pap16]	99.98	99.93
AlexNet-small [Pap16]	99.86	99.90
VGG [Pap16]	99.98	99.98
"Modified Cifar-10 Full" CNN model (this Thesis)	99.09	98.8

Σχήμα Α'.18: Συγκριτικά αποτελέσματα στην ακρίβεια ταξινόμησης των δεδομένων του SAT-6 airborne dataset με μεθόδους απ' τη βιβλιογραφία.

A'.4.3 Συμπιέζοντας το ΣΝΔ: Βελτιστοποίηση μήκους λέξης

Το Caffe framework, όπως και άλλα εργαλεία εκπαίδευσης δικτύων, χρησιμοποιούν αριθμητική ακρίβεια 32-βιτ κινητής υποδιαστολής. Ωστόσο οι πράξεις με σταθερή υποδιαστολή απαιτούν λιγότερους πόρους και ταιριάζουν καλύτερα σε αρχιτεκτονικές FPGA. Έχει επίσης αναδειχθεί τα τελευταία χρόνια ότι τα ΣΝΔ μπορούν να λειτουργήσουν καλά και με λίγων ψηφίων αριθμητική σταθερής υποδιαστολής [Gys16], [Cou14]. Σε αυτή τη διπλωματική εργασία χρησιμοποιούμε το Ristretto, μια επέκταση του Caffe για τον προσδιορισμό της επίδρασης του μήκους της λέξης στην ακρίβεια του ΣΝΔ μας. Χρησιμοποιήθηκε το σενάριο δυναμικής σταθερής υποδιαστολής: Εντός κάθε επιπέδου του ΣΝΔ η υποδιαστολή είναι σταθερή, αλλά μπορεί να μεταβάλλεται από επίπεδο σε επίπεδο. Όλα τα βάρη του ΣΝΔ χρησιμοποιούν το ίδιο πλήθος από bit και όλες οι τιμές των καναλιών εισόδου και εξόδου έχουν το ίδιο μήκος bit.

Τα αποτελέσματα της διερεύνησης αποτυπώνονται στην εικόνα A'.19 και το σενάριο που επιλέχθηκε για την παρούσα διπλωματική στην εικόνα A'.20.

Scenario	A	B	C	D	E
Group					
Conv Weights	8 bit	8 bit	8 bit	4 bit	Power-of-2
FC Weights	2 bit	2 bit	2 bit	2 bit	Power-of-2
Activations	32 bit	8 bit	4 bit	4 bit	8 bit
Accuracy %	0.9792	0.9783	0.9489	0.7588	0.9432

Σχήμα A'.19: Διαφορετικά σενάρια μήκους λέξης για το Modified Cifar-10 Full ΣΝΔ.

Scenario C	Conv1	Conv2	Conv3	FC
Bw Layer In	4	4	4	4
Bw Layer Out	4	4	4	4
Bw Weights	8	8	8	2
FL Layer In	-4	-3	-3	-3
FL Layer Out	-4	-5	-5	0
FL Weights	8	10	10	7

Σχήμα A'.20: Το σενάριο δυναμικής σταθερής υποδιαστολής που επιλέξαμε για το Modified Cifar-10 Full ΣΝΔ.

Α'.5 Υλοποίηση σε FPGA

Α'.5.1 Προσέγγιση

Τα Συνελικτικά Νευρωνικά Δίκτυα είναι εγγενώς πολύ μεγάλης παραλληλίας υπολογιστικά συστήματα και αλγόριθμοι. Εξετάζοντας το πρώτο μόνο συνελικτικό επίπεδο του Modified Cifar-10 Full παρατηρούμε ότι αυτό αποτελείται από 1,9 εκατομμύρια πράξεις πολλαπλασιασμού-συσσώρευσης, οι οποίες είναι όλες ανεξάρτητες η μια από την άλλη. Τα σύγχρονα βαθιά ΣΝΔ απαιτούν τόσο μεγάλο αριθμό υπολογιστικών πόρων, που στην πλειοψηφία των περιπτώσεων οι πόροι των FPGA δεν επαρκούν για την πλήρη εκτύλιξη της παραλληλίας τους. Για αυτό το λόγο και σε συνδυασμό με τα μεγάλα ποσά μνήμης που απαιτούνται, η πλέον συνηθισμένη τεχνική αποτύπωσης ΣΝΔ σε FPGA είναι με τη χρήση εξωτερικής μνήμης για αποθήκευση μέρους των δεδομένων και ενός κεντρικού επεξεργαστή ο οποίος συντονίζει το ποια δεδομένα βρίσκονται κάθε στιγμή εντός του FPGA προς επεξεργασία. Οι υπολογιστικοί πόροι του FPGA χρησιμοποιούνται σε αυτή την περίπτωση μόνο για την επιτάχυνση των πράξεων του πολλαπλασιασμού και συσσώρευσης [Che16b; Qiu16; Zha15]. Στην παρούσα διπλωματική ακολουθήσαμε ωστόσο μια διαφορετική προσέγγιση: Οι απαιτήσεις σε μνήμη του Modified Cifar-10 Full ΣΝΔ μπορούν να καλυφθούν από την on-chip μνήμη της προγραμματιζόμενης λογικής. Σε αυτή λοιπόν την περίπτωση η χρήση εξωτερικής μνήμης και CPU καθίσταται περιττή και το δίκτυο μπορεί να εκτελεσθεί εξ' ολοκλήρου στο FPGA. Κατά αυτό τον τρόπο αποφεύγουμε τα προβλήματα που εμφανίζονται σχετικά με το πότε και ποια δεδομένα θα μεταφερθούν από και προς την εξωτερική μνήμη [Zha15]. Ωστόσο ταυτόχρονα τίθεται περιορισμός στο μέγιστο μέγεθος εικόνας που η υλοποίησή μας μπορεί να υποστηρίξει και στο μέγιστο μέγεθος ΣΔΝ. Σε κάθε περίπτωση, τόσο στις υλοποιήσεις που έχουν προταθεί στη βιβλιογραφία όσο και στη δικιά μας, ενδιαφερόμαστε για την αύξηση της αποτυπωμένης παραλληλίας και την εύρεση της καταλληλότερης διαμόρφωσης του συστήματος, που θα διαμοιράζει με τέτοιο τρόπο τους υπολογιστικούς πόρους του συστήματος ώστε να επιτυγχάνει την καλύτερη απόδοση.

Υπενθυμίζουμε ότι ένα συνελικτικό επίπεδο L λαμβάνει M κανάλια εισόδου, διαστάσεων $H_{in} \times H_{in}$ και παράγει N κανάλια εξόδου διαστάσεων $H_{out} \times H_{out}$, εκτελώντας N συνελίξεις με N φίλτρα διαστάσεων $M \times K \times K$ το καθένα. Στην υλοποίηση ΣΝΔ σε FPGA εμφανίζονται οι εξής δυνατότητες παραλληλίας:

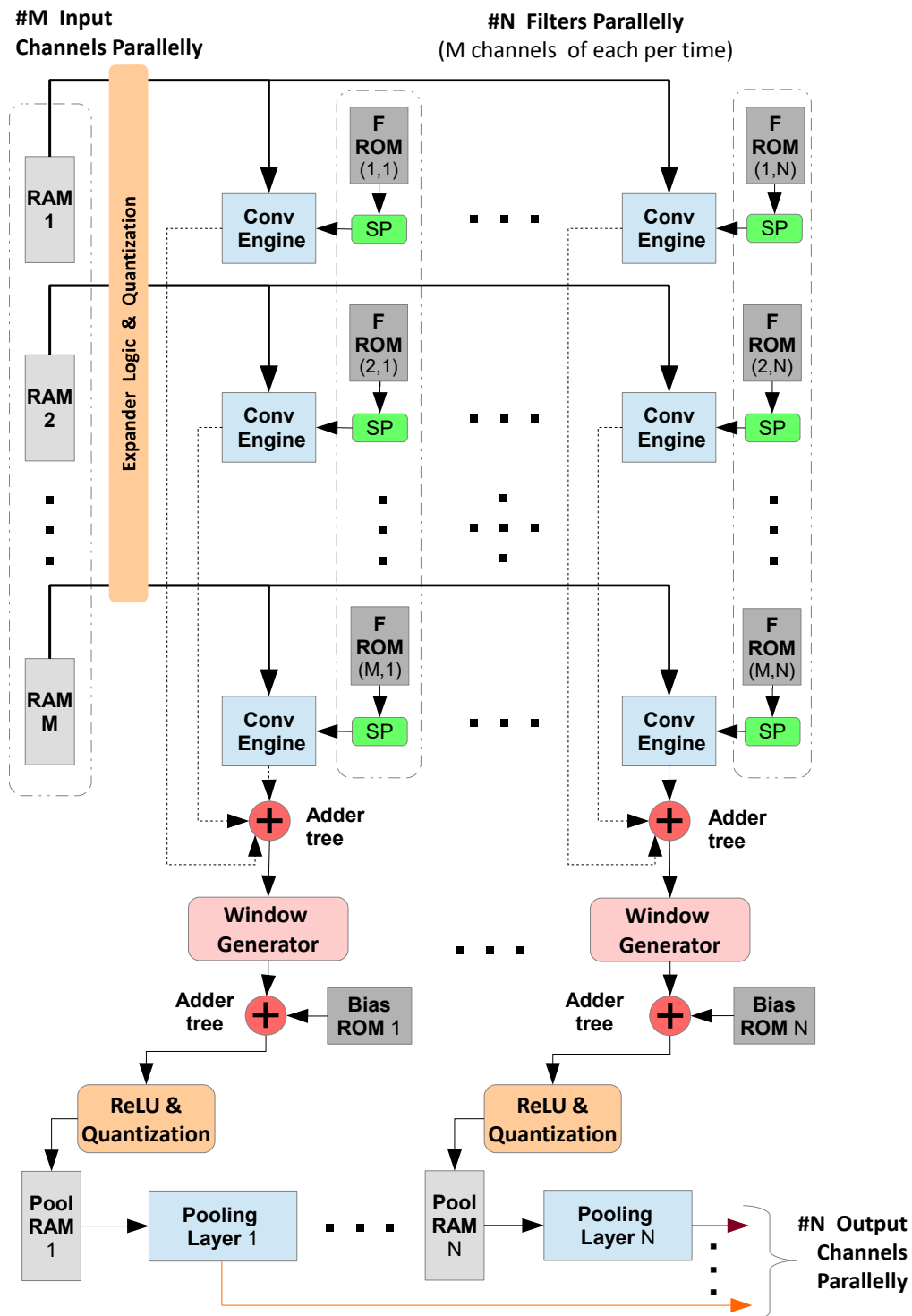
- Στην περίπτωση επεξεργασίας μίας εικόνας:
 1. Η παράλληλη λειτουργία των διαδοχικών επιπέδων. Το $L+1$ επίπεδο δεν χρειάζεται να αναμένει να ολοκληρωθεί πλήρως η λειτουργία του L επιπέδου. Αντίθετα το $L+1$ επίπεδο μπορεί να επεξεργάζεται τα δεδομένα που το L επίπεδο παράγει, στον ρυθμό με τον οποίο αυτά παράγονται.
 2. Ένα συνελικτικό επίπεδο L μπορεί να επεξεργαστεί παράλληλα όλα τα M κανάλια εισόδου που του αντιστοιχούν, εισάγοντας ένα εικονοστοιχείο από κάθε κανάλι ανά κύκλο ρολογιού. (Σύνολο M εικονοστοιχεία ανά κύκλο.)
 3. Ένα συνελικτικό επίπεδο L μπορεί να παράγει παράλληλα όλα τα N κανάλια εξόδου παράλληλα, ένα εικονοστοιχείο από το καθένα ανά κύκλο ρολογιού. (Σύνολο N εικονοστοιχεία ανά κύκλο.)
 4. Ένα συνελικτικό επίπεδο L μπορεί να επεξεργάζεται παράλληλα όλα τα $H_{in} \times H_{in}$ εικονοστοιχεία ενός καναλιού εισόδου του, ώστε να παράγει παράλληλα όλα τα $H_{out} \times H_{out}$ εικονοστοιχεία στην έξοδο.
 5. Σε ένα συνελικτικό επίπεδο ένας πυρήνας αποτελείται από $K \times K$ βάρη και συνεχώς απαιτούνται $K \times K$ πολλαπλασιασμοί, οι οποίοι μπορούν να υπολογιστούν παράλληλα.
- Στην περίπτωση ροής εικόνων τα διαφορετικά επίπεδα του δικτύου μπορούν να λειτουργήσουν pipelined. Σε αυτή την περίπτωση δεν χρειάζεται να περιμένουμε να εκτελεστούν όλοι οι υπολογισμοί για μια εικόνα μέχρι να εισάγουμε την επόμενη. Τα διαφορετικά διαδοχικά επίπεδα πρέπει όμως να αποτυπωθούν παράλληλα στο υλικό.
- Παράλληλη υλοποίηση πολλαπλών, ίδιων, ανεξάρτητων ΣΝΔ σε ένα FPGA για τον καταμερισμό του φόρτου ταξινόμησης πολλών εικόνων. Αυτή είναι μια επιλογή που το υλικό που έχουμε στη διάθεσή μας ακόμα δεν φαίνεται να την επιτρέπει για τα σύγχρονου μεγέθους ΣΝΔ.

Στην παρούσα διπλωματική εργασία από τις παραπάνω παραλληλίες επιχειρήσαμε να εκτυλίξουμε τις (2), (3), (5) και την pipelined λειτουργία των επιπέδων για ροή εικόνων. Για να επιτύχουμε την pipelined λειτουργία των επιπέδων για ροή εικόνων, αποτυπώνουμε κάθε επίπεδο του ΣΝΔ σε δικούς του υπολογιστικούς πόρους και χρησιμοποιούμε ανάμεσα στα επίπεδα την τεχνική του ping-pong double buffering. Για τα (2) και (3) οργανώνουμε κάθε συνελικτικό επίπεδο L του δικτύου ως ένα διδιάστατο πλέγμα διαστάσεων $M_{parallel} \times N_{parallel}$, το οποίο περιέχει $M_{parallel} \times N_{parallel}$ συνελικτικές μηχανές. Για το (5) χρησιμοποιούμε ως βασική συνελικτική μονάδα μια pipelined συνελικτική μηχανή από τη βιβλιογραφία.

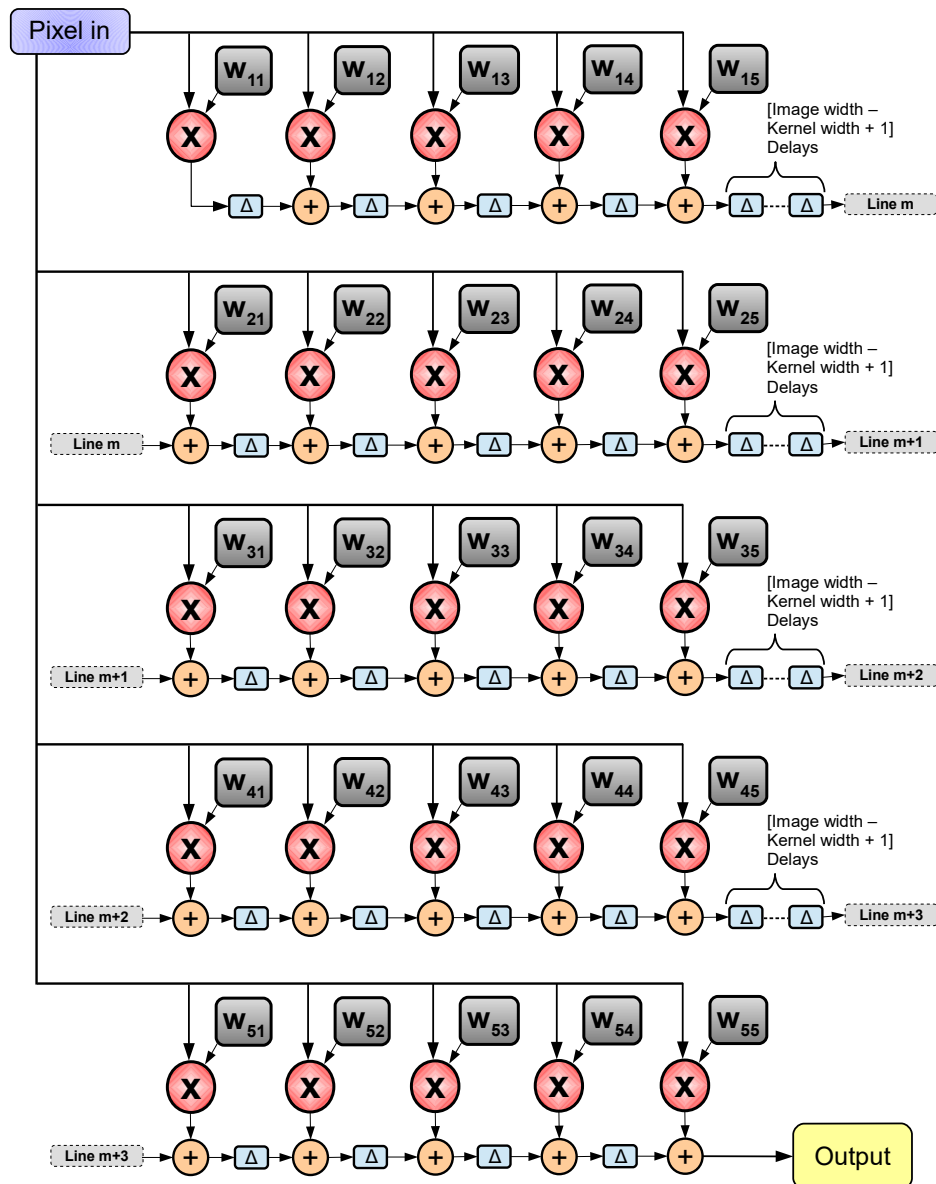
Όλος ο VHDL κώδικας της παρούσας εργασίας είναι πλήρως παραμετροποιήσιμος από ένα ενιαίο αρχείο διαμόρφωσης που περιλαμβάνει τα μήκη των λέξεων σε bit, το μέγεθος της εικόνας, την παραλληλία στις εισόδους και τις εξόδους των επιπέδων, διάφορες υπερ-παραμέτρους των βασικών πράξεων (το μέγεθος των πυρήνων, το πλήθος των μηδενικών επέκτασης, το βήμα της συνέλιξης, κτλ). Το στάδιο της επαλήθευσης της σχεδίασης βασίστηκε σε κατευθυνόμενα τεστ και σε επόπτευση των αποτελεσμάτων των προσομοιώσεων στο περιβάλλον εργασίας Vivado της Xilinx. Τα αποτελέσματα συγκρίθηκαν ως προς την αριθμητική τους ακρίβεια με αποτελέσματα από το Matlab.

A'.5.2 Αρχιτεκτονική FPGA - Τα βασικά στοιχεία

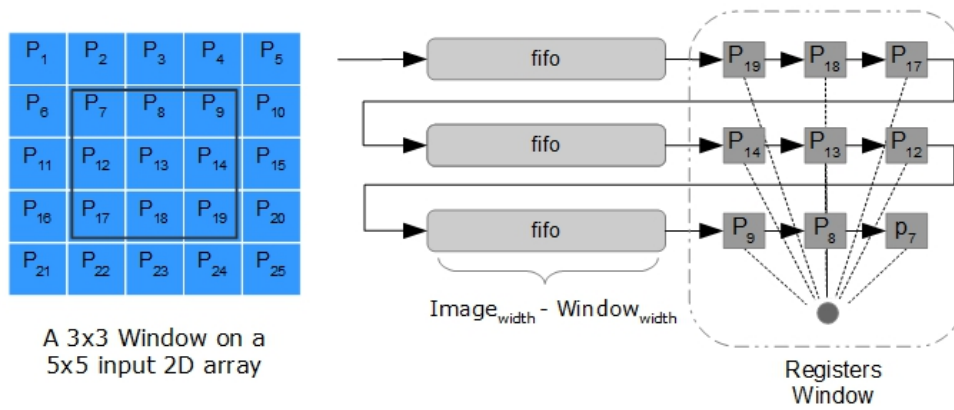
Η δομή ενός ενοποιημένου επιπέδου της αρχιτεκτονικής μας απεικονίζεται στην εικόνα A'.21. Ως ενοποιημένο επίπεδο ονομάζουμε ένα επίπεδο που εκτελεί την πράξη της συνέλιξης (convolution), την πράξη της χωρικής υποδειγματοληψίας (pooling), την επέκταση των καναλιών με μηδενικές τιμές (zero padding), τις αθροίσεις των αποτελεσμάτων, την πράξη Relu, καθώς και την περικοπή της ακρίβειας των ψηφίων εισόδου και εξόδου. Το επίπεδο λαμβάνει στην είσοδό του ένα εικονοστοιχείο σε κάθε κύκλο ρολογιού, για κάθε ένα από τα $M_{parallel}$ παράλληλα κανάλια εισόδου, δηλαδή συνολικά $M_{parallel}$ εικονοστοιχεία στην είσοδό του ανά κύκλο ρολογιού. Για να ολοκληρωθεί η επεξεργασία όλων των M καναλιών εισόδου και να παραχθούν όλα τα N κανάλια εξόδου του επιπέδου, χρησιμοποιείται πολυπλεξία στον χρόνο, τόσο στα κανάλια εισόδου, όσο και στα φίλτρα. Αν και δεν απεικονίζεται, βασικό στοιχείο της αρχιτεκτονικής είναι ο τρόπος με τον οποίο συνεργάζονται και συντονίζονται τα επιμέρους μέρη (control). Αυτό αποτελεί ένα από τα πιο δύσκολα σημεία στη σχεδίαση υλικού με γλώσσες περιγραφής υλικού, όπως η VHDL. Για την επίτευξή του απαιτείται εξαιρετική προσοχή στην λεπτομέρεια.



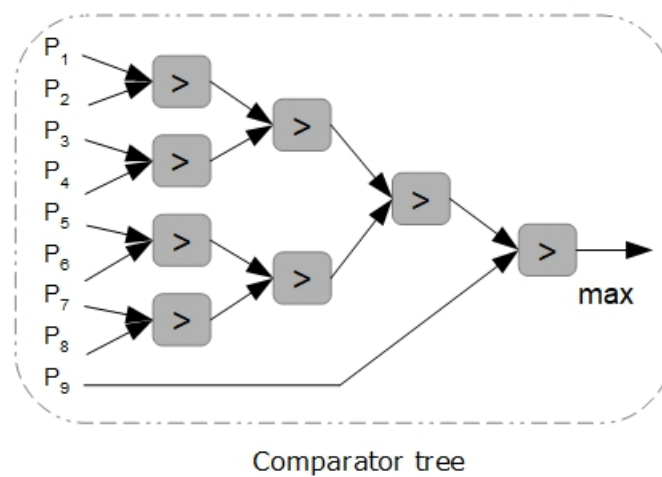
Σχήμα Α'.21: Η αρχιτεκτονική του ενοποιημένου επιπέδου. Εκτελεί τις πράξεις της συνέλιξης, της επέκτασης των καναλιών εισόδου με μηδενικές τιμές (zero padding), την περικοπή των bit στις εισόδους και εξόδους, την ReLU και την χωρική υποδειγματοληψία βασισμένη στη συνάρτηση μεγίστου (max pooling). Επιμέρους τέτοια επίπεδα συνδέονται για να σχηματίσουν το ΣΝΔ.



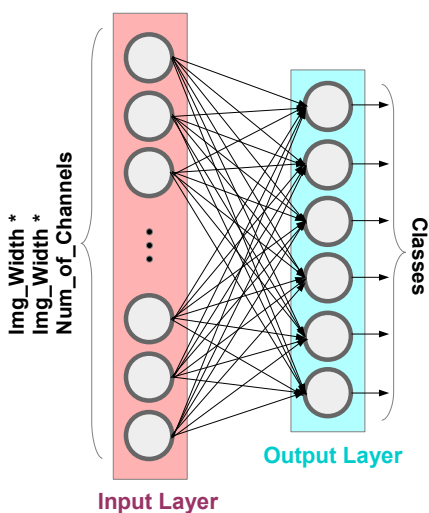
Σχήμα Α'.22: Η Συνελικτική Μηχανή. Απεικονίζεται για έναν πυρήνα διαστάσεων 5×5 .



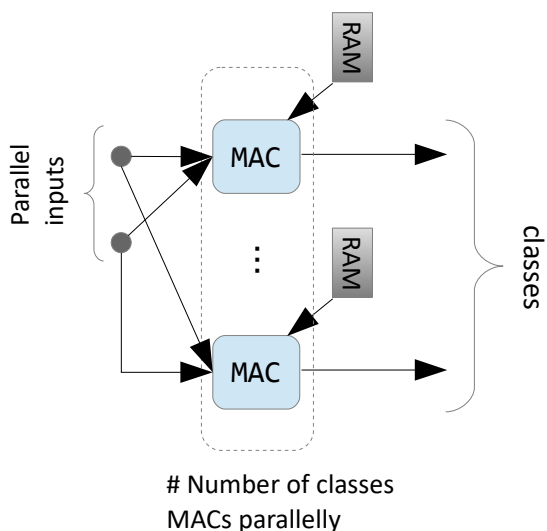
Σχήμα Α'.23: Δημιουργώντας το παράλληλο παράθυρο πάνω στο οποίο θα εφαρμοστεί η χωρική υποδειματοληψία.



Σχήμα Α'.24: Ένα δέντρο συγκριτών για την εξαγωγή του μέγιστου στοιχείου από τα παράλληλα δεδομένα στην είσοδό του. Αξιοποιείται στην χωρική υποδειματοληψία.



Σχήμα Α'.25: Η τοπολογία ενός πλήρως διασυνδεδεμένου επιπέδου. Σε πλήρη αντιστοιχεία με τα παραδοσιακά ΤΝΔ.



Σχήμα Α'.26: Το στοιχείο που εκτελεί το πλήρως διασυνδεδεμένο επίπεδο. Μια MAC μονάδα για κάθε κλάση του ΣΝΔ. Η αρχιτεκτονική που υλοποιεί το πλήρες συνδεδεμένου επίπεδο είναι pipelined ως προς το προηγούμενό του συνελικτικό επίπεδο. Μας επιτρέπει να χρησιμοποιήσουμε ένα μικρό πλήθος από Πολλαπλασιαστές-Συσσωρευτές, καθώς πρόκειται για τον πολλαπλασιασμό ενός πίνακα -τα βάρη του πλήρως διασυνδεδεμένου- με ένα διάνυσμα -η είσοδος του επιπέδου. Το διάνυσμα της εισόδου γίνεται διαθέσιμο σταδιακά.

A'.6 Οργάνωση της on-chip μνήμης

Η οργάνωση της on-chip μνήμης για κάθε επίπεδο εξαρτάται από το πλήθος των παράλληλων εισόδων M_{par} και παράλληλων εξόδων N_{par} του επιπέδου. Έχουμε: M_{par} μνήμες RAM στην είσοδο, N_{par} μνήμες RAM για τη χωρική υποδειγματοληψία, $M_{par} \times N_{par}$ μνήμες ROM για τα φίλτρα και N_{par} μνήμες ROM για την πόλωση.

For $i = 1 \cdots M_{par}$ and $j = 1 \cdots N_{par}$ and M_{total}, N_{total} το συνολικό πλήθος εισόδων και εξόδων αντίστοιχα, έχουμε:

- the Filter ROM_{ij} shall contain the filters in following order:

```

ROMij i= for k=1 to  $\frac{N_{total}}{N_{par}}$ 
    for v=1 to  $\frac{M_{total}}{M_{par}}$ 
        Filter[j + (k - 1) · Npar] : Channel[i + (v - 1) · Mpar]
    end
end

```

- the Bias ROM_j 's contents should have the following order:

```

ROMj i= for k=1 to  $\frac{N_{total}}{N_{par}}$ 
    Filter[j + (k - 1) · Npar]
end

```

- the Pooling RAM_j 's contents should have the following order:

```

ROMj i= for k=1 to  $\frac{N_{total}}{N_{par}}$ 
    Channel[j + (k - 1) · Npar]
end

```

- the input RAM_i 's contents should have the following order:

```

RAMi i= for v=1 to  $\frac{M_{total}}{M_{par}}$ 
    Channel[i + (v - 1) · Mpar]
end

```

Ένα παράδειγμα απεικονίζεται στην εικόνα A'.27.

ROM (1,1)	ROM (1,2)
Filter 1, Channel 1	Filter 2, Channel 1
Filter 3, Channel 1	Filter 4, Channel 1
...	...
Filter 31, Channel 1	Filter 32, Channel 1
ROM (2,1)	ROM (2,2)
Filter 1, Channel 2	Filter 2, Channel 2
Filter 3, Channel 2	Filter 4, Channel 2
...	...
Filter 31, Channel 2	Filter 32, Channel 2
ROM (3,1)	ROM (3,2)
Filter 1, Channel 3	Filter 2, Channel 3
Filter 3, Channel 3	Filter 4, Channel 3
...	...
Filter 31, Channel 3	Filter 32, Channel 3

Σχήμα Α'.27: Οργάνωση μνήμης για 32 φίλτρα των 3 καναλιών έκαστο, για ένα πλέγμα συνελικτικών μηχανών 3×2 ($M_{par} \times N_{par}$).

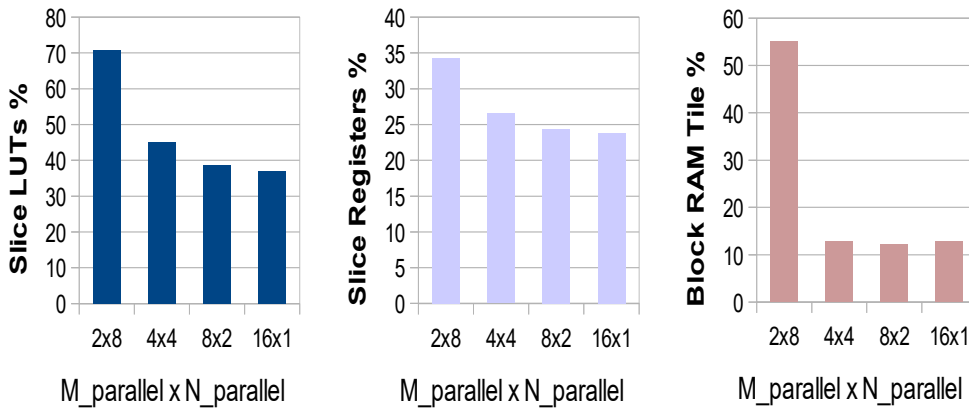
Α'.7 Εξερεύνηση του χώρου σχεδίασης

Έχοντας σχεδιάσει την παραμετροποιήσιμη αρχιτεκτονική υλοποίησης του Modified Cifar-10 Full ΣΝΔ, καλούμαστε να ρυθμίσουμε το μέγεθος της παραλληλίας στις εισόδους και εξόδους όλων των ενοποιημένων επιπέδων. Διαφορετικές ρυθμίσεις αυτών των παραλληλιών μπορούν εν δυνάμει να οδηγήσουν σε ριζικά διαφορετική συμπεριφορά του συστήματος. Μπορούμε να αξιολογήσουμε τη συμπεριφορά του συστήματος με βάση τις κάτωθι μετρικές:

- **Ποσοστό χρήσης των πόρων του συστήματος.** Αυτό το μέγεθος αποτελεί ταυτόχρονα στόχο βελτιστοποίησης και περιορισμό της σχεδίασης. Δεν μπορούμε να υπερβούμε τους διαθέσιμους πόρους μιας συσκευής, αλλά επιθυμούμε να τους αξιοποιήσουμε πλήρως.
- **Χρονική Καθυστέρηση.** Είναι το μέγεθος του χρόνου που χρειάζεται το σύστημα για να επεξεργαστεί μια είσοδο (μία εικόνα).
- **Ρυθμός ταξινόμησης.** Είναι το πλήθος των εικόνων που μπορεί να ταξινομήσει ανά μονάδα χρόνου (Img/sec).
- **Χρόνος αξιοποίησης υλικού.** Είναι ποσοστό που εκφράζει πόσο αποτελεσματικά χρησιμοποιούμε τους πόρους του υλικού που η σχεδίασή μας καταλαμβάνει, δηλαδή εάν και κατά πόσο οι πόροι που έχουμε δεσμεύσει είναι σε συνεχή λειτουργία.
- **Κατανάλωση ενέργειας.** Το ποσό της ενέργειας που το σύστημα που σχεδιάσαμε χρειάζεται για τη λειτουργία του.

Οι ελεύθερες παράμετροι του συστήματος που αφορούν την παραλληλία των εισόδων & εξόδων των επιπέδων είναι οι εξής: M_{par}^{L1} , N_{par}^{L1} , M_{par}^{L2} , N_{par}^{L2} , M_{par}^{L3} , N_{par}^{L3} . Το συνολικό πλήθος

από συνελικτικές μηχανές στις οποίες αντιστοιχούν είναι $Z = M_{par}^{L1} \cdot N_{par}^{L1} + M_{par}^{L2} \cdot N_{par}^{L2} + M_{par}^{L3} \cdot N_{par}^{L3}$. Είναι συνετό να περιορίσουμε τις τιμές που αυτές οι μεταβλητές μπορούν να πάρουν. Στην αντίθετη περίπτωση ο χώρος σχεδίασης αποτελείται από $= \prod_{i=1}^3 (Layer_{in}^i \cdot Layer_{out}^i) = 3 \cdot 32 \cdot 32 \cdot 32 \cdot 64$ ρυθμίσεις, που είναι $> 2^{27}$. Οι περισσότερες από αυτές τις ρυθμίσεις είναι μη εφικτές επιλογές λόγω του τεράστιου αριθμού πόρων της συσκευής FPGA που θα χρειαζόμασταν, ενώ κατά κανόνα οι συσκευές FPGA που εξετάζουμε είναι αρκετά πιο μικρές. Οι τρόποι με τους οποίους αποφασίσαμε πώς και πόσο θα περιορίσουμε αυτές τις μεταβλητές είναι διαισθητικοί και εμπειρικοί. Σε μεγάλο βαθμό εξαρτήθηκαν από αλληπάλληλες συνθέσεις πλήθους σχεδιάσεων προκειμένου να αποκτήσουμε αίσθηση των μεγεθών στα οποία αποτυπώνεται η σχεδιάσή μας. Μια παρατήρηση είναι ότι στις περισσότερες περιπτώσεις επιθυμούμε $N_{par}^L < M_{par}^L$. Αυτό μπορεί να γίνει αντιληπτό και από την απεικόνιση της αρχιτεκτονικής (σχήμα Α'.21) όπου φαίνεται ότι η αύξηση των παράλληλων εξόδων N_{par} συνεπάγεται την αύξηση του πλήθους και άλλων στοιχείων, κι όχι μόνο των συνελικτικών μηχανών. Το αποτυπώνουμε με μετρήσεις από τη σύνθεση και στην παρακάτω εικόνα, όπου φαίνεται ότι μια ρύθμιση ενός επιπέδου $M_{par} \times N_{par} = 2 \times 8$ απαιτεί $\times 2$ πόρους LUT και $\times 4,5$ πόρους BRAM σε σχέση με μια ρύθμιση του ίδιου επιπέδου $M_{par} \times N_{par} = 8 \times 2$.



Σχήμα Α'.28: Χρήση των πόρων του συστήματος για διαφορετικές ρυθμίσεις, όταν το πλήθος των συνελικτικών μηχανών παραμένει το ίδιο. Η αύξηση των παράλληλων εισόδων φαίνεται ως προτιμότερη έναντι της αύξησης των παράλληλων εξόδων.

Υπολογισμός χρονικής καθυστέρησης

Κάθε διδιάστατο κανάλι χρειάζεται $C = E^2 + E \cdot X + b$ κύκλους για να εκτελεστεί η συνέλιξή του με έναν πυρήνα $k \times k$, όπου E είναι το μέγεθος του διδιάστατου καναλιού εξόδου και $X = k - 1$ είναι οι φορές που ο πυρήνας θα βρεθεί στην άκρη του καναλιού εισόδου καθώς 'γλυστράει' επάνω του και θα χρειαστεί να αλλάξει σειρά. Αυτοί οι χρόνοι προκύπτουν από την αρχιτεκτονική της συνελικτικής μηχανής που σχεδιάσαμε. Για να πραγματοποιηθούν όλες οι συνελίξεις εντός των επιπέδων χρειάζονται οι κάτωθι κύκλοι μηχανής:

- **Layer 1** : $C_{total}^{L=1} = 1029 \cdot \text{ceil}(\frac{3}{M_{par}^{L=1}}) \cdot \text{ceil}(\frac{32}{N_{par}^{L=1}})$
- **Layer 2** : $C_{total}^{L=2} = 329 \cdot \text{ceil}(\frac{32}{M_{par}^{L=2}}) \cdot \text{ceil}(\frac{32}{N_{par}^{L=2}})$
- **Layer 3** : $C_{total}^{L=3} = 126 \cdot \text{ceil}(\frac{32}{M_{par}^{L=3}}) \cdot \text{ceil}(\frac{64}{N_{par}^{L=3}})$

Ρυθμός ταξινόμησης

$$\text{Max Latency} = \max(\text{Latency}^{L1}, \text{Latency}^{L2}, \text{Latency}^{L3})$$

$$\text{Throughput} = \frac{\text{Design's Clock Frequency}}{\text{Max Latency}} \left(\frac{\frac{\text{cycles}}{\text{sec}}}{\frac{\text{cycles}}{\text{Img}}} \right) \quad (\text{A.3})$$

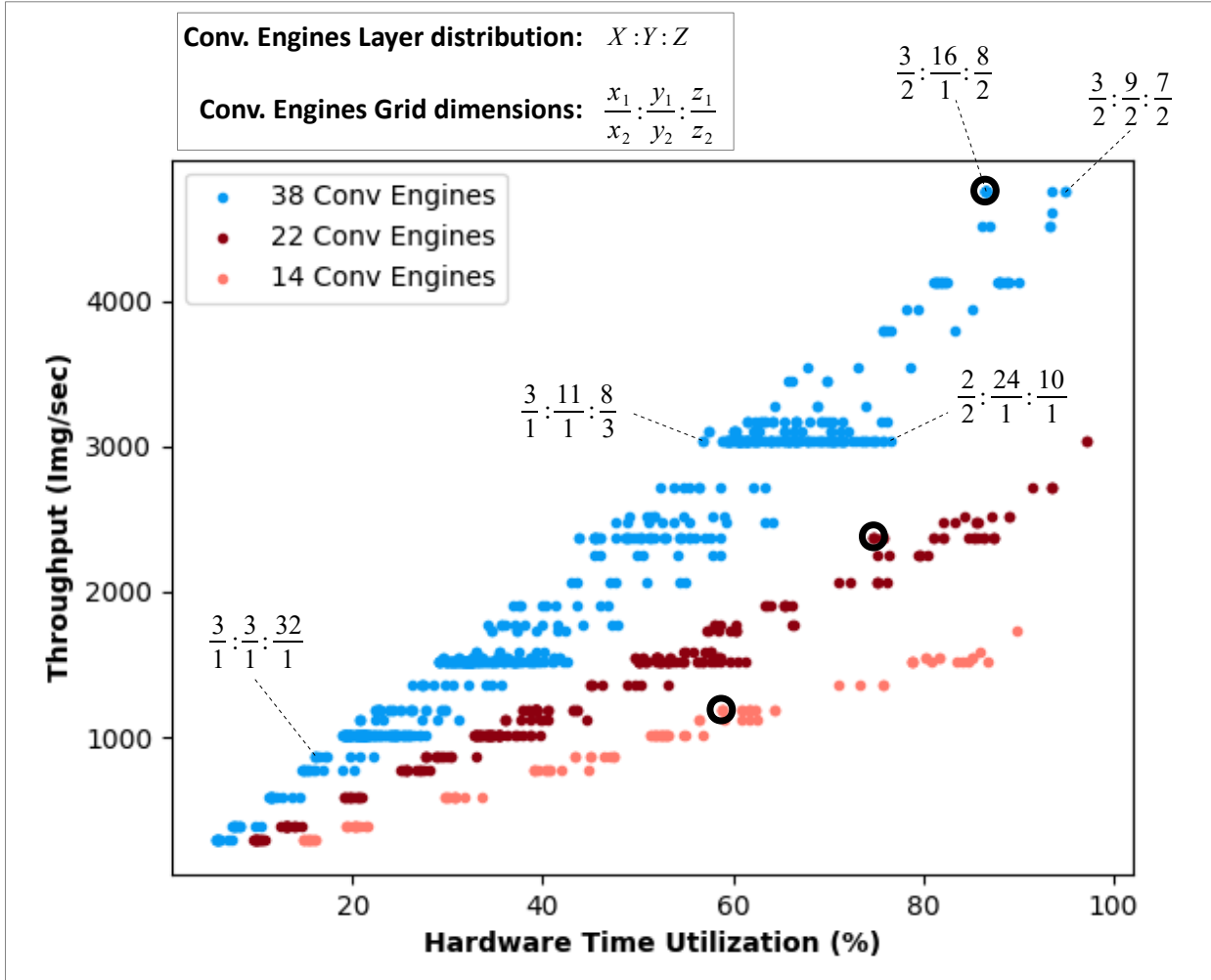
Χρόνος αξιοποίησης υλικού

Συνολικό πλήθος από συνελικτικές μηχανές στο δίκτυο: $Z = M_{par}^{L1} \cdot N_{par}^{L1} + M_{par}^{L2} \cdot N_{par}^{L2} + M_{par}^{L3} \cdot N_{par}^{L3}$.

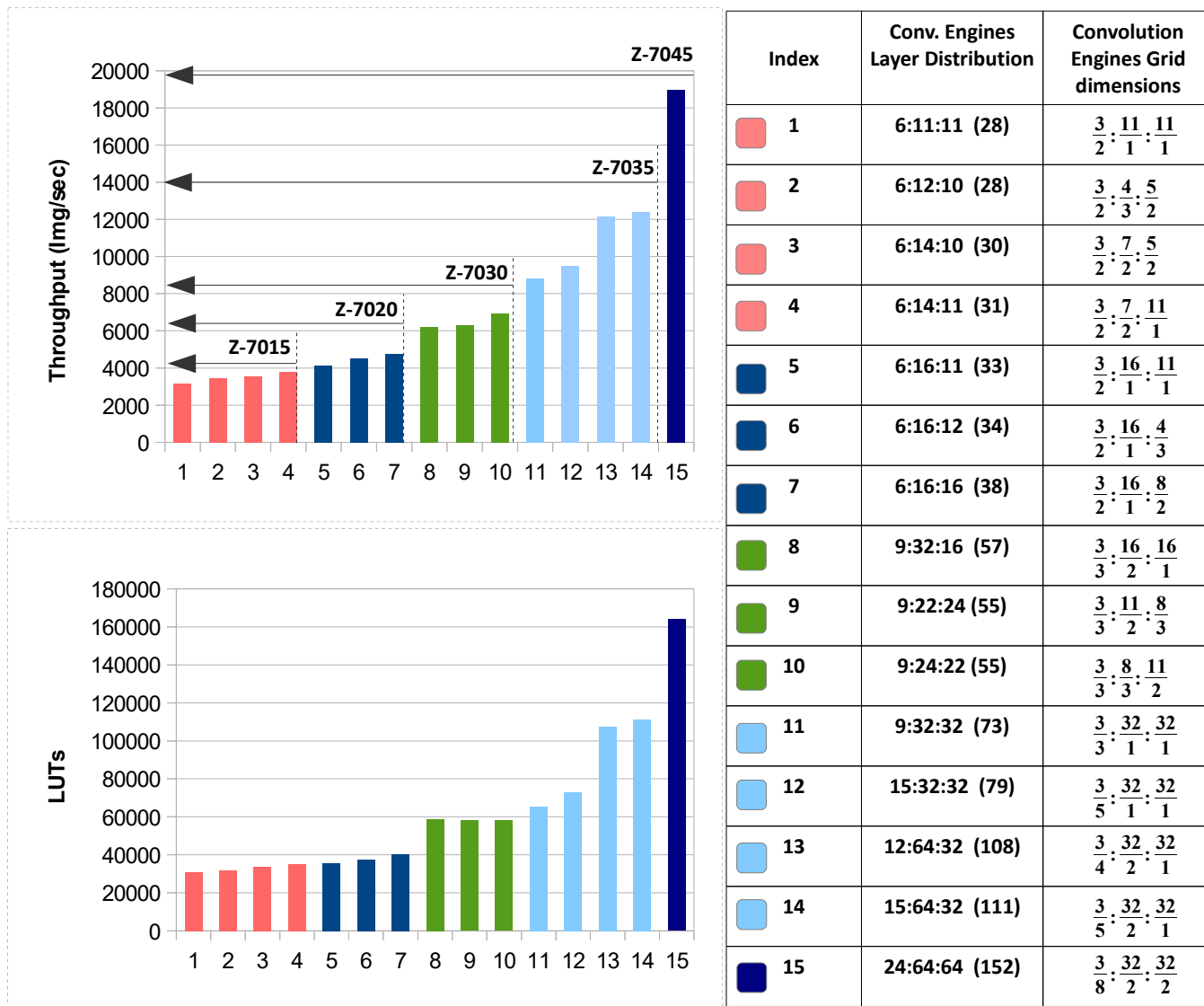
$$\text{Number of Relative Engines} = \sum_{L=1}^3 \frac{(M_{par}^L \cdot N_{par}^L) \cdot (\text{Latency})^L}{(\text{Max Latency})}$$

$$\text{Hardware Time Utilization} = \frac{\text{Number of Relative Engines}}{Z} \cdot 100\% \quad (\text{A.4})$$

A'.7.1 Αποτελέσματα εξερεύνησης χώρου σχεδίασης



Σχήμα Α'.29: Αποτελέσματα εξερεύνησης του χώρου σχεδίασης για τρία διαφορετικά μεγέθη συνολικών συνελικτικών μηχανών. Εδώ μια αναζήτηση όπου ο χώρος περιορίστηκε θέτοντας τις ελεύθερες μεταβλητές $N_{parallel}^L \leq M_{parallel}^L$ για όλα τα επίπεδα, $M_{parallel}^{L=1} \in [1, 3]$ και $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in [1, 32]$. Με μαύρους κύκλους σημειώνουμε τα αποτελέσματα που προκύπτουν αν θέσουμε την παραλληλία των εισόδων και των εξόδων τέτοια ώστε τα M_{par}, N_{par} να είναι διαιρέτες του αρχικού πλήθους εισόδων και εξόδων που προκύπτουν απ την αρχιτεκτονική του ΣΝΔ. Σε εκείνη την περίπτωση οδηγούμαστε σε υποβέλτιστες λύσεις όπως αυτές που έχουμε σημειώσει στο διάγραμμα. Περισσότερη συζήτηση στο αγγλικό μέρος της εργασίας.



Σχήμα Α'.30: Αποτελέσματα των καλύτερων ρυθμίσεων των παραμέτρων όταν στοχεύουμε στις συσκευές της οικογένειας Xilinx Zynq-7000 SoC. Παρουσιάζουμε το ρυθμό ταξινόμησης και την αντιστοιχία της σχεδίασης σε LUT. Η μικρότερη συσκευή στην οποία μπορεί να τοποθετηθεί το Modified Cifar-10 Full ΣΝΔ είναι η Z-7015 λόγω των απαιτήσεων σε μνήμη επί της προγραμματιζόμενης λογικής.

A'.7.2 Συγκριτική εκτίμηση της υλοποίησης

Μέσα από την εξερεύνηση του χώρου σχεδίασης προσδιορίσαμε ότι για την υλοποίηση του Modified Cifar-10 Full ΣΝΔ στο Zynq Z-7020, μια από τις καλύτερες παραμετροποιήσεις της παραλληλίας είναι η $[3 \times 2, 16 \times 1, 8 \times 2]$, η οποία μπορεί να ταξινομήσει εικόνες με τον μέγιστο ρυθμό για αυτή τη σχεδίαση και αυτή τη συσκευή. Οι λεπτομέρειες της υλοποίησης απεικονίζονται στις εικόνες που ακολουθούν. Για να επιτύχει ρυθμό ταξινόμησης 4650 εικόνες το δευτερόλεπτο, η συσκευή πρέπει να τροφοδοτείται με τουλάχιστον τόσες εικόνες το δευτερόλεπτο στην είσοδό της. Αυτό συνεπάγεται ένα εύρος ζώνης $4650 \cdot 28 \cdot 28 \cdot 3 \cdot 8 = 87.5$ Mbit/sec. Στο εργαστήριο Microlab-NTUA έχουμε μετρήσει ότι η προγραμματιζόμενη λογική μπορεί να επικοινωνήσει με τον ενσωματωμένο επεξεργαστή με ένα ρυθμό μετάδοσης ~ 3 Gbit/sec. Επιπρόσθετα, ένα 100Mbit/sec Ethernet καλώδιο μπορεί να καλύψει αυτές τις ανάγκες μετάδοσης για την σύνδεση του Zynq Z-7020 με κάμερα, κτλ. Το επιθυμητό εύρος ζώνης εισόδου είναι λοιπόν εφικτό και ο χρόνος που απαιτείται για να μεταφέρουμε τα δεδομένα δεν αποτελεί περιορισμό. Επί του παρόντος, ο ρυθμός ταξινόμησης της συσκευής περιορίζεται μόνο από την συχνότητα του ρολογιού στην οποία λειτουργεί και τους διαθέσιμους πόρους στο FPGA.

Σημειώνουμε ότι για τους σκοπούς αυτής της διπλωματικής εργασίας δεν κρίθηκε απαραίτητο να υλοποιήσουμε τη σχεδίασή μας σε ένα λειτουργικό πρωτότυπο. Αντ' αυτού στοχεύσαμε στην ακριβή εκτίμηση της επίδοσης και του κόστους της υλοποίησης ενός ΣΝΔ σε FPGA συσκευές.

FPGA chip on Xilinx Zynq Z-7020 SoC	
7 Series PL Equivalent	Artix-7
Logic Cells	85K
Look-Up Tables (LUTs)	53200
Flip-flops	106400
Total Block RAM	4.9 Mb
DSP Slices	220

Σχήμα A'.31: Οι προδιαγραφές της συσκευής Zynq Z-7020 SoC.

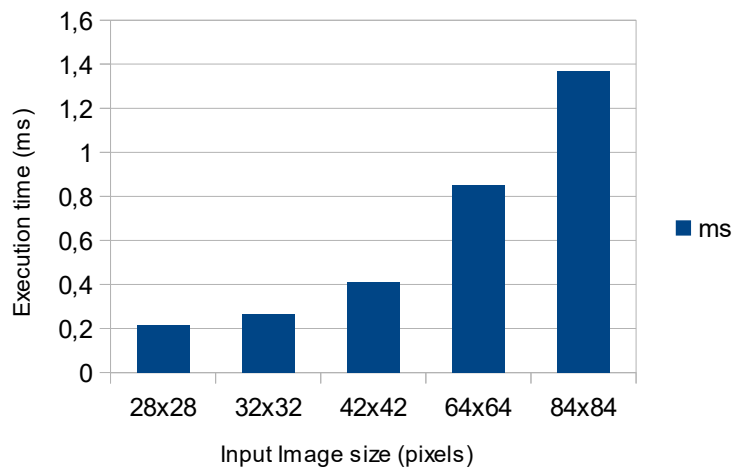
	L1	L2	L3	FL
Engines	3x2	16x1	8x2	6 (2 inputs each)
Slice LUTs %	13.33	35.97	26.49	0.08
Slice Registers %	8.79	23.75	19.8	0.02
Block RAM Tile %	15.71	18.56	20.71	2.14
DSPs %	0	0	90.91	8.18
Latency (cycles)	16574	21502	17022	288 (as stand alone) 4 (attached to L3)
Possible Clock (ns)	6	9	8	-
Dynamic Power (W)	0.412	0.46	0.752	0.026
Device Static Power (W)	0.106	0.107	0.11	0.101

Σχήμα Α'.32: Αποτελέσματα της σύνθεσης της προτεινόμενης αρχιτεκτονικής για το Modified Cifar-10 Full ΣΝΔ στην πλατφόρμα Zynq Z-7020 SoC ανά επίπεδο.

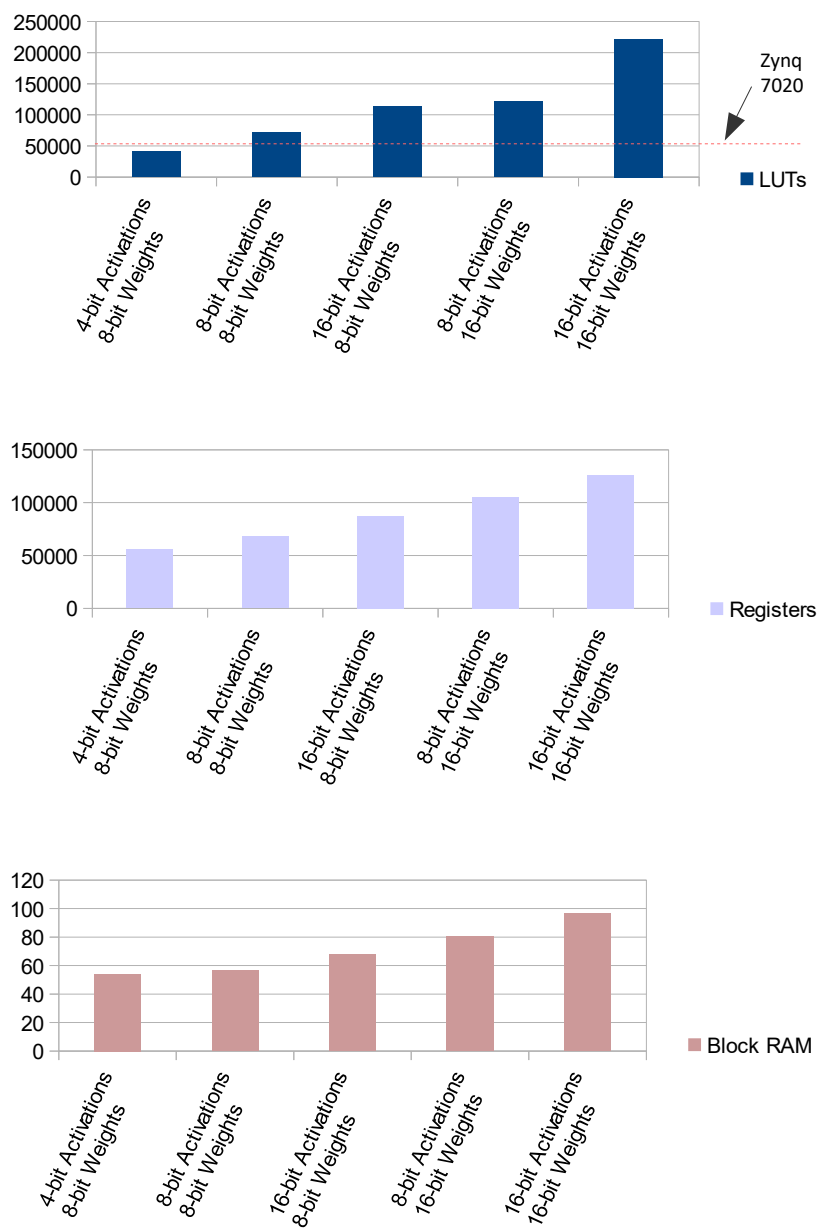
Deliverable Design			
Algorithm : "Modified Cifar-10 Full" Convolutional Neural Network			
Target Platform	Xilinx Zynq Z-7020	Convolution Engines	38
Slice LUTs %	75.81	Latency (ms)	0.5510
Slice Registers %	52.36	Throughput (Images/sec)	4650
Block RAM Tile %	57.12	Hardware Time Utilization %	87.6
DSPs %	99.09	Image size (RGB)	28x28x3, 8-bit values
Clock (ns)	10 (100 MHz)	Word Length	Inputs & Outputs : 4 bit Conv Weights : 8 bit FC Weights: 2 bit
Total On-chip Power (W)	1.76	CNN's Classification Accuracy	94.89%
Designed with	VHDL	Workload per Image (Million OPs)	18.864

Σχήμα Α'.33: Αποτελέσματα της σύνθεσης της προτεινόμενης αρχιτεκτονικής για το Modified Cifar-10 Full ΣΝΔ στην πλατφόρμα Zynq Z-7020 SoC για το συνολικό δίκτυο.

Καθώς η αρχιτεκτονική που σχεδιάσαμε βασίζεται στην on-chip μνήμη, δεν μπορεί να υποστηρίξει εικόνες οποιουδήποτε μεγέθους. Στο Zynq Z-7020 SoC μπορεί να επεξεργαστεί εικόνες μέχρι περίπου 3 φορές το μέγεθος των εικόνων του SAT-6 airborne dataset. Ο χρόνος εκτέλεσης αυξάνεται τετραγωνικά με το μέγεθος της εικόνας (εικόνα Α'.34).



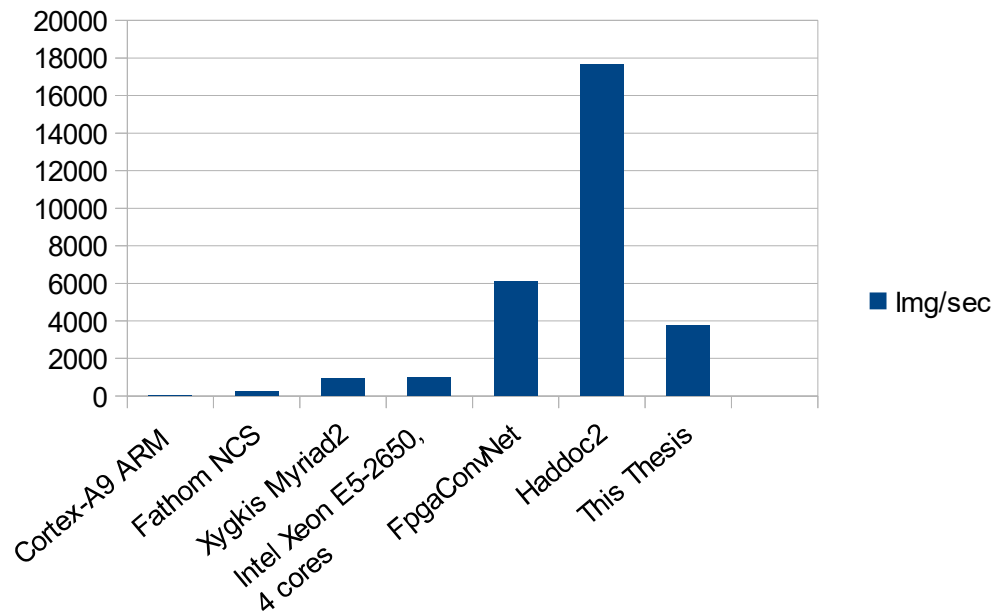
Σχήμα Α'.34: Χρόνος εκτέλεσης του 'Modified Cifar-10 Full' ΣΝΔ στην προτεινόμενη αρχιτεκτονική για το Zynq Z-7020 SoC σε συνάρτηση με το μέγεθος της εικόνας. Το Zynq Z-7020 μπορεί να υποστηρίξει εικόνες μέχρι $\sim 50 \times 50$ εικονοστοιχεία.



Σχήμα Α'.35: Η μεταβολή του πλήθους των πόρων FPGA που χρησιμοποιεί η προτεινόμενη αρχιτεκτονική για το Zynq Z-7020 [3×2, 16×1, 8×2], όταν μεταβάλλεται το μήκος της λέξης. Η βελτιστοποίηση στο μήκος της λέξης (Α'4.3) ανέδειξε τα 8-bit βάρη και 4-bit εισόδους και εξόδους ως κατάλληλη επιλογή για τον συγκεκριμένο ΣΝΔ στα συγκεκριμένα δεδομένα εικόνων.

Implementation Device	CNN Model	Input Image Size	Programming	Bitwidth	Freq	Power	Throughput (img/sec)
VPU: Fathom NCS [Xygi17]	"Cifar-10 Quick"	32x3x3	Caffe software	16 fixed	N/a	N/a	232
VPU: Myriad2 [Xygi17]	"Cifar-10 Quick"	32x3x3	C & Assembly	16 fixed	600 MHz	1,09 W	909
CPU: Intel Xeon E5-2650 v2, 4 cores [Xygi17]	"Cifar-10 Quick"	32x3x3	Caffe software	32 float	2.6 GHz	N/a	1000
CPU: Cortex-A9 ARM [Dan18]	"Cifar-10"	32x3x3	Caffe software	32 float	N/a	N/a	10
FPGA: FpgaConvNet [Ven17]	"Cifar-10"	32x3x3	Automatically generated HLS	16 fixed	125 MHz	N/a	6080
FPGA: Haddoc2 [Abd17]	"Cifar-10"	32x3x3	Automatically generated VHDL	6 fixed	54.17 MHz	N/a	17633
This Thesis	"Modified Cifar-10 Full"	32x3x3	VHDL	<i>Dynamic Fixed Point</i> Inputs & Outputs : 4 Conv Weights : 8 FC Weights : 2	100 MHz	1.76 W	3778

Σχήμα Α.36: Σύγκριση με άλλες υλοποιήσεις από τη βιβλιογραφία, εκτελώντας το "Cifar-10" ΣΝΔ.



Σχήμα Α.37: Σύγκριση με άλλες υλοποιήσεις από τη βιβλιογραφία, εκτελώντας το "Cifar-10" ΣΝΔ. Τα FpgaConvNet και Haddoc2 είναι FPGA υλοποιήσεις επί της Xilinx Zynq Z-7045 συσκευής, η οποία είναι ~4 φορές μεγαλύτερη από την Zynq Z-7020

Σχήμα Α'.38: Σύγκριση της αρχιτεκτονικής FPGA της παρούσας διπλωματικής με τις υλοποιήσεις των αυτοματοποιημένων εργαλείων FpgaConvNet και Haddoc2 επί της ίδιας FPGA συσκευής.

Device : Xilinx Z -7045 CNN : Cifar10 Input Image Size : 32x32x3						
Implementation	Bitwidth	Freq	LUTs %	DSPs %	Power	Throughput (Img/sec)
FpgaConvNet (v)	16 fixed	125 MHz	N/a	N/a	N/a	6080
Haddoc2 (vi)	6 fixed	54.17 MHz	79 %	0 %	N/a	17633
This Thesis Layer Configuration: 3x8, 32x2, 32x2	Dynamic Fixed Point Inputs & Outputs : 4 Conv Weights : 8 FC Weights : 2	100 MHz	75 %	88.89 %	4.79 W	15115

Implementation	Throughput (Images/sec)
FpgaConvNet	6080
Haddoc2	17633
This Thesis	15115

CHAPTER 2

Introduction

2.1 Motivation

Computer Vision is of particular interest in creating autonomous machines and further automate parts of the human activities. We seek to equip machines with the capability of dealing with sensory inputs and employ them for tasks that the humans are carrying out based on their visual ability. Over the past decade Convolutional Neural Networks (CNNs) emerged as the state-of-the-art approach to tackle problems of Computer Vision, such as image classification and object detection. Convolutional Neural Networks are a particular class of Artificial Neural Networks (ANNs), which are computing systems and/or algorithms vaguely inspired by the way brain works and constitute an important part of the Machine Learning field. ANNs learn to perform the desired function by iteratively examining a large amount of data and tuning their internal components accordingly. Even though CNNs were first introduced at the beginning of the 1990s, work on ANNs had already emerged by the mid-twentieth century with software and hardware implementations of ANNs in a multitude of applications, like character recognition in OCR, time series prediction in real estate and process control in manufacturing [Lei] [Wid94].

The high performance in accuracy of the contemporary, deep Artificial Neural Networks in a variety of tasks -and particularly, of CNNs in image classification and object detection, alongside the exponential growth of digital data and the emergence of the Internet Of Things (IoT), has renewed the interest in applying these computational systems in research as well as in commercial and industrial applications. In many of these applications, such as assisting the navigation of autonomous vehicles, the computing systems must be physically small, have low power consumption and be able to operate within demanding timing constraints (real-time). Contemporary Convolutional Neural Networks have very high computational complexity and in order to be effectively applied, they need to be integrated in low-power, high-performance embedded computing systems.

Field Programmable Gate Array (FPGA) is a reconfigurable integrated circuit (IC) technology that has been identified as an exceptional candidate for the implementation of the inference stage of Deep Neural Networks.

The FPGA architecture can be customized for specialized parallel computing, naturally matching Artificial Neural Networks, which are inherently massively parallel computing systems and algorithms. FPGAs provide huge processing capabilities with great power efficiency, reducing thermal management and space requirements. This feature allows the integration of acceleration hardware in small housings, on-board equipment, or extreme temperature environments. Moreover, FPGAs have deterministic timing in the order of nanoseconds, which can be an especially important feature in real-time applications. Compared against the high-end GPUs, FPGAs consume up to two orders of magnitude less power. For example, the NVIDIA Tesla K40 GPU consumes 235 Watt, while the Xilinx Zynq Z-7020 System-On-Chip consumes 2.5W. The low-power, high-performance and dynamic reconfigurability of FPGAs has led companies like Microsoft and Amazon to utilize them in order to accelerate datacenter operations with reduced power consumption, or provide Artificial Intelligence (AI) cloud computing. Although cloud computing solutions for Convolutional Neural Networks are available, in applications where i) system reaction time is critical, ii) fast connection to the cloud is not guaranteed, iii) privacy is highly important and, iv) energy and/or monetary costs are limited, computation needs to be done locally. To benefit from these advantages, many hardware accelerators of Deep Neural Networks have recently been proposed.

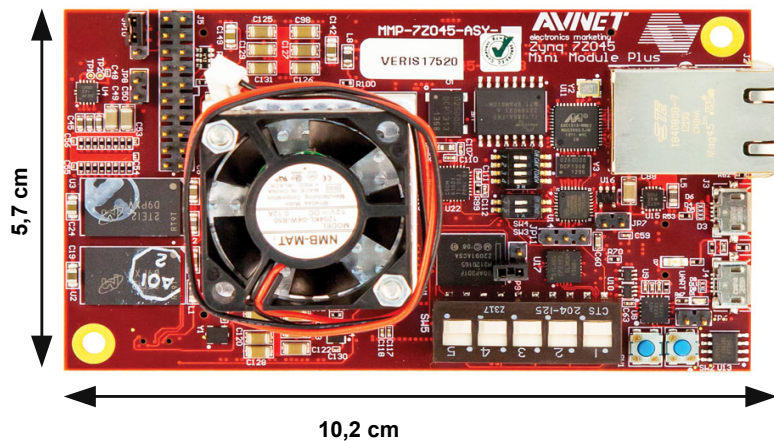


Figure 2.1: Example of a System-on-Module that can be used for embedded computing solutions: the Xilinx Zynq MMP, which contains a Zynq-Z7045 SoC. This board costs about 3000€. Lower cost solutions do exist, like the Zybo Zynq-7000, which features a smaller SoC and costs approximately 300€.

Highly accurate image classification and object detection is of great importance in several applications that depend on airborne/satellite data. Traditionally in these applications the data are not evaluated locally at the time of their acquisition, but rather they are stored or transmitted to be processed remotely. Placing embedded systems capable of executing the inference of Convolutional Neural Networks on UAVs-drones and satellites can offer automated, highly accurate, real-time and locally computed image classification & object detection. This can be of great benefit in numerous cases, such as satellite-based early warning systems, creating autonomous UAVs-drones for emerging technical solutions in disaster response, automated precision agriculture, drone delivery systems, etc. In such applications both the power consumption and physical size of the embedded devices are crucial and therefore choosing the appropriate embedded platform becomes important. In several applications we are interested in land cover classification. Land cover represents the physical and biological cover of the Earth's surface including classes such as forests, water bodies, build-up areas, and agricultural areas. Land cover classification maps allow to track issues like deforestation/reforestation, water sources reduction and urban growth.

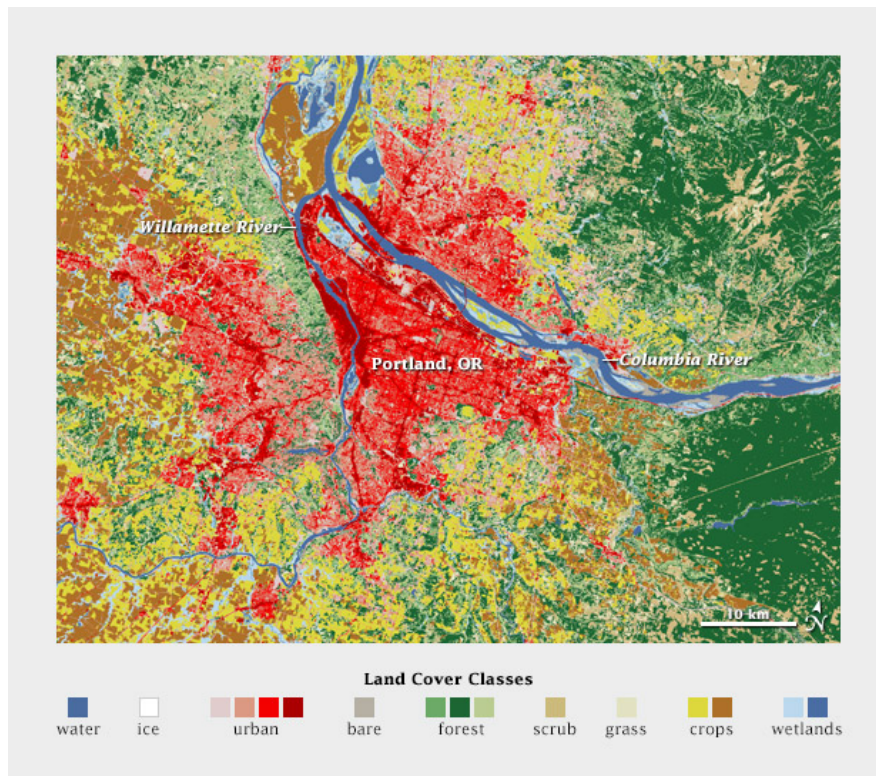


Figure 2.2: A land cover map from the area of Portland, Oregon, in the USA (NASA Earth Observatory).

2.2 Thesis Goals

This thesis has the following goals:

- Review the recent developments in the field of Machine Learning regarding Convolutional Neural Networks.
- Review the prior work in the approaches of designing hardware CNN implementations.
- Select and train a suitable CNN architecture for the problem of satellite image classification.
- Optimize the chosen Convolutional Neural Network architecture for implementation on FPGA.
- Design FPGA architectures to implement Convolutional Neural Networks.
- Optimize and evaluate the FPGA architecture of the CNN designed, regarding the throughput performance -images classified per second- and the device utilization for several devices of the Xilinx Zynq-7000 SoC family.

CHAPTER 3

Background and concepts

This chapter's goal is to present the two main technological advancements that this Thesis is based on: Convolutional Neural Networks (CNNs) and Field-Programmable Gate Arrays (FPGAs).

3.1 Convolutional Neural Networks

3.1.1 A step back: The bigger picture of Machine Learning

Before we dive into the specifics of Convolutional Neural Networks, we first have to put them in context: CNNs belong to a wide range of algorithms in the field of Machine Learning (ML). "What is Machine Learning", one may ask, "and why should one care"? If we search the net about the importance of ML, we'll find plenty of success stories of how ML is already integrated in several aspects of everyday-life in the developed world: e-mail spam filters, voice, text and image recognition, reliable web search engines, Grandmaster's level chess opponents, personal recommendations of music, increasingly autonomous vehicles, etc. However important, this simple listing of ML applications, does not really answer the questions posed.

Machine Learning is a computational sub-field of Artificial Intelligence. Artificial Intelligence itself, poses two main questions: "What is intelligence and how does it work?" and "Can we build intelligent machines?". Correlated with the latter one and, as its name suggests, in Machine Learning we try to *train* computers, in a way that they can *learn* to solve problems, without being explicitly programmed. Using a more formal definition for "learning" in this context: "A computer program is said to learn from experience E, with respect to some task T, and some performance measure P, if its performance on T as measured by P, improves with experience E". At the core of Machine Learning lies the assumption that *knowledge can be derived from data*. Based on this assumption, the majority of ML algorithms so far are data-driven, in contrast to other AI approaches which may be symbolic, knowledge-based, etc. Machine Learning

takes steps towards "intelligent" machines, promising a wider range and greater depth of automation in human activities. Both the theoretical work and its technological applications contribute to the material preconditions for a world where monotonous, repetitive tasks will be carried out by machines.

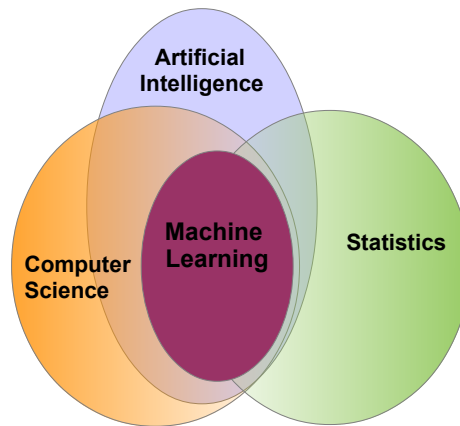


Figure 3.1: Machine Learning's relation to other fields

Types of Machine Learning

So far, there are two major types of Machine Learning: Supervised Learning and Unsupervised Learning. CNNs usually employ supervised learning techniques.

- In *Supervised Learning* we have the input variables (X) and the output variables (Y) and the goal is for the algorithm to learn an approximation of the mapping function from the input to the output, $Y = f(X)$. It is called "supervised" because there is a form of "teacher", giving feedback to the algorithm, based on the already known correct answers. The most common problems that fall under supervised learning are classification and regression, depending on whether the output is discrete (i.e classes) or continuous, respectively.
- In *Unsupervised Learning* we are only given the input data (X) and no corresponding output variables. The goal is to model the underlying structure in the data, if of course there is one. The most common problems in unsupervised learning are those of clustering: We are interested in grouping a set of objects in such a way that objects in the same group, the "cluster", are more *similar* to each other than to those in other clusters.

In this thesis, the problem at hand is a classification problem. With that in mind, we'll make a small intro to supervised learning.

A bit of Supervised Learning

Machine learning algorithms are data-driven. In general, obtaining data is no easy task, neither is deciding what kind of data we need for a specific problem.

From a systems perspective, we feed an algorithm's inputs with data, paired with their corresponding, known output and the algorithm iteratively calibrates the parameters of a model to fit these data. The dataset used to train our algorithm is called the "training set". After the training, our model's performance will be assessed using previously *unseen* data, usually called the "test set". The model's ability to perform good on data with which it has not been calibrated, has to do with the concept of *model generalization* and of course, with the need for a representative training dataset. The ability of the models to generalize is a central goal in machine learning. We are not only looking for our model to decently fit the training data through minimizing a cost function, we also want the generalization error, also called the test error, to be low as well [Goo16, p. 108]. The factors determining how well a machine learning algorithm will perform are its ability to:

- Make the training error small.
- Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: under-fitting and over-fitting. Under-fitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Over-fitting occurs when the gap between the training error and test error is too large (figure 3.3) [Goo16, p. 109]. A training set for which the two aforementioned conditions are sufficiently met is a *representative* training set.

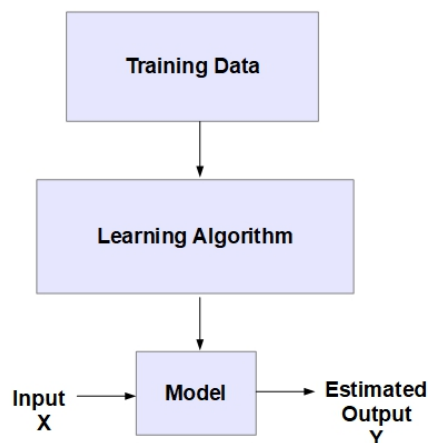


Figure 3.2: Supervised Learning

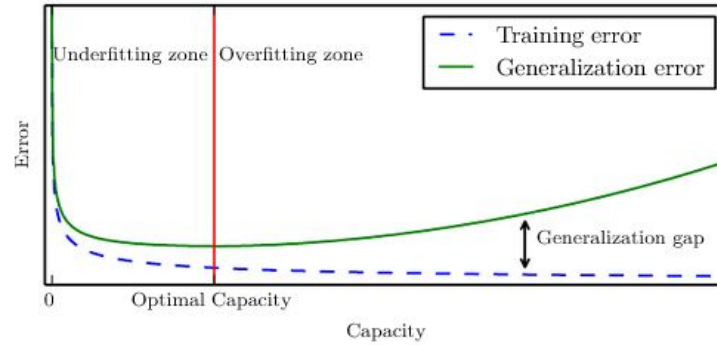


Figure 3.3: Relation of Training Error and Generalization Error to Under-fitting & Over-fitting. Source: "Deep Learning", MIT Press, p.113

As we mentioned, the algorithm calibrates the parameters of the model. The form of the model, also called the "hypothesis", may be pre-decided by the engineer. For example, in the case of linear regression, we have a vector of independent variables $X \in R^n$, also called the predictors or features, and the output is a continuous function of the input, $Y \in R$. We want to specify & construct a function between the input and the output of the model, based on known pairs of inputs and outputs. Based on the hypothesis that there is a linear relationship between X and Y , we define our model as:

$$H = W^T X + B, \quad (3.1)$$

where $H \in R$ and W^T is a $1 \times n$ vector. W and B are the parameters of this model and they control its behaviour. The algorithm will iteratively change the values of W, B in order for our model H to approximate the given values of Y . Having a function for the model, we also need a function that will quantify how well our model approximates the desired output Y . This function is called the "objective" or "cost function". A frequently used cost function is for example the mean squared error of the predictions $H(x)$ and the real values Y for every point i in in our training set.

$$J(a_0, a_1) = \frac{1}{2m} \sum_{i=1}^m (H(x_i) - Y_i)^2 \quad (3.2)$$

In order for our model H to perform better, we need to minimize the equation 3.2. One way to achieve this is through an algorithm called *Gradient Descent*. For example, if we suppose a hypothesis $h(x) = a_1 x + a_0$, then the Gradient Descent algorithm would be:

$$a_i = a_i - L \frac{\partial}{\partial a_i} J(a_0, a_1) \quad (3.3)$$

where $i = 0$ and $i = 1$ are simultaneously updated until convergence and L is defined as the learning rate.

Several variations of the Gradient Descent algorithm are also used when training Convolutional Neural Networks. We should note that the learning rate is of great importance: If L is too small, gradient descent can be very slow. If L is too large, gradient descent may overshoot the minimum, which means it may fail to converge, or it could even diverge. After we have calibrated our model to all the data in the training set, we then use the same cost function to calculate the test error.

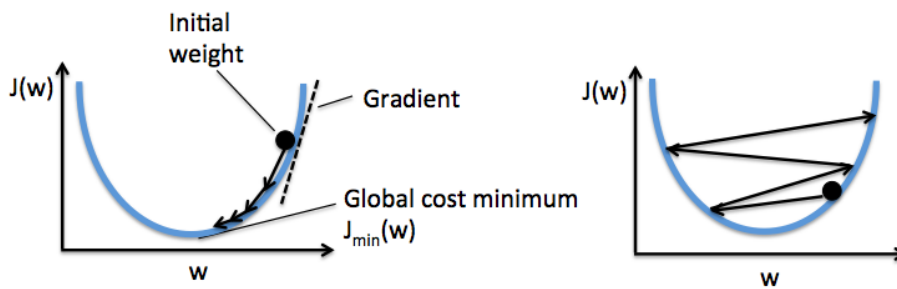


Figure 3.4: Learning Rate in Gradient Descent.

In the example presented above, the hypothesis space of the model described by equation 3.1, meaning the set of functions that the learning algorithm is allowed to select as being the solution, is the set of all linear functions of its input. If we define the *capacity* of a model as its ability to fit a wide variety of functions, then we can observe that: i) Models with low capacity may struggle to fit the training set and ii) Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set (figure 3.5). Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task, they may overfit [Goo16, p. 110].

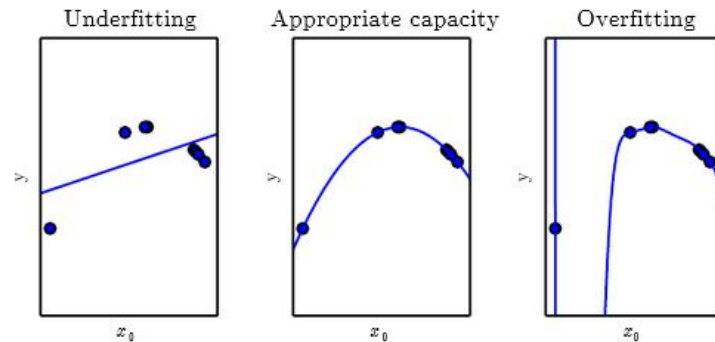


Figure 3.5: Under-fitting, Fitting and Over-fitting. Models' Capacity. Source:[Goo16]

3.1.2 The problem of Image Classification

The problem of image classification is the task of assigning an input image one label from a fixed set of categories. It is one of the core problems in Computer Vision and has a large variety of practical applications. To a computer an image is represented as a multi-dimensional array of numbers, usually as a 3-D array of brightness values when using the RGB color model. Thus a 32x32 pixels image may be represented as a 32x32x3 array, a total of 3072 numbers.

The data-driven approach to this algorithmic problem relies on first accumulating a training dataset of labeled images, which will serve as examples of each class, upon which the learning algorithms will be trained. Although recognizing a visual concept is trivial for humans, there are various challenges involved from the perspective of a Computer Vision/ Machine Learning algorithm. Some of these challenges are [Kara]:

- **Viewpoint variation** A single instance of an object can be oriented in many ways with respect to the camera.
- **Scale variation** Visual classes often exhibit variation in their size (size in the real world, not only in terms of their extent in the image).
- **Deformation** Many objects of interest are not rigid bodies and can be deformed in extreme ways.
- **Occlusion** The objects of interest can be occluded. Sometimes only a small portion of an object (as little as few pixels) could be visible.
- **Illumination conditions** The effects of illumination are drastic on the pixel level.
- **Background clutter** The objects of interest may blend into their environment, making them hard to identify.

- **Intra-class variation** The classes of interest can often be relatively broad, such as "chair". There are many different types of these objects, each with their own appearance.

A good image classification model must be invariant to the cross product of all these variations, while simultaneously retaining sensitivity to the inter-class variations.

Over time, different algorithms have been used for image processing tasks. Traditionally these algorithms were making use of handcrafted features, like detecting the edges at various positions inside the image. Neural networks, in contrast, can automatically create both high-level and low-level features. This is one of the key properties that lies in the core of Deep Learning and distinguishes it from the various other machine learning fields. Deep Learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction, while the models extract the features needed in the procedure by themselves. In order to achieve this, an abundance of data needs to be available, to serve as training examples for the algorithms. Advances in computer systems and the emergence of big data have enabled the training of increasingly complex and deep models, which have achieved superior performance in a variety of tasks in the past few years.

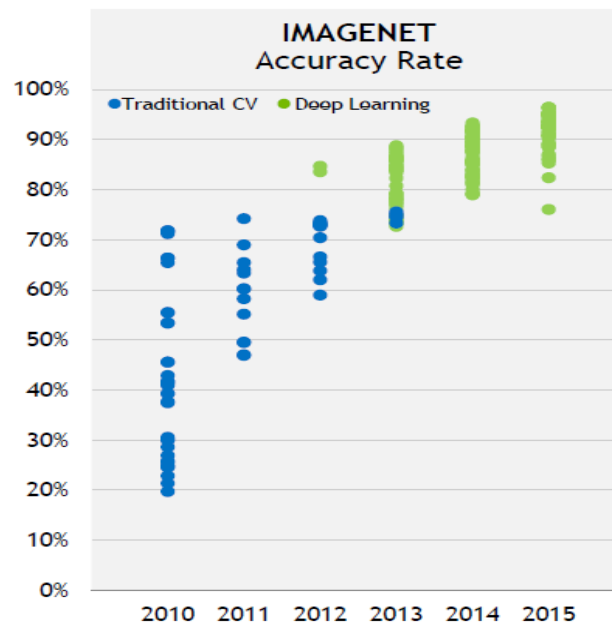


Figure 3.6: Performance of traditional Computer Vision and Deep learning algorithms in the Imagenet competition, 2010-2015. Top-5 accuracy illustrated. [Source: openpowerfoundation.org](http://openpowerfoundation.org)

3.1.3 Introduction to Artificial Neural Networks

In this section we'll briefly present Artificial Neural Networks (ANNs), a very interesting part of Machine Learning, with rich history and theoretical foundation. Artificial Neural Networks have regained attention in the past two decades, being in the center of the *Deep Learning* approach. They have been employed in multiple tasks, such as: Pattern Association, Pattern Recognition, Function Approximation and Processes Control.

Work on ANNs emerged in the mid-twentieth century and has been motivated by the recognition that the human brain computes in an entirely different way compared to the conventional digital computer. The brain is a highly complex, nonlinear, and parallel information-processing system, with the capability to organize its structural constituents, known as neurons [Hay09]. While computers perform extremely well in a variety of tasks, outperforming humans in speed and accuracy when coming to the manipulation of numerical data, that has not been the case for problems like pattern recognition, perception and motor control. Artificial Neural Networks are computing systems inspired by the biological neural networks.

A biological and a mathematical model of a neuron can be seen in figure 3.7. Neurons are the basic computational units. In the biological model each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model, the artificial neuron, also called the perceptron, receives a number of input signals x_i from other neurons. These input signals are multiplied with weights w_i to simulate the synaptic interaction at the dendrites. The weighed input signals are summed up, biased with a fixed w_b and fed into a non-linear activation function φ , which produces the neuron's output signal $y = \varphi(\sum [x_i \cdot w_i] + w_b)$ [Karb].

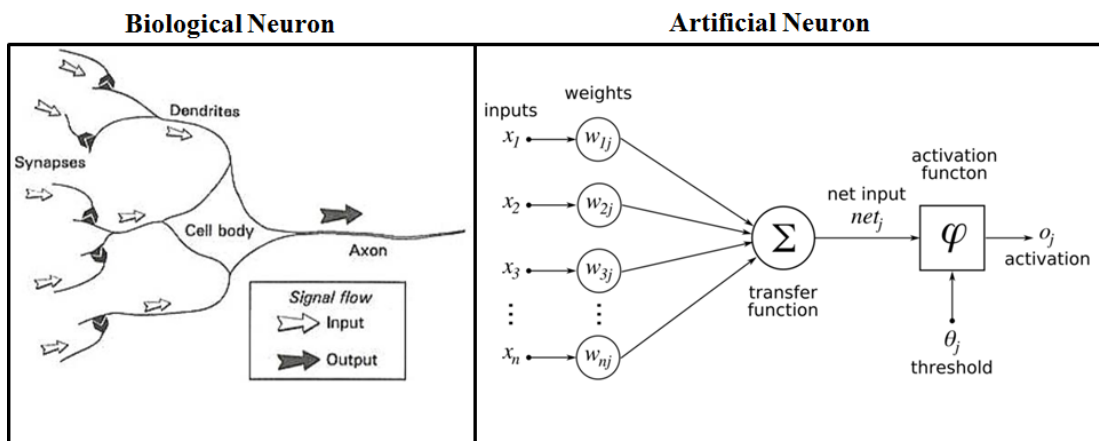


Figure 3.7: A simplified model of a biological neuron (left) and its mathematical model (right).

The weights w can be seen as the tuning knobs that define the neuron's reaction to a given input signal, and their values can be adjusted in order to learn to approximate a desired output signal. Historically, a common choice of the activation function φ is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. [Karb]

A single neuron is the basic building block in Artificial Neural Networks. By interconnecting many of these neurons, a network is created, which exhibits behaviour far more complex than that of a single neuron.¹ The network's weights are adjusted based on a learning algorithm, the most common of which is *Backpropagation*. The behaviour of the network is highly dependent on its structure and new forms of neural networks are continually being created. Several techniques also exist with which the ANNs can dynamically adapt both their weights and their own structure. An illustration of several types of ANNs follows in the next page².

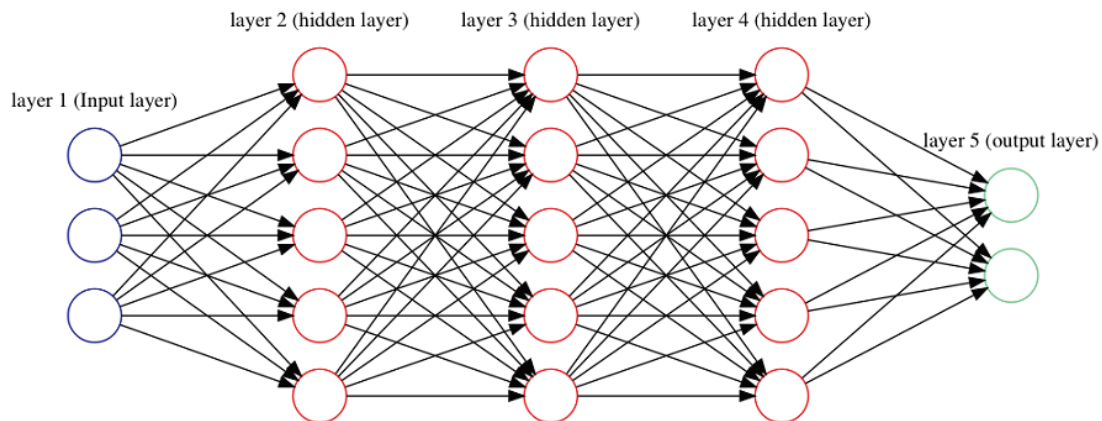


Figure 3.8: Example of a Feed-Forward Neural Network with three hidden layers, three inputs and two outputs. The weights of the network are illustrated with the arrows.

In the example of figure 3.8, the input is a $[1 \times 3]$ vector and the output a $[1 \times 2]$ vector. All the weights of a layer can be stored in a single matrix. For example, the weights that connect the input to the second layer in figure 3.8 are a matrix of size $[3 \times 5]$. Then, in the forward pass phase, the values at the second layer can be calculated through multiplication of the input vector by the weight matrix, resulting to a $[1 \times 5]$ vector. By applying the activation function φ to each one of the vector's elements, the values at the second layer are calculated. In a similar way the output of each of the next layers is calculated. The full forward pass of this 4-layer neural network is then simply four matrix multiplications, interwoven with the application of the activation function.

1 A classical, simple and insightful example is how a single layer of perceptrons can not implement the XOR function, while two layers of perceptrons can.
2 Source of illustration: Asimov Institute

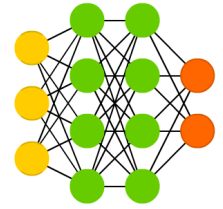
A mostly complete chart of

Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

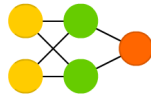
Deep Feed Forward (DFF)



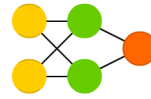
Perceptron (P)



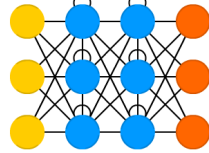
Feed Forward (FF)



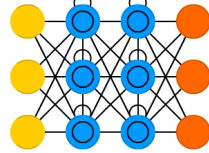
Radial Basis Network (RBF)



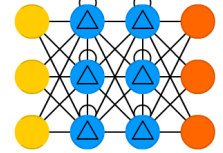
Recurrent Neural Network (RNN)



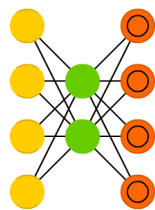
Long / Short Term Memory (LSTM)



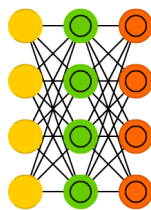
Gated Recurrent Unit (GRU)



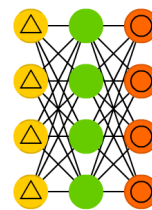
Auto Encoder (AE)



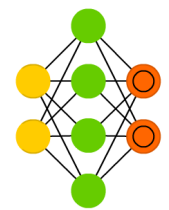
Variational AE (VAE)



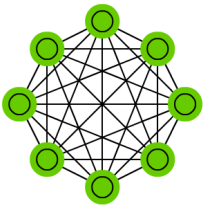
Denoising AE (DAE)



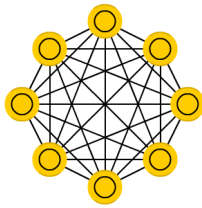
Sparse AE (SAE)



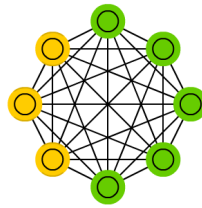
Markov Chain (MC)



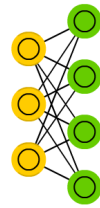
Hopfield Network (HN)



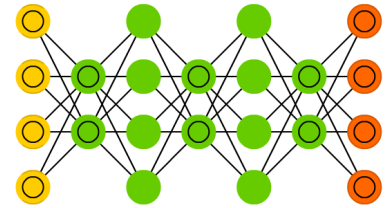
Boltzmann Machine (BM)



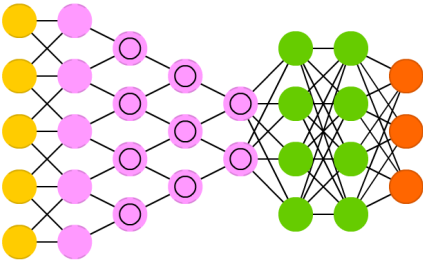
Restricted BM (RBM)



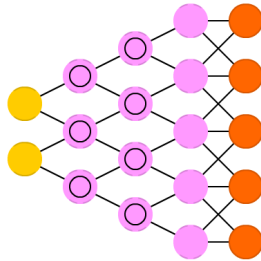
Deep Belief Network (DBN)



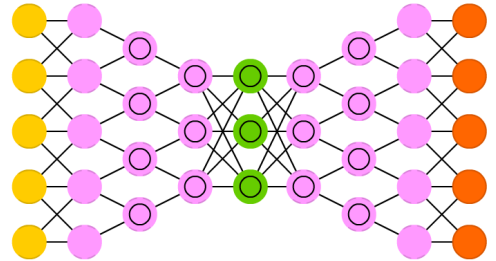
Deep Convolutional Network (DCN)



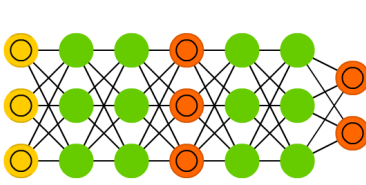
Deconvolutional Network (DN)



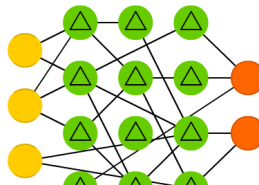
Deep Convolutional Inverse Graphics Network (DCIGN)



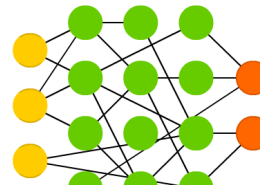
Generative Adversarial Network (GAN)



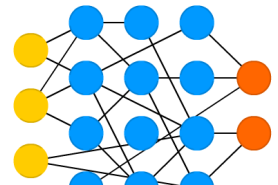
Liquid State Machine (LSM)



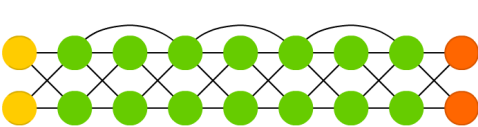
Extreme Learning Machine (ELM)



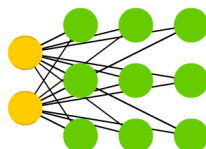
Echo State Network (ESN)



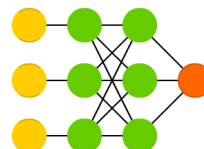
Deep Residual Network (DRN)



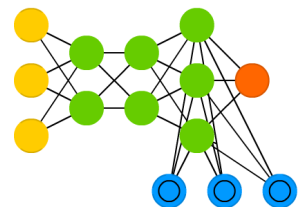
Kohonen Network (KN)



Support Vector Machine (SVM)



Neural Turing Machine (NTM)



3.1.4 Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special type of acyclic Artificial Neural Networks, which aim to the processing of data that has a known grid-like topology. Convolutional Neural Networks are in the center of the current intensity of interest in Deep Learning, since they have been proven to perform far better than other algorithms in problems like image classification and object detection (figure 3.6). CNNs exhibit a high degree of invariance to translation, scaling, skewing, and other forms of distortion of its input.

Convolutional Neural Networks are constructed by utilizing a set of mathematical operations. Each operator can be seen as a layer, which receives a finite number of inputs, performs the respective mathematical operation and generates a finite number of outputs. Multiple layers are inter-connected in a directed, acyclic graph to form a CNN. Common types of layers include the convolution, pooling, inner product, normalization and activation layers. When employed for the problem of image classification, CNNs take as input an image and generate a vector of scores, indicating the class that the image belongs to. This kind of problem falls under the category of supervised learning and as such, there are two distinct computational phases: the training and the inference phase.

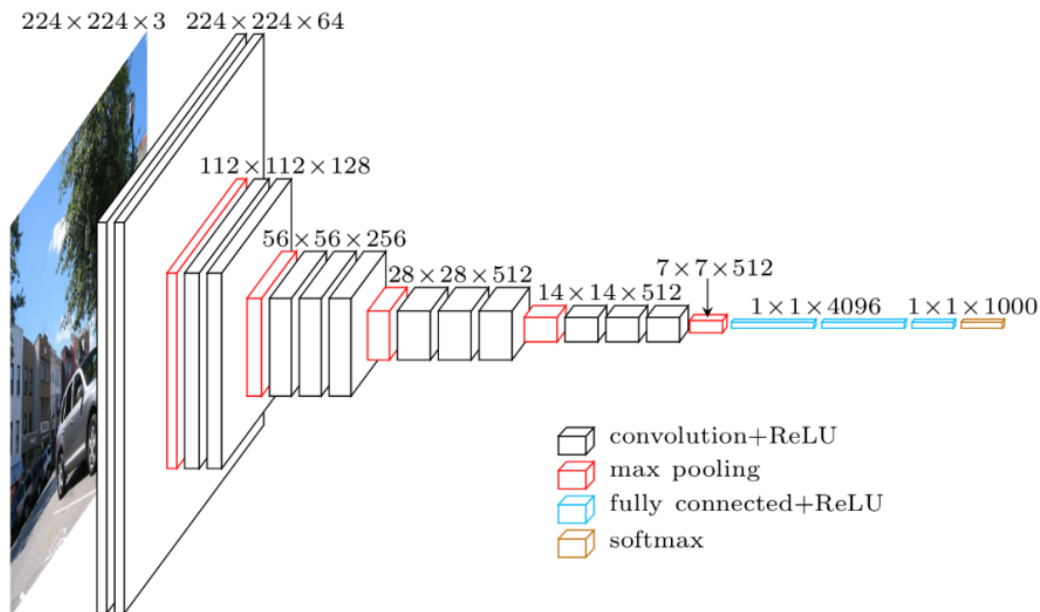


Figure 3.9: Example of a Convolutional Neural Network: Several layers are placed one after the other and the data flow in one direction (to the right). Network "VGG-16", introduced in 2014. This network achieved 92.7% top-5 accuracy in the ImageNet competition.

Compared to the Deep Feed Forward Neural Networks (DFF), Convolutional Neural Networks exhibit three unique properties: **(i)** local receptive fields of the neurons in the convolution layers, **(ii)** shared weights among the neurons of a single layer and **(iii)** spatial sub-sampling. If we examine the DFF networks, which consist of layers where each neuron is fully connected to the neurons of the adjacent layers (for example the network in figure 3.8), we can observe that they completely ignore the topology of the input. When we input an image to such a fully connected network, the order of the image's pixels does not affect the outcome of the network.¹ This comes in contrast with the strong 2D local structure that images inherently have. Convolutional Neural Networks exploit the grid topology of the input, by restricting the receptive field of the neurons in the convolution layers to a local, subset of the input, instead of each neuron being connected to every neuron of the preceding layer. The idea of connecting the units to local receptive fields on the input can be traced back to the early 60s and was strengthened based on the discovery that the neurons in the cat's visual system are also locally sensitive (Hubel & Wiesel). Beside the local connectivity, neurons within the same convolution layer share the same set of weights. This is very convenient from a computational perspective, since it results to a reduced number of weights, compared to DFFs with the same number of neurons. It also makes sense from a high level perspective, since feature detectors -e.g detecting edges, corners, etc- that are useful on one part of the image, are likely to be useful across the entire image. This property also embeds into the network the ability of being invariant to displacements of the input image. Spatial sub-sampling corresponds to the operation of the pooling layer. The high-level idea is that once a feature has been detected, its exact location becomes less important, as long as its approximate position -relative to other features- is preserved. Pooling reduces the network's output sensitivity to shifts and distortion by preserving and feeding the output of the most important neurons in each region to the next layers. Moreover, pooling improves the computational efficiency of the network because the next layer has fewer inputs to process.[Goo16] [Yan98]

These three properties of the CNNs described above result in reducing the free parameters in the network, and thus, in a reduction of the space of possible functions that the network can generate. As the capacity of the network is reduced, training the network becomes relatively easier and the chance of over-fitting is also reduced. Convolutional Neural Networks are a successful example of how information about the problem at hand can be built into the structure of the network.

¹ When we input an image to a fully connected Feed Forward neural network, each pixel of the image is connected to a different input neuron.

Data arrangement in a CNN

Each layer in a CNN takes as input a stack of C_{in} 2D matrices, of dimension $h_{in} \times w_{in}$ each; the "input feature maps". Each layer then produces a stack of C_{out} 2D matrices, of dimension $h_{out} \times w_{out}$ each, the "output feature maps". Since the input at each layer is essentially a 3D matrix, we tend to think the neurons of a CNN as being arranged in 3D (width, height, depth)¹. The feature maps are also called channels or activation maps, or activation volumes, or slices of depth, planes, etc. As a general observation, there is still not a unified nomenclature in the field of Machine Learning and several concepts and techniques keep re-appearing with different names.

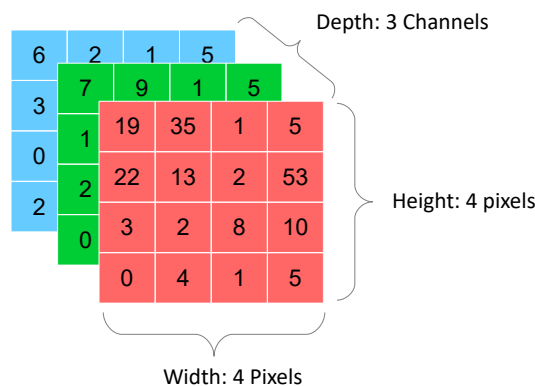


Figure 3.10: Example of an RGB image: A 3D matrix of size 4x4x3. 8 bit for each value (24 bit per pixel).

3.1.5 Common layers used to build CNNs

Convolution Layer

The convolution layer is the basic building block of a Convolutional Neural Network, as its name suggests. These are also the layers that occupy the most of the computation time.

A convolution layer extracts N *output feature maps*, from M *input feature maps*, by convolving each one of the M input feature maps with N *filters*. Each filter is of size $K \times K \times M$. Each one of the $K \times K$ 2D matrices of a filter is called a *kernel*. Each one of the $N \cdot K \times K \times M$ values that compose the N filters of a convolution layer is called a *weight*. If each input feature map is square, of size $H_{in} \times H_{in}$ and each output feature map is of

¹ The term *depth* here refers to the dimension of a single layer and not to the total number of consecutive layers in the network.

size $H_{out} \times H_{out}$, then the convolution layer consists of $H_{out} \times H_{out} \times N$ neurons (height x width x depth) and $N \cdot K \cdot K \cdot M$ weights. The receptive field of each neuron is to an are of size $K \times K \times M$ of the input 3D volume. A total of $H_{out} \cdot H_{out} \cdot N \cdot K \cdot K \cdot M$ multiplications is required for every convolution layer. The forward pass in these layers is computed as:

$$\begin{aligned}
 &\forall n = 1 : N \text{ (Number of output feature maps)} \\
 &\forall i = 1 : R \text{ (Feature map rows)} \\
 &\forall j = 1 : C \text{ (Feature map columns)} \\
 &F[n,i,j] = b[n] + \sum_{m=1}^M \sum_{x=0}^{K-1} \sum_{y=0}^{K-1} \Phi[m,i+x,j+y] \cdot w[n,m,x,y] \quad (3.4)
 \end{aligned}$$

where

- F is a tensor of output feature maps
- $b[n]$ is the bias term applied to each "pixel" of the output feature map n
- Φ is a tensor of input feature maps
- w is a tensor of pre-learned filters

Although it is called convolution layer, this layer actually performs a cross-correlation operation, also known as a sliding dot product, between each of the input feature maps and the filters' kernels. This is equal to the convolution operation with a flipped kernel in reverse row-major order. The 3D operation can be performed by adding the results of multiple 2D operations. In the equation 3.4 above, the two inner sums perform the 2D cross-correlation over an input feature map, while the external sum ($\sum_{m=1}^M$) realizes the 3D operation by adding the results of all the M input feature maps at each kernel location. In the 2D cross-correlation operation, each "pixel" of the input feature map is replaced with a linear combination of its neighbours. Figure 3.13 illustrates an example of the 3D convolution operation, while figure 3.11 illustrates how a kernel slides upon the input feature map during the 2D cross-correlation operation.

The size of each one of the output feature maps can be calculated with the following equation:

$$H_{out} = \frac{H_{in} - K + 2 \cdot P}{S} + 1 \quad (3.5)$$

where S is the stride with which the 2D kernel slides upon the 2D input feature map and P is the amount of padding used at the border of the input feature map. The size

of the output feature map is $H_{out} \times H_{out}$. This is also the number of times that the 2D kernel fits inside the 2D input feature map.

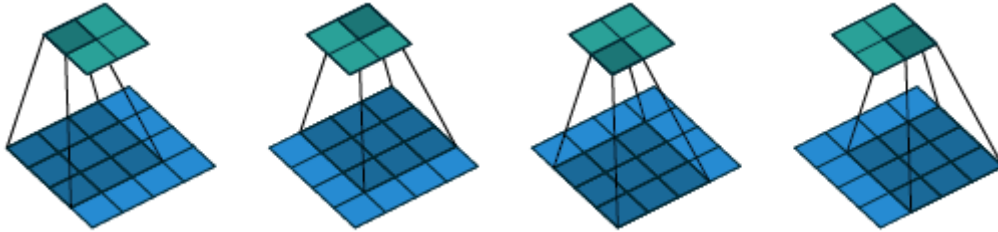


Figure 3.11: 2D Cross-correlation: a 3×3 kernel slides over a 4×4 input with unit stride and no padding (i.e. $H_{in} = 4, K = 3, S = 1, P = 0$). Results to an output of size 2×2 . Source: [Dum16]

Looking a bit ahead, the computations described in equation 3.4 exhibit a large amount of potential parallelism, since all the multiplications involved are independent of one another. That means that all the $K \times K$ multiplications within a kernel can be computed concurrently, while all the M input feature maps can be convolved concurrently, while all the N output feature maps can be generated concurrently. Moreover, all the $H_{out} \times H_{out}$ "pixels" of each of the N output feature maps can be calculated simultaneously.

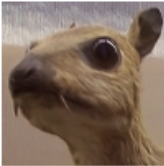

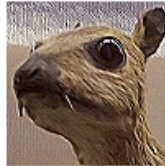
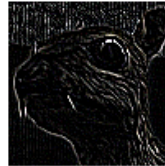
<i>Original</i>	<i>Gaussian Blur</i>	<i>Sharpen</i>	<i>Edge Detection</i>
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Figure 3.12: Convolution of an image with known kernels, from the traditional Computer Vision field: Blur, sharpen, edge detection. In Convolutional Neural Networks the kernels are the variables that will be calibrated in order to approximate a function that maps the input image to a class.

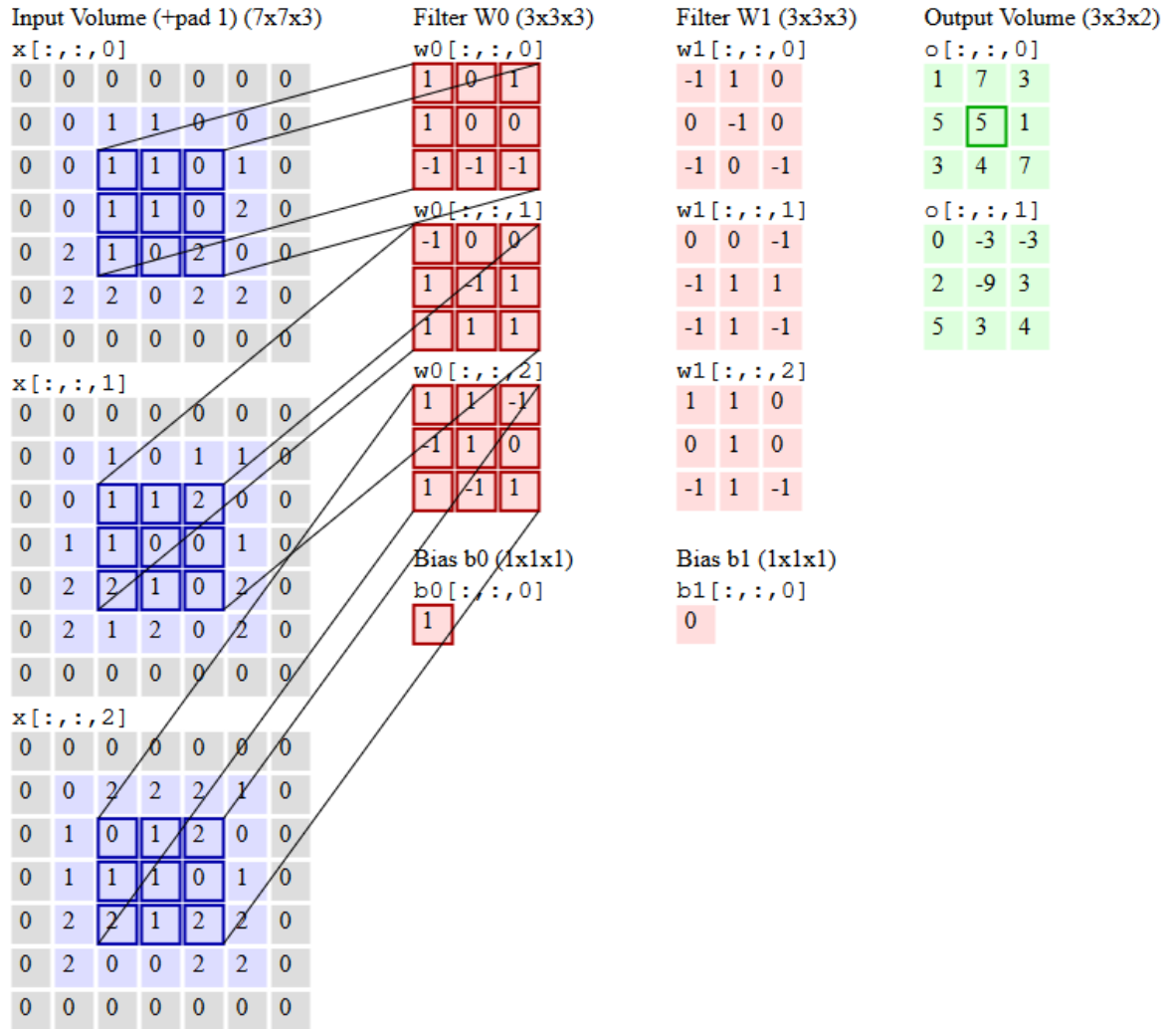


Figure 3.13: Example of 3D Convolution operation: Three input feature maps, getting convolved with two filters, generating two output feature maps. Padding=1, Stride=1. The marked "pixel" on the output feature map is a sum of all the dot products between the marked area of the input feature maps and the W0 filter's kernels. Source: [Karb]

Pooling Layer

The pooling layer operates independently on every feature map and it performs a spatial sub-sampling of its input. It is common to insert a pooling layer between successive convolution layers in a CNN architecture. The operation of this layer is similar to that of the convolution layer in the sense that there is a kernel that slides upon the input feature map. This time however, the kernel does not have a set of tunable weights. Instead, at each position it performs a pre-defined function on the corresponding "pixels" of the input feature map. Common functions are the *max* and the *average*. The pooling layer accepts a volume of size $H_1 \times W_1 \times D_1$ and outputs a volume of size $H_2 \times W_2 \times D_2$, where $W_2 = (W_1 - K)/S + 1$, $H_2 = (H_1 - K)/S + 1$ and $D_2 = D_1$. S is the stride with which the $K \times K$ kernel slides upon the feature map. The pooling layer leaves the depth dimension of its input unchanged. Figure 3.14 illustrates this procedure when the max function is utilized.

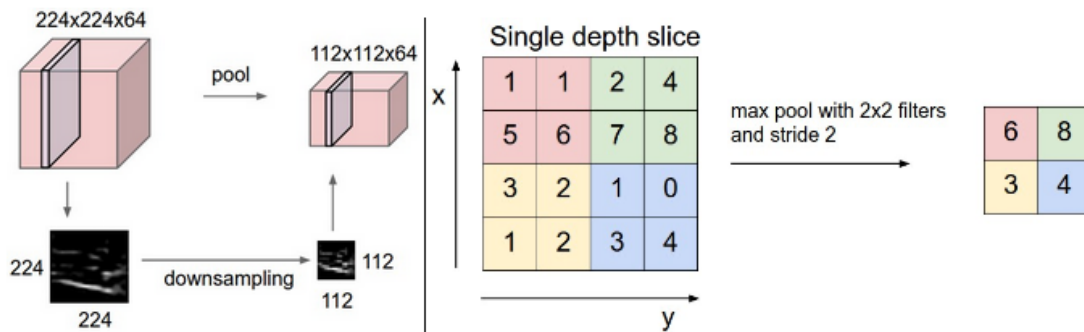


Figure 3.14: Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size $[224 \times 224 \times 64]$ is pooled with filter size 2, stride 2 into output volume of size $[112 \times 112 \times 64]$. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2×2 square). Source: [Karb]

Inner product Layer

The inner product layer, also called fully connected layer, is a regular neural network layer, as seen in Deep Feed Forward networks in section 3.1.3. All neurons in these layers are connected to all neurons in the previous layer. We think of these neurons as being arranged in a $[1 \times N]$ vector.

Normalization Layer

Many types of normalization layers have been proposed for use in Convolutional Neural Networks architectures, sometimes with the intentions of implementing inhibition schemes

observed in biological brains. However, these layers have since fallen out of favor because in practice their contribution has been shown to be minimal, if any [Karc].

Activation Layer

The activation layer is an element-wise operator that applies a non-linear function to the output of each neuron. Although this computation can be structured as a distinct layer and, for computational reasons it often is, this is a bit problematic from a conceptual point of view: The activation function is an integral part of the artificial neuron's model (figure 3.7). Without it, an Artificial Neural Network would just be performing a multiplication between its input and a weighted matrix, which means that the model of the classic ANN would degenerate to that of linear regression. Thus, an activation layer is being attached to every convolution layer and the element-wise operation is being performed at the output of every neuron. Multiple non-linear functions have been proposed and used as activation functions, most popular of which nowadays is the Rectified Linear Unit (ReLU), computing the $f(x) = \max(0, x)$. Other functions used are the sigmoid, the parametric ReLU, the Maxout, the exponential linear unit (ELU), tanh etc. Figure 3.15 illustrates several of these functions. Among these, ReLU is the easiest one to implement and the less expensive one from a computational point of view.

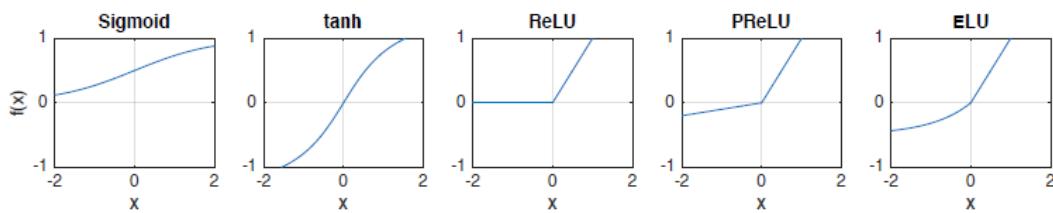


Figure 3.15: The Non-Linear Activation Functions Sigmoid, tanh, ReLU, PReLU and ELU. Source: [Gsc16]

Softmax Layer

The softmax layer is the most common classifier. A classifier layer is added after the last convolution or fully-connected layer in image classification CNNs, and normalizes the raw class scores Z_i produced from the rest of the network in the $[0,1]$ range, to interpret them as probabilities P_i according to:

$$P_i = \frac{e^{Z_i}}{\sum_{k=1}^K e^{Z_k}} \quad i = 1, \dots, K$$

3.1.6 Convolutional Neural Network Architectures

Artificial Neural Networks rely on the interconnection of multiple simple components, i.e the artificial neurons, in order to achieve overall complex computations. Issues such as the way these simple components interconnect, the number of simple components used, the choice of the learning procedure employed for network calibration, etc have been major questions since the emergence of the field in the 60s. Although our understanding has greatly advanced both in depth and range since the first ANNs were created, these questions still remain open today. The design of Artificial Neural Networks is still largely dominated by empiricism, which is why ANNs are considered in some degree a "black box" and designing them is partially viewed as an "art". Nevertheless, Artificial Neural Networks have been proven to be able to solve complex problems and perform better than other approaches, although the theoretical development -for the time being- lags behind.

Convolutional Neural Networks can be traced back at least to the start of the 90s [Lec89]. The current intensity of interest in CNNs began in 2012, when the AlexNet model outperformed in terms of accuracy the algorithms from the traditional Computer Vision field in the ILSVRC competition (figure 3.6). The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was an annual competition where participants developed algorithms to classify images from a subset of the ImageNet database. The ImageNet database consists of more than 14 million photographs collected from the Internet, each labeled with one ground-truth class. The ILSVRC training set consists of approximately 1.2 million images in 1000 different classes, covering a huge variety of objects [Rus15]. The ImageNet database became the most popular dataset upon which Convolutional Neural Networks and other Computer Vision algorithms are being evaluated. New architectures of CNNs are continually being created and new approaches and guidelines on training are continually being proposed. Several of the most well-known CNN architectures are presented in figure 3.16 and 3.17.

As one can observe in figure 3.16, the CNN architectures were getting increasingly "deeper" in every generation, as it was observed that CNNs with more convolution layers were performing better in terms of accuracy. However, as CNNs are getting bigger & deeper, the amount of memory they need and their computational load increases. Smaller CNNs are easier to train and more suited to be deployed on hardware platforms where resources are limited. SqueezeNet was one of the first CNN architectures that aimed in that direction, instead of aiming at increased accuracy.

Network	Year	#conv. layers	#MACCs [millions]	#params [millions]	# activations [millions]	Imagenet top-5 error
AlexNet	2012	5	1140	62.4	2.4	19.7%
Network-in-Network	2013	12	1100	7.6	4.0	19.0%
VGG-16	2014	16	15470	138.3	29.0	8.1%
GoogLeNet	2014	22	1600	7.0	10.4	9.2%
ResNet-50	2015	50	3870	25.6	46.9	7.0%
Inception-ResNet-v2	2016	96	9210	31.6	74.5	4.9%
SqueezeNet	2016	18	860	1.2	12.7	19.7%

Figure 3.16: Well-known Convolutional Neural Network Architectures in chronological order.

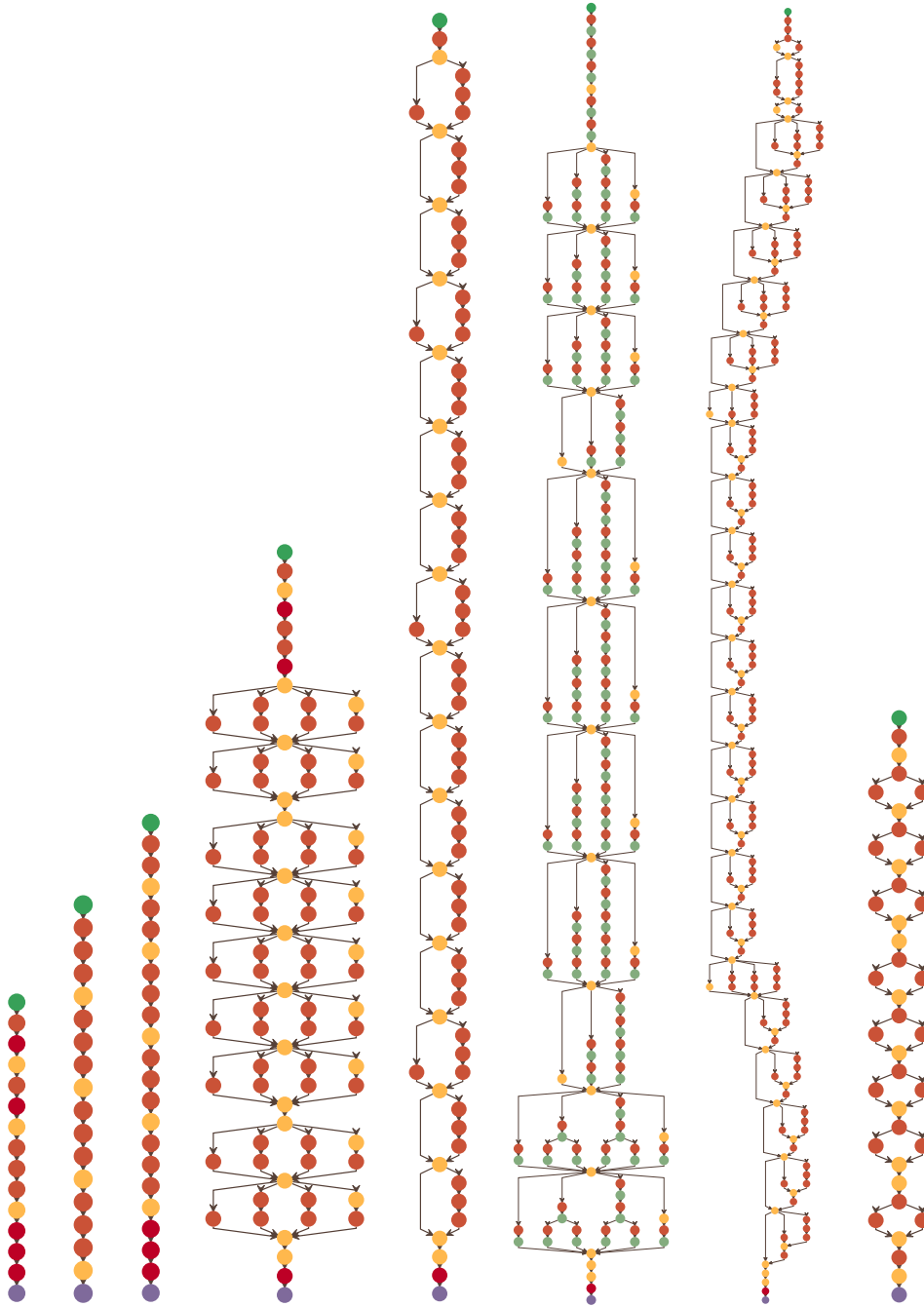


Figure 3.17: Visualization of well-known CNN Architectures. Left to right: AlexNet, Network-in-Network, VGG-16, GoogLeNet, ResNet-50, Inception v3, Inception-ResNet-v2, SqueezeNet. Data flows from top to bottom. Convolution layers are presented with brown. Source: [Gsc16]

3.2 Field-Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that were developed to answer the need for *reconfigurable hardware*. Based around a matrix of configurable logic blocks (CLBs) that are connected via programmable interconnects, FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing and perform specific tasks as a digital circuit, hence "field-programmable". Modern high-end FPGA generations feature hundreds of thousands of configurable logic blocks and additionally include an abundance of hardened functional units which enable fast and efficient implementations of common functions. Commonly integrated hard blocks are on-chip SRAM, Digital Signal Processors, PCIe as serial interconnect, Ethernet Transceivers and even full ARM processor cores, to provide software programmability, forming System on Chip boards (SoC).

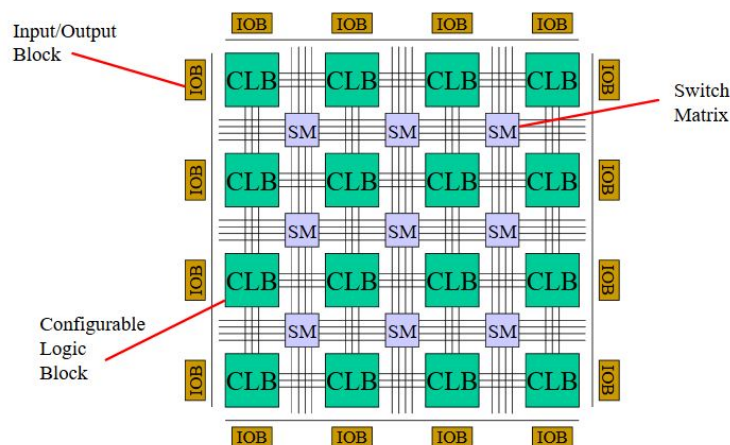


Figure 3.18: FPGA Fabric

FPGAs vs CPUs The advantage of FPGA-based systems over traditional processor-based systems such as desktop computers, smart-phones and most embedded systems, is the availability of freely programmable general-purpose logic blocks. These can be arranged into heavily specialized accelerators for very specific tasks, resulting in improved processing speed, higher throughput and energy savings. FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously. As a result, the performance of one part of the application is not affected when you add more processing because the application logic is implemented in hardware circuits rather than executing on top of an OS, drivers, and application software. This advantage comes at the price of reduced agility and increased complexity during the development, where the designer needs to carefully consider the available

hardware resources and the efficient mapping of his algorithm onto the FPGA architecture.

FPGA vs ASIC Application-Specific Integrated Circuits (ASIC) are custom-tailored semiconductor devices. In contrast to FPGAs, they do not suffer any area or timing overhead from configuration logic and generic interconnects, and therefore typically result in the smallest, fastest and most energy-efficient systems. However, the sophisticated fabrication processes for ASIC results in lengthy development cycles and very high upfront costs, which demands a first-time-right design methodology and very extensive design verification. Therefore ASIC are mostly suited for very high-volume, cost-sensitive applications where the non-recurring engineering and fabrication costs can be shared between a large number of devices. FPGAs with their reprogrammability are better suited for prototyping and short development cycles [Kae12].

FPGAs vs GPUs Both allowing massively parallel computing, the comparison between these two technologies on specific tasks in terms of speed, remains still open. GPUs run software, while FPGAs are hardware implementations. FPGAs are designed to perform concurrent fixed-point operations with a close-to-hardware programming approach, while GPUs are optimized for parallel processing of floating-point operations using thousands of small cores. FPGAs require specialized design engineers and increased developing time compared to GPU programming, on the other hand they are far more energy-efficient and, arguably, better suited for real-time applications; the latter also due to their highly deterministic nature. A nice comparison between the two technologies can be found at [Berb].

For a history of FPGAs and their place in the semiconductor universe, one can refer to [Woo17].



Figure 3.19: Flexibility vs Efficiency: CPU, GPU, FPGA, ASIC technologies.

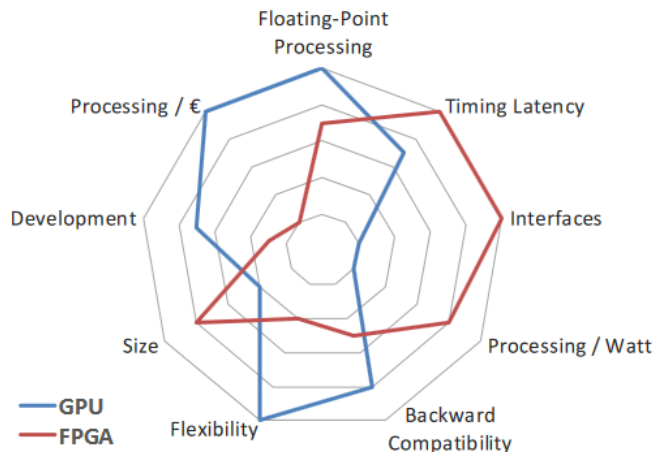


Figure 3.20: GPU vs FPGA qualitative comparison. Source: [Berb]

3.2.1 FPGA Programming

The implementation of a design in modern FPGAs requires thousands or millions of programmable switches and configuration bits set to proper state. This task is not achieved through manually reprogramming each element individually -that would be too complex and time consuming. Instead, the designer tackles the specific problem at higher levels of abstraction: Either by describing the specific circuit using a Hardware Description Language, for instance, VHDL or Verilog, or at an even higher level of abstraction, by describing the desired algorithmic behavior in a high-level language such as C, in a procedure known as High-Level Synthesis. The whole process is highly dependent on software tools, responsible for simulating, synthesizing, mapping, placing and routing the design on a target device and also calculating various metrics regarding the implementation's performance. In fact, the whole process would not be feasible without the use of dedicated software suites.

In this thesis, the digital circuits were created with the use of VHDL -standing for Very High-Speed Integrated Circuit Hardware Description Language- and Xilinx's "Vivado" software. As its name suggests, VHDL is a language invented to describe hardware. In contrast with most higher-level computer languages that are used to describe algorithms and are sequential, in VHDL instructions are all executed *concurrently*. The steps to map a design on an FPGA are briefly described below and are also illustrated in figure 3.21:

The first stage of synthesis converts the circuit description from an HDL file into a netlist

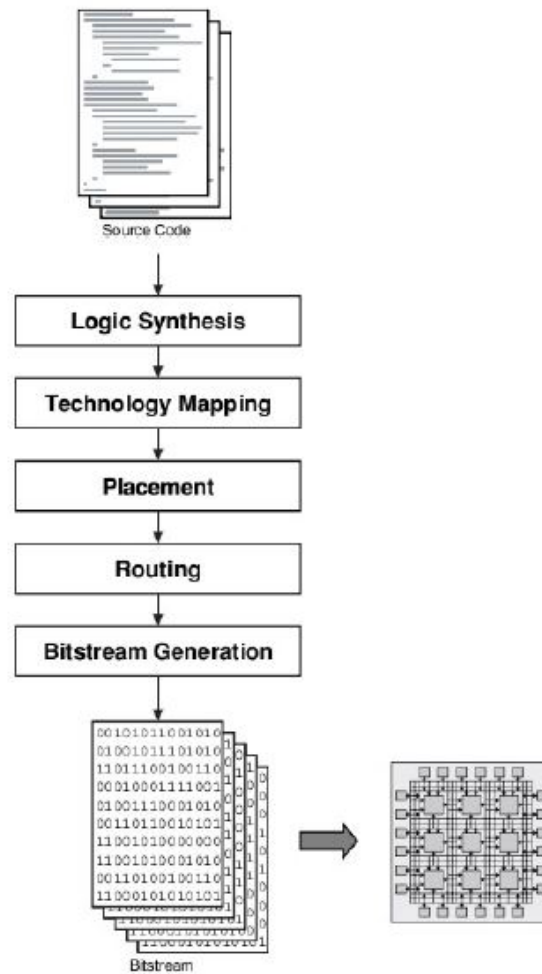


Figure 3.21: FPGA mapping flow

of basic gates. This netlist is converted then into a netlist of FPGA logic blocks according to the desired synthesis properties (speed, area or power specifications). During this stage, several optimizations take place removing redundant logic and simplifying the design. After the synthesis stage, the implementation stage follows where the netlist is translated into a placed and routed FPGA design. During the physical design stage, in the technology mapping stage, several LUTs and registers are packed into one logic block respecting limitations imposed by the FPGA platform. In this stage, a number of optimizations are available depending on the goals the designer has chosen. Important optimizations are LUT combining in order to minimize resource utilization and minimize number of signals to be routed between logic blocks. Once the circuit has been mapped on a specific device, the placement stage begins where heuristic placement algorithms

determine which logic block within the FPGA should implement each of the logic blocks required by the circuit. The optimization goals are to place connected logic blocks close together to minimize the required wiring (wirelength-driven placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement). After the location for all the logic blocks in the circuit has been chosen, it is necessary to program the switches on the device to be used in order to connect all logic block input and output pins required by the circuit. In this stage, the routing architecture of the device is represented as a directed graph. Thus, routing a connection corresponds to finding a path in this routing-resource graph. Since most of the delay in FPGA designs is routing delay, a timing-driven optimization in the routing stage is crucial to minimize overall circuit delay [23]. Finally, after the design has been successfully placed and routed (PAR) on the chosen FPGA, the design tool creates a bitstream of the final design after PAR which is then downloaded to the FPGA and configures the device accordingly.

3.2.2 FPGA Fabric

Although the internal structure of an FPGA is technology-specific, the various fabrics share common design concepts. In this section basic properties of FPGA's structure will be presented, based on Xilinx's 7-Series.

CLB and their Arrangement

In Xilinx's 7 Series, CLBs are arranged in columns. Each CLB element contains a pair of *slices*, and each slice is composed of four 6-input LUTs and eight storage elements. Carry chains run vertically in a column from one slice to the one above as illustrated in 3.22. Connections between CLBs and other resources use the fabric routing, extending vertically, horizontally and diagonally.

The structure of a slice can be seen in figure 3.24 and a simplified diagram in 3.23. Each slice contains:

- Four 6-input Look Up Tables (LUT), also called logic-function generators
- Wide-function multiplexers
- Carry logic
- Four flip-flop/latches
- Four additional flip-flops

Generally, an N-input-LUT is a functional unit capable of computing any function of N inputs. The operation of a LUT resembles the process of finding the value of a logical function via its truth table. Given the truth table of a function, the LUT is programmed

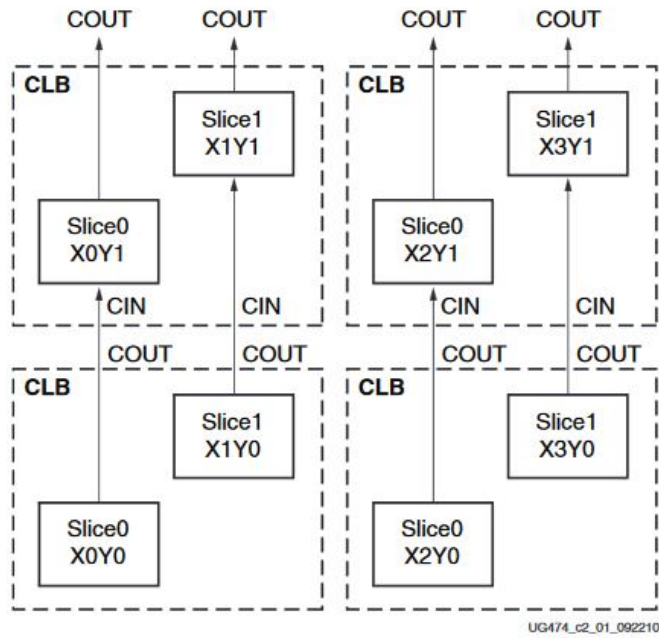


Figure 3.22: Row and Column Relationship between CLBs and Slices. Source: Xilinx’s User Guide

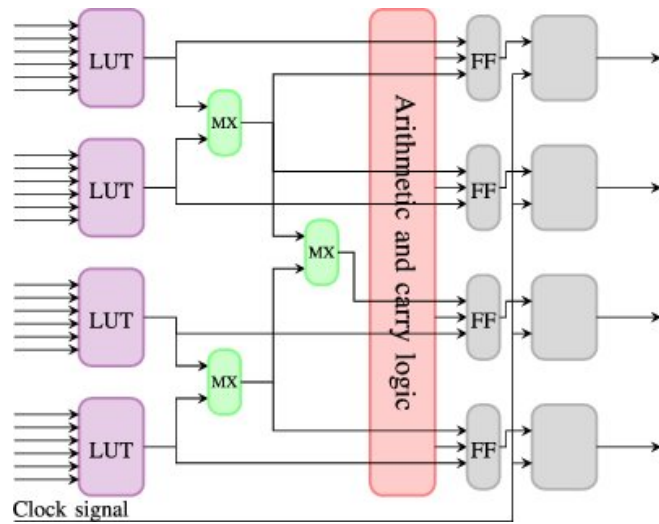


Figure 3.23: Simplified diagram of an FPGA slice. Since the Xilinx Virtex-5 generation introduced in 2006, the slices contain 6-input LUTs.

accordingly. Then it is responsible for matching a pattern of the N inputs with one of the 2^N rows of the table and generate the corresponding output value. LUTs can be combined to implement more complex functionalities than a N -bit logical function. Specifically, a LUT is able to implement a logical function of N inputs, a N -bit shift register or, alternatively be used as N -bit distributed memory.

In 7 Series each LUT has 6 inputs, meaning that a single LUT can implement:

- Any arbitrarily defined six-input Boolean function
- Two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs
- Two arbitrarily defined Boolean functions of 3 and 2 inputs or less

The slice may also be used to implement a synchronous RAM resource, called a distributed RAM element, or a shift register.

Hardwired Blocks

As already mentioned, configurable logic blocks serve as the main functional unit of an FPGA, with the look up tables playing an important role in their operation. However, it is currently the rule for an FPGA to have common functionalities embedded into the silicon, in order to reduce the required area and provide increased speed compared to building those functionalities from primitives. Examples of hardwired blocks include multipliers, generic DSP blocks, embedded processors, high-speed I/O logic and embedded memories.

FPGA boards are equipped with various memory elements that can be utilized as RAM, ROM or shift registers. One of these elements is the look up table which is discussed in the previous sub-section. Flip-flops also serve as a basic storage unit in an FPGA design. Another significant memory element is the BRAM (Block RAM). The BRAM is a dual-port RAM component which is embedded into the FPGA board and can achieve storage of a large set of data. The capacity of block RAMs usually instantiated is 18KB and 32KB. Of course, each and every board comes with a specific number of embedded BRAMs [3]. A key element in BRAMs is the dual-port operation which is introducing a parallel behavior as it is providing access to different locations in the same clock cycle.

One of the most important and complex computational unit embedded into the FPGA fabric is the DSP (Digital Signal Processing) Block. The usage of embedded DSP blocks has been established in order to support the increasing amount of computational load. A DSP block is a combination of adders, subtractors and multipliers put together to compose an arithmetic logic unit (ALU).

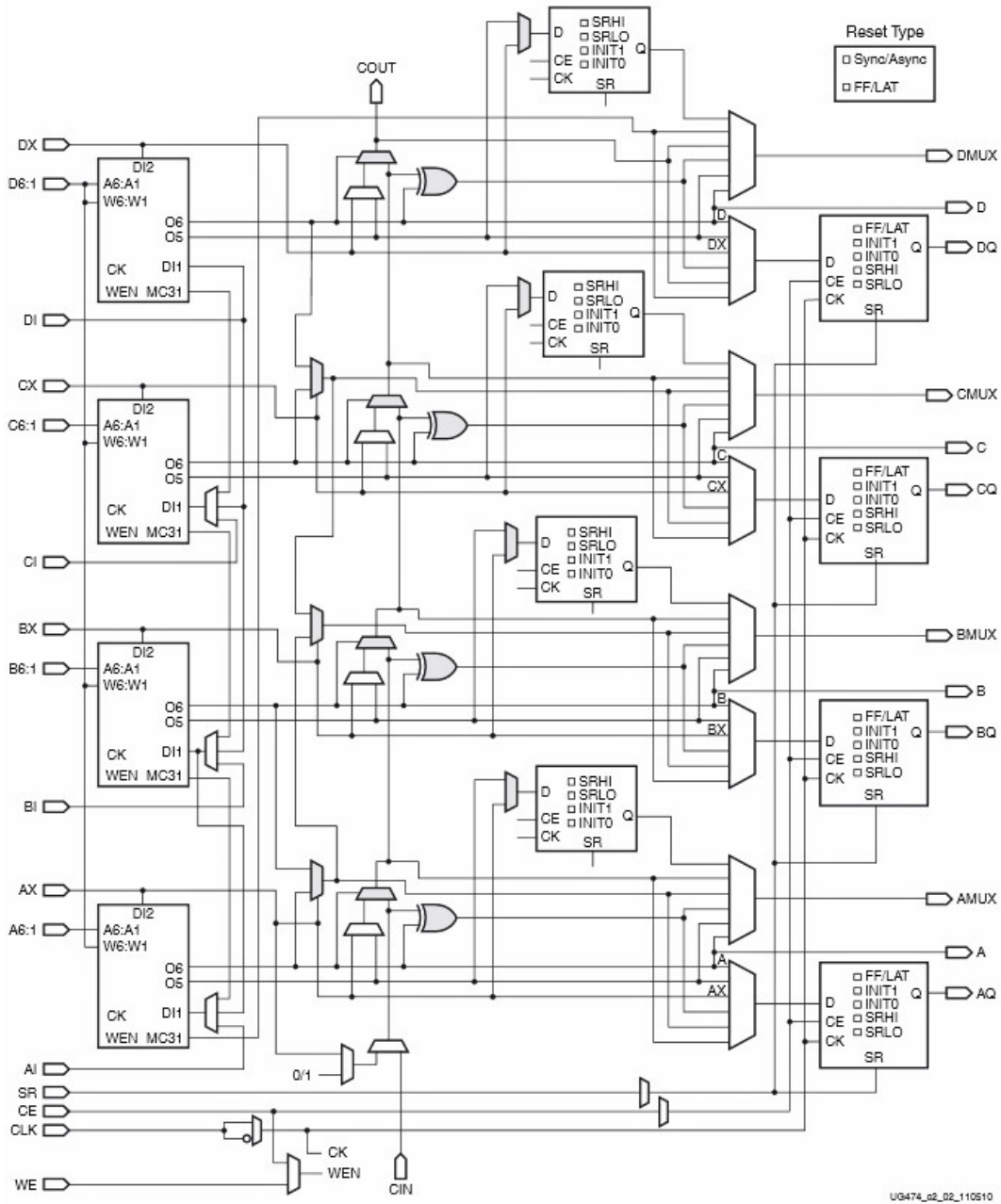


Figure 3.24: Internal structure of a single FPGA slice.

3.3 Convolutional Neural Networks in FPGAs

Currently, in most cases, the training of Convolutional Neural Networks is done offline using GPU-based training systems or CPUs. As far as the inference stage, i.e the forward pass, of CNNs is concerned, FPGAs have been identified as an exceptional candidate for Neural Networks implementations, for multiple reasons:

- The FPGA architecture can be customized for specialized parallel computing, naturally matching Artificial Neural Networks, which are inherently massively parallel computing systems and algorithms, with very few conditional branch operations.
- In many computer vision applications the computing systems must be small in physical dimensions, have low power consumption and be able to operate within demanding timing constraints. FPGAs offer deterministic timing in their execution and relatively low power consumption. [Berb]
- Currently there is plenty of research in Deep Neural Networks and new techniques and architectures are constantly being proposed. FPGAs can be reconfigured to account for the new functionalities needed, in order to implement the state-of-the-art CNNs of each time.
- CNNs have been shown to work well with limited numerical precision. FPGAs can be configured with non-standard word-lengths, optimized for each network and application. This allows both less memory requirements and denser logic, which in turn leads to an increase to the amount of parallel processing blocks and thus faster execution.

Over the past few years many hardware accelerators for Deep Neural Networks have been proposed and a few a commercial solutions have been made available. Recently, several reviews examining the approaches of designing hardware CNN implementations in FPGA were published [Abd18a], [Guo17], [Zha18].

When porting a CNN to an FPGA device, the problem boils down to finding an efficient mapping between the computational model of the CNN and the execution model supported by the FPGA. In figure 3.25 the main strategies to address this mapping problem are illustrated. Current FPGA-based accelerators for CNNs rely on at least one of the following optimization techniques to efficiently infer CNNs.

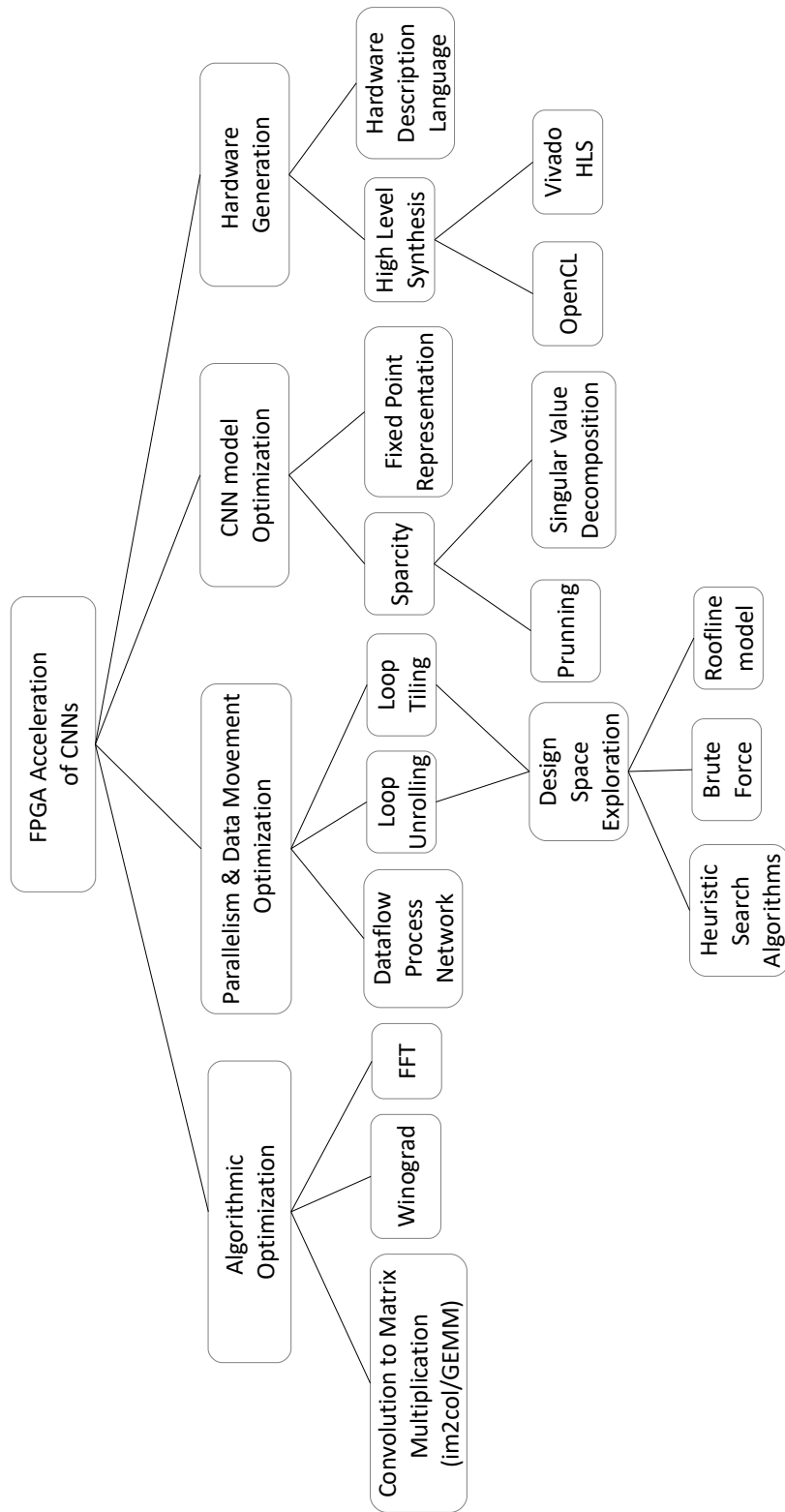


Figure 3.25: Main approaches to accelerate CNN inference on FPGAs.

CHAPTER 4

CNN training and Optimization

This chapter's goal is threefold: First, we will briefly present Berkeley's "Caffe" framework for training and deploying deep models. Then, a case study of training a CNN over the SAT-4 & SAT-6 airborne datasets will be presented. Finally, some optimizations of the chosen network architecture from the hardware implementation perspective will be discussed and explored.

4.1 The Caffe Framework

There are many popular software frameworks specifically built for the design and training of neural networks, including, among others, the Neural Network Toolbox for MATLAB, Theano, Torch, TensorFlow and Caffe. Most of these frameworks, apart from CPUs, they can utilize one or multiple GPUs in order to heavily accelerate the training of neural networks. In this thesis, we used the Caffe framework.

Caffe is a deep learning framework developed by Berkeley AI Research (BAIR) and by community contributors, first introduced in 2014. It powers ongoing research projects, large-scale industrial applications, and startup prototypes in vision, speech, and multimedia. The framework is a BSD-licensed C++ library with Python and MATLAB bindings for training and deploying general-purpose convolutional neural networks and other deep models. [Jia14]. Using CUDA GPU computation it can process over 60 million images per day with a single NVIDIA K40 GPU running at 235 Watt, meaning about 1 ms/image for inference and 4 ms/image for learning, for images of the ImageNet competition (256 * 256 pixels [Bera]). For the purposes of this thesis, it is worth to note that the NVIDIA Tesla K40 GPU needs about two orders of magnitude more energy than the Xilinx Zynq-7000 SoC targeted in this thesis.

Caffe offers a wide variety of layers, implementing various mathematical operations, and an extensive library of pre-trained models in its "Model Zoo". There are four steps in training a Convolutional Neural Network using Caffe:

1. **Data preparation:** Image pre-processing and storage in a format that can be used by Caffe.
2. **Model definition:** Choosing a CNN architecture and defining its parameters in a configuration file with extension ".prototxt".
3. **Solver definition:** Defining the solver parameters in a configuration file with extension ".prototxt".
4. **Model training:** Caffe will iteratively train the chosen model on the provided data and will output two files: a file with extension ".caffemodel" and a file with extension ".solverstate", both being binary files. The first one is generated at a specified interval while training and contains a snapshot of the network's weights, while the latter contains the information required to continue training the model from where it last stopped. During this procedure, the user can monitor the network's performance on the training and test sets. After the training phase, the ".caffemodel" file can be used in deploying the model and make predictions on new, unseen data.

We will see a bit more of how to use the Caffe framework, dealing with a specific problem of image classification, utilizing the SAT-4 & SAT-6 airborne datasets.

4.2 The SAT-4 & SAT-6 Airborne datasets

SAT-4 and SAT-6 are publicly available, high resolution satellite multispectral datasets, assembled for addressing the problem of satellite image classification. In general, extracting information regarding the various terrain objects, land cover and land usage status, is valuable for a wide range of applications in areas like urban planning, agriculture, geology and environmental studies. DeepSat contains patches extracted from the National Agriculture Imagery Program (NAIP) dataset with about 330,000 scenes spanning the entire Continental United States. The average image tiles in the (NAIP) dataset are 6000 pixels in width and 7000 pixels in height, measuring around 200 megabytes each. The images consist of 4 bands: red, green, blue and Near Infrared (NIR) and were acquired at a ground sample distance (GSD) of 1 meter, having horizontal accuracy up to 6 meters. The patches in SAT-4 and SAT-6 were sampled from a multitude of scenes, covering different landscapes like rural areas, urban areas, densely forested, mountainous terrain, small to large water bodies, agricultural areas, etc. covering the whole state of California, with each patch being of size 28x28. Each of these datasets is split in train and test sets. SAT-4 has four different classes which are: barren land, trees, grassland and a class that consists of all land cover classes other than the above three. The SAT-4 dataset consists of 400,000 training and 100,000 testing patches. SAT-6, contains six

different classes: barren land, trees, grassland, roads, buildings and water bodies and consists of 324.000 training and 81.000 testing patches [Bas15].

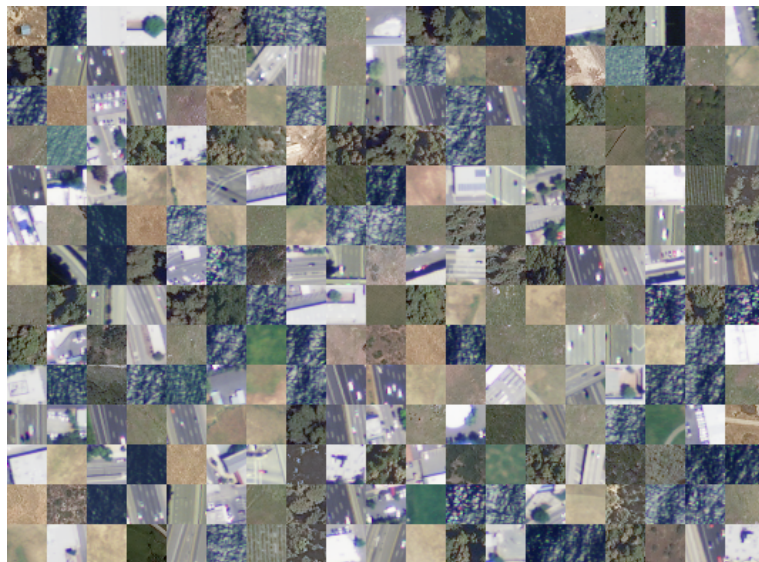


Figure 4.1: Sample images from the SAT-6 dataset. Source: [Bas15]

4.3 CNN models in this Thesis

4.3.1 The original "Cifar-10 Full" CNN model

"Cifar-10 Full" is a, reproduced in Caffe, Convolutional Neural Network model, created by Alex Krizhevsky. It takes its name after the respective "Cifar-10" dataset that it was initially targeting. The dataset consists of 60000, 32x32 colour images in 10 classes. The model can be found in the Caffe source code at "[examples/cifar10/cifar10_full_train_test.prototxt](#)". It composes layers of convolution, pooling, rectified linear unit (ReLU) nonlinearities, and local contrast normalization with a softmax classifier on top of it all.

Cifar-10 Full is essentially the model that we trained to classify the images from the SAT-4 & SAT-6 ariborne datasets. We customized the model to fit these datasets -i.e the input image size, the number of output classes- and we also customized it based on design decisions from the hardware design perspective. These changes in the model will be explained in the next section 4.3.3. First, let's take a look at the original structure.

As we can see in figure 4.2, there are three convolutional layers, each one followed by a layer of pooling, a rectified linear unit and, in the case of the first two convolution layers (Convs), a normalization layer. All the kernels used in the convolutions are of size 5x5. The pooling operation is applied on kernels of size 3x3, utilizing either a max function, or an average function. The rectified linear unit layer applies the function $f(x) = \max(x, 0)$ on each element of its input. The normalization layer is a Local Response Normalization

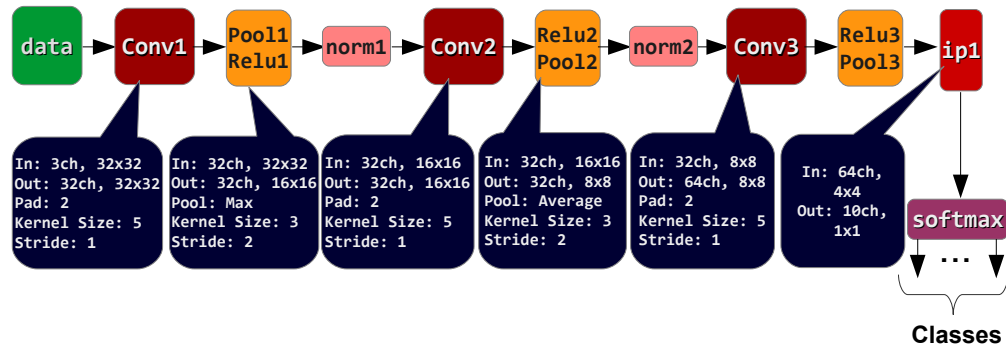


Figure 4.2: Original "Cifar-10 Full" CNN Model structure

(LRN), which performs a kind of “lateral inhibition” by normalizing over local input regions. The network has 32 filters of size 5x5x3 in the first convolution layer, 32 filters of size 5x5x32 in the second convolution layer and 64 filters of size 5x5x64 in the third convolution layer, for a total of about 80 thousand learnable parameters.

In the prototxt file that describes the model’s architecture, each layer is assigned a name, for example "conv1". The flow of data is described by determining the previous layer, namely the "bottom" one, from which the data come. An example of how a Convolution layer looks like within a prototxt file can be seen below. A detailed list of the supported layers can be found at [Caffe Layers](http://caffe.berkeleyvision.org/tutorial/layers.html)¹.

```

1
2 layer {
3   name: "conv1"
4   type: "Convolution"
5   bottom: "data"
6   top: "conv1"
7   param {
8     lr_mult: 1           # learning rate multiplier for the filters
9   }
10  param {
11    lr_mult: 2           # learning rate multiplier for the biases
12  }
13  convolution_param {
14    num_output: 32       # learn 32 filters
15    pad: 2               # number of pixels to add to each side of the input (
zero-padding)

```

¹ caffe.berkeleyvision.org/tutorial/layers.html

```
16     kernel_size: 5      # each filter is 5*5
17     stride: 1          # step 1 pixel between each filter application
18     weight_filler {
19         type: "gaussian" # initialize the filters from a Gaussian distribution
20         std: 0.0001     # with stdev 0.01
21     }
22     bias_filler {
23         type: "constant" # initialize the biases (default: zero)
24     }
25 }
26 }
```

The other prototxt file needed, defines the solver parameters. It can be found in Caffe's source code, at "[examples/cifar10/cifar10_full_solver.prototxt](#)". Beside the solver's parameters, in this file we can determine how frequently will the framework pause the training process in order to test the model's performance and when to create snapshots of the model's state. Alex Krizhevsky who created this solver targeting the "Cifar-10" dataset also determined the number of training epochs at which lowering the learning rate can help the algorithm converge to a better minimum.

4.3.2 Why use the "Cifar-10 Full" model?

The Cifar-10 Full model delivers high accuracy for the targeted dataset while being small enough to be stored in on-chip memory. Therefore, this model is a good candidate to address the realistic problem of real time classification of SAT6-like images. Even though Cifar-10 Full is not as deep as more recent models (e.g. ResNet), Cifar-10 Full cannot be regarded as outdated as long as it fulfills its purpose. Overall, Cifar-10 Full is a good fit for the purposes of this thesis. More complex models have demonstrated similar or slightly better results in accuracy on the SAT-4 & SAT-6 datasets (figure 4.3), but one should not use a bazooka to kill a mosquito. Finally, the basic problems that emerge in implementing this model in an FPGA, using VHDL, also present themselves in models of higher complexity, which makes Cifar-10 a good place to start.

The exact complexity of the model that we used is presented in detail in section 4.3.4. For reference purposes and to enable comparison, table 4.1 presents info on some of the most popular CNN models and an entry of how would the specs of the "Cifar-10 Full" model scale, in case it was used for images of 224x224x3 pixels size (the model originally aimed the Cifar dataset, with images of size 32x32x3).

Table 4.1: Comparison of Different CNN Topologies for Image Classification on ImageNet dataset. Source: [Gsc16]. The "Cifar-10 Full" model is included with its input image dimensions altered to 224x224, to get a better feeling of the network's size compared to the other networks.

	Image Size	#conv layers	#MACCs (millions)	#Params (millions)	#Activations (millions)	ImageNet top-5 error
AlexNet	227x227x3	5	1140	62,4	2,4	19,70%
Network-in-Network	224x224x3	12	1100	7,6	4	19,00%
VGG-16	224x224x3	16	15470	138,3	29	8,10%
GoogLeNet	227x227x3	22	1600	7	10,4	9,20%
ResNet-50	224x224x3	50	3870	25,6	46,9	7,00%
Inception v3	299x299x3	48	5710	23,8	32,6	5,60%
Inception-ResNet-v2	299x299x3	96	9210	31,6	74,5	4,90%
SqueezeNet	227x227x3	18	860	1,2	12,7	19,70%
Cifar-10 Full (scaled)	224x224x3	3	604	0,58	4,4	not available

4.3.3 Customizing the "Cifar-10 Full" model to our needs

Adjusting the Input & Output to the SAT datasets

This is an obvious change; Cifar-10 model is targeting a dataset with 10 classes, while SAT-4 and SAT-6 have four and six output classes respectively. Moreover, the Cifar-10 dataset consists of RGB images of 32x32x3 pixels size, while the SATs contain images of 28x28x4 pixels size, with four channels: RGB-NIR.

Removing the NIR-input channel

Although in hyperspectral and multispectral imaging the Near Infrared (NIR) band offers valuable information in identifying different materials, in our case we decided to remove the NIR channel. Initially this was a trial & error decision, based on the fact that most of the existing models for image classification in the Caffe's "Model Zoo" take as an input images with three channels in depth and we wanted to use a mostly pre-defined network. The top-1 accuracy that the model achieved in the preliminary training on the SAT-4 & SAT-6 datasets, while omitting the NIR-channel, exceeded our expectations and was deemed enough for our case (over 90% top-1 accuracy). Thus, we kept the R,G,B subset and no permutations were tested. This decision was further strengthened based on the results of [Pap16], where training a deep CNN model on the SAT-4 & SAT-6 datasets, with and without the NIR band, proved to produce similar results in terms of accuracy. From a hardware design perspective, this reduction in the dimensionality of the input space is both convenient and important: less data as input, means fewer computations and less memory needed.

Beside the specific datasets used in this thesis, there is also one more reason why one should consider favoring the R-G-B channels over NIR, when available: Most models are trained on images that use channels from the visible spectrum, as in the case of the ImageNet dataset (RGB and CMYK format). In order to use pre-trained models and utilize "transfer learning" techniques, the target dataset *should be similar* to the one originally used. Transfer learning can be highly useful, when training data are scarce, or when we want to reduce computational time for training. In our case, the models were trained from scratch, since there were plenty of data available in the SAT datasets.

Removing the Local Response Normalization layers

The contribution of these layers to the accuracy of CNN models has been in doubt for the past few years and they seem to have fallen out of favor [Karc] [Che16a]. Although Alex Krizhevsky mentioned a 2% increase in the accuracy for Cifar-10 when using the normalization layer [Kri12], removing the layer didn't have any significant impact in our case. Thus we regarded that the hardware implementation of the model should not spare resources for this kind of layer. Moreover, it should be noted that although the LRN layers have fewer Multiply-Accumulate operations than the convolution layers, they use division with non-constant numbers, which is computationally and resource expensive.

Using only "max" operation for Pooling

The "Cifar-10 Full" model alternates between average pooling and max pooling in its layers. To have a more uniform network we decided to keep only one kind of pooling and arbitrarily chose the max-pooling operation.

Order of Pooling & Rectifier Linear Unit layers

In the original "Cifar-10 Full" model we can observe that the Pooling and Relu layers do not always appear in the same order. This decision did have some impact when the average pooling function was used. However, in the case of max pooling and relu (x^+), if we suppose a submatrix B of a real matrix A, then $max_pooling(relu(B)) = relu(max_pooling(B))$. Since we decided to use only the max pooling function, we can re-order these two layers. Although the layer's result will be the same regardless the order, these two cases are not completely equivalent: In the case of $max_pooling(relu(B))$, the relu operation will have to be applied to all the MxN elements of the submatrix B, whereas in the case of $relu(max_pooling(B))$ it will have to be applied only to one element, the scalar result of max_pooling. Thus, Pooling layers in our network should be placed before the Relu layers.

Pooling layer's output size

The Caffe framework does not calculate the output size of the convolution layer and the pooling layer in exactly the same way. In the case of the convolution layer, a floor function is used to make sure that the calculated dimension is an integer number, whereas in the case of pooling the ceiling function is used. Output's size in one dimension is:

$$\text{Pooling Output_height} = \text{ceil}\left(\frac{\text{Input_height} - \text{Kernel_height}}{\text{Stride}} + 1\right)$$

When the fraction is not an integer number itself, Caffe implements the ceiling function by zero-padding half of the input plane's matrix sides (figure 4.3).

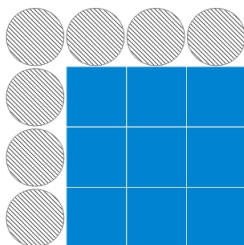


Figure 4.3: Half zero-padding the input plane. Ceiling pooling layer's output size.

In order to avoid the extra logic needed to conditionally handle the incoming pixels in the pooling layer, we slightly modified the pooling kernel size wherever needed, so that the fraction in the equation above would be an integer and thus zero-padding would not be needed in this stage. Specifically, we changed the pooling kernel in layers 1&2 to height=2, but kept the kernel in the third layer with height=3. However, it should be relatively easy to include this functionality if needed, since a FSM for zero-padding is already written for the convolution layers.

4.3.4 The "Modified Cifar-10 Full" CNN model

Having in mind the adjustments mentioned in the subsections above, the network that we used has the architecture illustrated in figure 4.4. The CNN model is identical for both the SAT-4 & SAT-6 datasets, with the exception of the number of output classes. Using [Netscope](https://github.com/dgschwend/netscope)¹, a web-based CNN analyzer, we can quickly calculate the network's complexity in terms of number of operations and memory (table 4.2). "Activations" is the

¹ [dgschwend.github.io/netscope/quickstart.html](https://github.com/dgschwend/netscope/quickstart.html)

layer's number of outputs, while "parameters" is the number of learnable weights. The number of Multiply-Accumulate (Macc) operations per convolution layer is calculated as: Number of Input Channels \cdot (Width of the kernel)² \cdot Number of Filters in the Layer \cdot (Width of the Output plane)². For example, in the first convolution layer (conv1) that is $3 \cdot 5 \cdot 5 \cdot 32 \cdot 28 \cdot 28 = 1881600$ Macc operations.

In figure 4.5 we can see that the second convolution layer is the most computationally exhausting, while the third convolution layer has the largest number of weights.

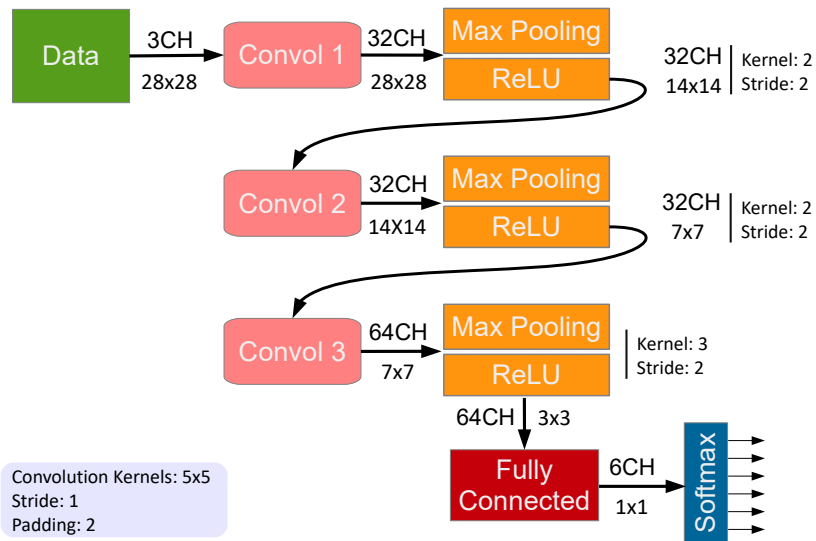


Figure 4.4: The "Modified Cifar-10 Full" CNN model: based on the "Cifar-10 Full".

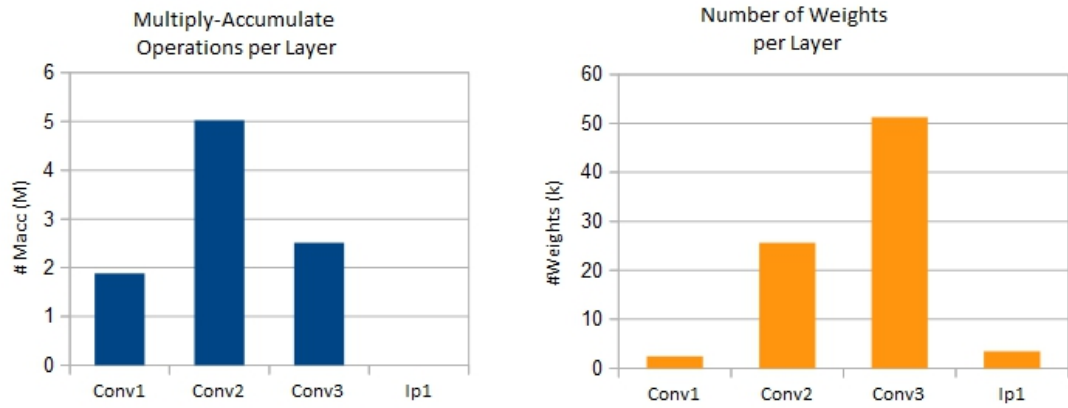


Figure 4.5: Number of Operations & Weights per layer

Table 4.2: "Modified Cifar-10 Full" CNN: Number of Operations & Network's Size

Name	Type	Channels In	Dimension In	Channels Out	Dimensions Out	Operations	Memory
data	Input	3	28x28	3	28x28		activation: 2.35k
conv1	Convolution	3	28x28	32	28x28	Macc: 1.88M	activations: 25.09k parameters: 2.43k
pool1	Max Pooling	32	28x28	32	14x14	Comp: 25.09k	activations: 6.27k
relu1	ReLU	32	14x14	32	14x14	Comp: 6.27k	activations: 6.27k
conv2	Convolution	32	14x14	32	14x14	Macc: 5.02M	activations: 6.27k parameters: 25.63k
pool2	Max Pooling	32	14x14	32	7x7	Comp: 6.27k	activations: 1.57k
relu2	ReLU	32	7x7	32	7x7	Comp: 1.57k	activations: 1.57k
conv3	Convolution	32	7x7	64	7x7	Macc: 2.51M	activations: 3.14k parameters: 51.26k
pool3	Max Pooling	64	7x7	64	3x3	Comp: 5.18k	activations: 576
relu3	ReLU	64	3x3	64	3x3	Comp: 576	activations: 576
ip1	Inner Product	64	3x3	6	1x1	Macc: 3.46k	activations: 6 parameters: 3.46k
prob	Softmax	6	1x1	6	1x1	add: 6 div: 6 exp: 6	activation: 6
TOTAL						Macc: 9.41M Comp: 44.96k	activations: 53.69k parameters: 82.79k

4.3.5 The "Modified Cifar-10 Full" CNN model's Training & Results

There are plenty of tutorials online on how to install and use the Caffe framework for training. In this thesis, the framework's command line interface and python interface were used.

SAT4 & SAT6 datasets are pre-split in train and test sets. Having in mind that they were created and published for being used by machine learning algorithms, no cleansing, augmenting, or transforming the data was applied for this thesis. The only pre-processing applied before training was the per channel mean subtraction, calculated over all the images, as common practice suggests. For the solver configuration, which requires the choice of several hyper-parameters that heavily influence the learning process, we used the configuration described for the "Cifar-10 Full" as a starting point and adjusted the total number of epochs and the learning rate by observing the network's performance. This method falls under the category of manually searching the hyper-parameter space and it is a bit primitive. However, luck was on our side and the algorithm quickly converged, thus no further exploration was carried out. Although not in the scope of this thesis, it should be noted that methods of grid search, random search and various methods of optimization to tackle the very interesting problem of selecting the hyper-parameters of a machine learning problem do exist.

At about four epochs¹ of training (with batch size=100), the results obtained for the SAT-6 dataset were those illustrated in figure 4.6. We can see that at around 8000 iterations the model's accuracy gradually stopped improving and reached a plateau. The model reached a top-1 accuracy of 98.5% after just 3 epochs of training, which is pretty fast. After a total of 6 epochs of training and having lowered the learning rate by a factor of 10, the algorithm achieved a top-1 accuracy of 98.8% for the SAT-6 test dataset. It is exciting how quickly the algorithm converged, although we didn't use a pre-trained model, but rather initiated the training from scratch. The results for the SAT-4 dataset were similar, achieving a 99.09% top-1 accuracy. Comparative results of applying different learning frameworks on these two datasets can be seen in table 4.3.

1 One epoch consists of one full training cycle on the entire training set. Here, with batch size =100, a training epoch is equal to 3240 iterations, since the dataset consists of 324000 training samples.

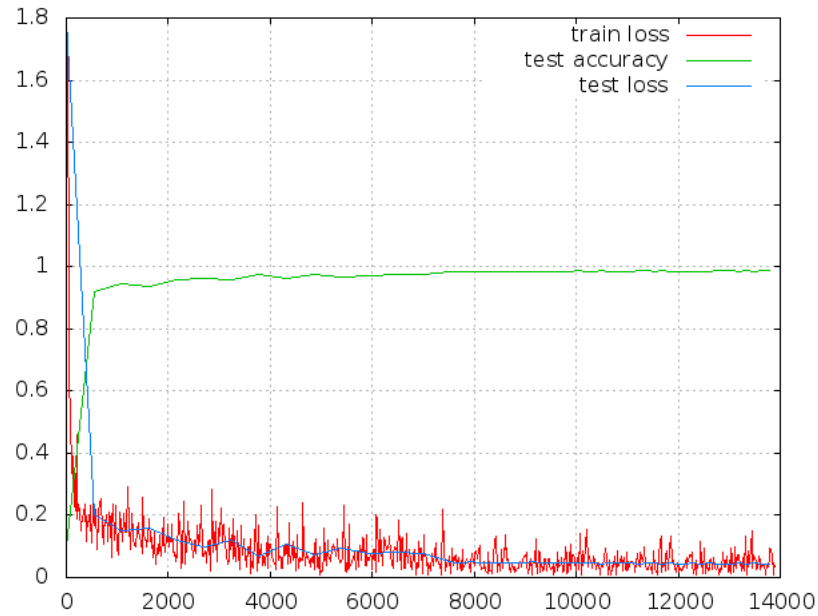


Figure 4.6: Training loss and accuracy of the "Modified Cifar-10 Full" CNN model, for the SAT-6 dataset after 4 epochs of training.

Table 4.3: Accuracy rates for SAT-4 & SAT-6 datasets using different learning frameworks [Pap16].

Method	Overall Accuracy %	
	SAT-4	SAT-6
DBN [Bas15]	81.78	76.41
CNN [Bas15]	86.83	79.06
SDAE [Bas15]	79.98	78.43
Semi-Supervised [Bas15]	97.95	93.92
Pretrained-AlexNet [Vak15]	99.46	99.57
AlexNet [Pap16]	99.98	99.93
AlexNet-small [Pap16]	99.86	99.90
VGG [Pap16]	99.98	99.98
"Modified Cifar-10 Full" CNN model (this Thesis)	99.09	98.8

4.4 Compressing the network: Word-length Optimization

Caffe trains the network using 32-bit floating point precision. However, fixed point arithmetic is less resource hungry than floating point arithmetic and better suited for most FPGA designs. Moreover, it has been shown that fixed point arithmetic is adequate for neural network computation [Gys16], [Cou14]. (figure 4.7).

In this thesis the "Ristretto"¹ tool was employed to estimate how different bit-width representations will affect the network's accuracy. Ristretto is an extension of Caffe that offers automated CNN-approximation to condense the 32-bit floating point networks. It allows the user to test, train and fine-tune networks with limited numerical precision.

Ristretto offers three approximations schemes: Dynamic Fixed Point, Minifloat and Power-of-two parameters. In this thesis, the Dynamic Fixed Point approximation was applied to the delivered design, while the accuracy of the Power-of-two parameters approximation was also calculated.

	Layer outputs	CONV parameters	FC parameters	32-bit floating point baseline	Fixed point accuracy
LeNet (Exp 1)	4-bit	4-bit	4-bit	99.1%	99.0% (98.7%)
LeNet (Exp 2)	4-bit	2-bit	2-bit	99.1%	98.8% (98.0%)
Full CIFAR-10	8-bit	8-bit	8-bit	81.7%	81.4% (80.6%)
SqueezeNet top-1	8-bit	8-bit	8-bit	57.7%	57.1% (55.2%)
CaffeNet top-1	8-bit	8-bit	8-bit	56.9%	56.0% (55.8%)
GoogLeNet top-1	8-bit	8-bit	8-bit	68.9%	66.6% (66.1%)

Figure 4.7: Fixed point arithmetic is adequate for neural network computation: Comparative presentation of the achieved accuracy between the full precision and a quantized version of various networks. These are fine-tuned networks with dynamic fixed point parameters & outputs. The numbers in parentheses indicate accuracy without fine-tuning. [Gys16]

Dynamic Fixed Point approximation

In the fixed point representation used, each number n is represented as follows:

$$n = (-1)^s \cdot 2^{-FL} \sum_{i=0}^{B-2} 2^i \cdot x_i \quad (4.1)$$

Here \mathbf{B} denotes the bit-width, \mathbf{s} the sign bit, \mathbf{FL} is the fractional length, and \mathbf{x} the mantissa bits. The "dynamic" refers to the fact that each different part of the network

¹ http://lepsucd.com/?page_id=621

is allowed to have a different fixed point representation, in order to better cover their dynamic range. Specifically, each layer of the network is split into two groups: one for the layer outputs and one for the layer weights. Within each group, all numbers are represented using the same integer and fractional length. Examples of this representation can be seen in figure 4.8.

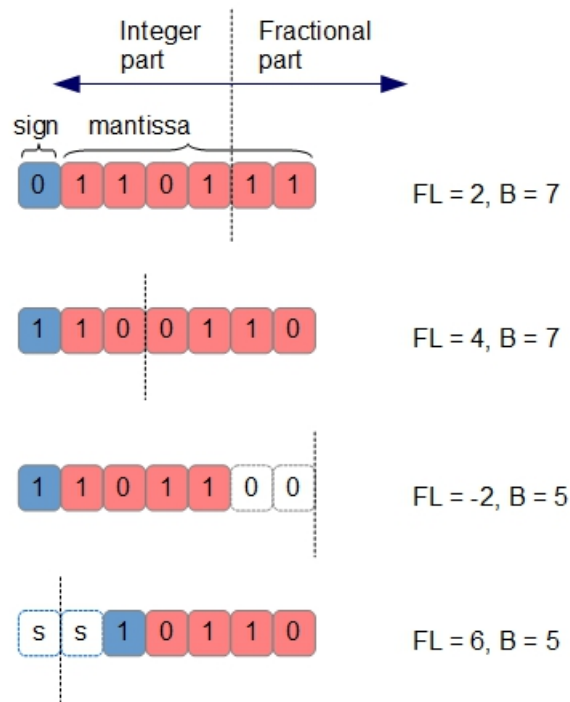


Figure 4.8: Examples of dynamic fixed point numbers. Note that the fractional length may be negative or greater than the word length.

Applying the Ristretto tool: Results

In table 4.4 the "Modified Cifar-10 Full" CNN model's accuracy analysis produced by the Ristretto is displayed. This analysis presents how the network's accuracy is affected when applying fixed point representation to individual network parts. During this analysis each one of the categories is quantized to dynamic fixed point, while the rest remain in full precision.

Table 4.4: "Modified Cifar-10 Full" CNN's accuracy analysis. Applying dynamic fixed point representation to one category at a time, while the rest remain in full 32-bit precision.

Baseline 32bit float Accuracy: 0.9874					
	16 bit	8 bit	4 bit	2 bit	1 bit
CONV Weights	0.9874	0.9873	0.8862	0.2399	-
FC Weights	0.9863	0.9863	0.9865	0.9737	0.0251
Activations	0.9656	0.9631	0.9316	0.3756	-

In table 4.5 the accuracy results of five different quantization scenarios are presented. In the power-of-two weights scheme, weights are represented as $n = (-1)^s \cdot 2^{\text{exp}}$, $\text{exp} \in \{-8, \dots, -1\}$. The power-of-two approximation comes in very handy for hardware implementations, since it exempt us from the need of using multipliers. Instead, the multiplications can be computed by bit-shifts, which are less resource & energy hungry than multipliers. Although this approximation is ideal for a hardware accelerator and the accuracy achieved for the specific dataset-network, without fine-tuning, is close to the baseline of using 32-bit floating numbers, we decided to not follow the power-of-two weights approximation scheme in our design. Instead, we followed the scenario C, in an attempt to study a more generic, and currently more frequent, version of the CNN-to-FPGA problem, while still using low-bit approximations.

It is often suggested that after reducing the precision of the network's activations and parameters, one should fine-tune the network to achieve even better performance. In our case however, re-training the network didn't have any effect on the achieved performance, despite experimenting with various magnitudes of learning rate.

Table 4.5: Different approximation scenarios applied on the "Modified Cifar-10 Full" network. Scenarios A-D use dynamic fixed point approximation. Scenario E forces weights to be power-of-two and applies dynamic fixed point to the activations.

Scenario \ Group	A	B	C	D	E
Conv Weights	8 bit	8 bit	8 bit	4 bit	Power-of-2
FC Weights	2 bit	2 bit	2 bit	2 bit	Power-of-2
Activations	32 bit	8 bit	4 bit	4 bit	8 bit
Accuracy %	0.9792	0.9783	0.9489	0.7588	0.9432

Table 4.6: Exact configuration of the chosen approximation. It corresponds to the scenario **C** in table 4.5, achieving 94.89% accuracy. **Bw** stands for "bit-width" and **FL** for "floating-length". Also note that the Bias follows the approximation of the Weights at each layer.

Scenario C	Conv1	Conv2	Conv3	FC
Bw Layer In	4	4	4	4
Bw Layer Out	4	4	4	4
Bw Weights	8	8	8	2
FL Layer In	-4	-3	-3	-3
FL Layer Out	-4	-5	-5	0
FL Weights	8	10	10	7

Extracting the Quantized Weights

Ristretto performs the quantization on-the-fly, without storing the quantized weights. The weights remain unmodified within the tool, in their initial 32-bit float representation. However we need to extract the quantized version of the weights, in their binary representation, in order to feed the RAMs of our design. One way to perform this is without changing Ristretto's source code: Through Caffe's python interface we access the network and read its weights. Then we quantize them, in the same way that Ristretto does ¹, and save them in external files, split as needed². The following code in Python is an example of a way to achieve this.

```

1 import numpy as np
2 import caffe
3 import bitstring
4
5 net = caffe.Net('SAT6.prototxt', 'latest_run.caffemodel', caffe.TEST)
6 Weights_Layer1 = net.params['conv1'][0].data
7 # The Filters are stored in ['conv1'][0], the Bias in ['conv1'][1]
8 max_data_weights_C1 = (2**(bit_width-1)-1)*(2**(-FL))
9 min_data_weights_C1 = -(2**(bit_width-1))*(2**(-FL))
10 # Weights_Layer1.shape={32,3,5,5}. This is a 4D array.
11 filters_num = Weights_Layer1.shape[0]
12 channels_num = Weights_Layer1.shape[1]
13
14 for filters in range(1, filters_num+1):
15     for channels in range(1, channels_num+1):

```

¹ Described at "Trim2FixedPoint_cpu" function, at http://github.com/pmgysel/caffe/blob/master/src/caffe/ristretto/layers/base_ristretto_layer.cpp

² In the next chapter, we describe that for in our FPGA design the weights must be organized in a very specific way (see section 5.4).


```

16     with open('conv1_quantized_weights%i_%i.txt'%(filters,channels), 'a') as
        outfile:
17         for x in np.nditer(Weights_Layer1[filters-1][channels-1],order='C',
op_flags=['readwrite']):
18             # For each one of the 5x5 weights.
19                 float_x = float(x)
20                 # Saturate data ("cut-off the msb")
21                 float_x = max(min(float_x,max_data_conv1), min_data_conv1)
22                 # Divide & Round Data ("cut-off the lsb")
23                 float_x /= 2**(-8)
24                 float_x = round(float_x)
25                 int_x = np.int8(float_x)
26                 integer_out=bitstring.BitArray(int=int_x, length=8)
27                 outfile.write('b"{}"\n'.format(integer_out.bin))

```

Listing 4.1: Example code of extracting the quantized weights

In figure 4.9 we can see how many of the weights, per layer, are equal to zero, one, or power-of-two, after the dynamic-fixed point quantization. It is possible that a hardware design, fined-tuned over these specific weights, could decrease the amount of multipliers needed.

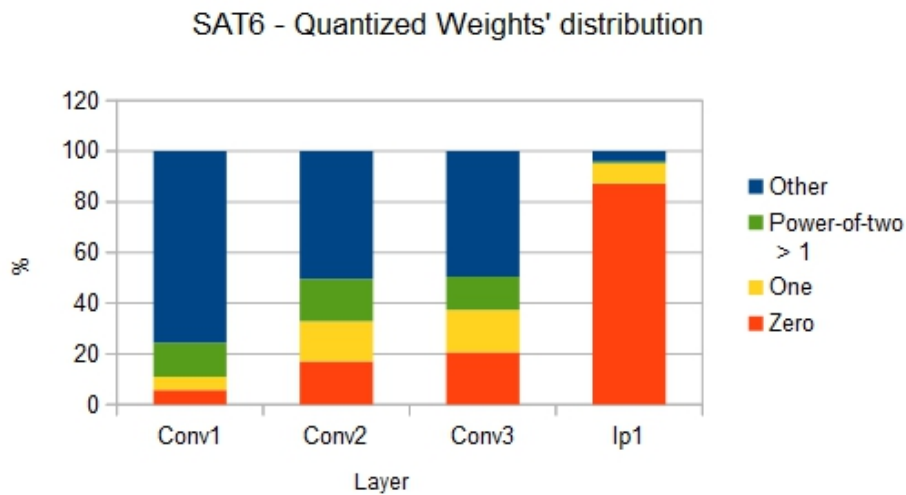


Figure 4.9: "Modified Cifar-10 Full" CNN model's Weights after quantization: Per layer percentages of how many weights are equal to zero, one, or a power-of-two number.

CHAPTER 5

FPGA Implementation

This chapter presents the architecture of the system that we designed, that implements the CNN topology described in the previous chapter. We also discuss some of the design decisions and options and we present metrics of the design's performance.

5.1 Design Approach

Convolutional Neural Networks are inherently massively parallel algorithms. For example, if we examine the first convolution layer of our topology (table 4.2), we can see that it consists of 1.88 million multiply operations. We shall observe that these multiplications are all independent of one another. Thus, a FPGA design that implements a convolution layer should utilize the potential of computing these multiplications in parallel, meaning the potential of loop unrolling the algorithm along the various dimensions. Our approach is pretty direct: Within a single layer create multiple instances of the basic components in order to reduce the need of multiplexing the components in time, effectively trading off resources for speed-up. Due to the data independence between the multiply operations within the same layer, the loop unrolling is limited only by the number of available hardware resources. However, since hardware resources are finite, design questions emerge regarding which loops and to what extent to unroll and how these configurations affect the other metrics of the system. Besides the loop-unrolling, there is also potential to pipeline parts of the design: From a higher level perspective, the different layers of the network can be pipelined, especially when computing a stream of images, while from a low-level design perspective, the RTL can be optimized to decrease the levels of logic and enable a higher clock frequency.

It is common in CNN-to-FPGA mappings to use the FPGA as an accelerator for the multiply-accumulate operations, while the whole process is controlled from a CPU and most of the data are stored on external to the FPGA memory [Che16b; Qiu16; Zha15]. However, in this thesis, the size of the specific network used enables us to perform all the

processing and the control on the FPGA chip, without the need for an external memory to store the intermediate results, or a CPU to coordinate and monitor the algorithm's execution. Throughput is therefore not limited by the off-chip memory bandwidth¹. Since the algorithm is solely executed on the FPGA chip, we decided to concurrently map all the layers of the network on the design. Each layer's architecture and control is dependent on the size of its inputs and outputs, which varies across the network. Moreover, due to the dynamic fixed point quantization scheme described in the previous chapter, these inputs and outputs use different fixed-point representation from layer to layer. Thus, the architecture that implements the 1st Convolution Layer wouldn't be possible to implement the 2nd Convolution Layer, etc without some reconfiguration of the design or complex control.

For each layer L of the CNN, we unrolled the computation in three dimensions: (i) The parallel processing of M input channels, (ii) the parallel generation of N output channels, which can also be seen as the parallel processing of N filters and (iii) the parallel computation of all the k^2 multiplications within each one of the $M \cdot N$ parallel filters' channels. This M, N configuration creates a grid of multiple instances of the design's basic components, some of which are of multiplicity 1 (one), some of M , some of N and some of $M \times N$. The memory organization of the activations (the layer's inputs & outputs) and the parameters (the layer's filters) is highly dependent on the choice of M, N .

In order to increase the portability and adjustability of the design, all the VHDL code in this thesis is completely parameterizable through a single configuration file, including the various bit-lengths, the size of the image, the multitude of the instantiated components and the hyper-parameters of the basic operations (e.g the kernel's size, the padding size & the stride of the convolution). The verification of the design was based on directed testing and on manual inspection of the simulation results in Xilinx's Vivado Suite, which were then compared to results from MATLAB.

¹ This claim is further justified in section 5.8

5.2 The design's key components

The design needs components that implement the following operations:

- Zero-pad the incoming data array at each convolution layer.
- Keep track and apply the different fixed point representations between the layers. Use full-precision accumulation inside each convolution layer before truncating.
- The 3D convolution operation.
- The max-pooling operation.
- The $\max(x,0)$ element-wise operation, as a rectifier linear unit.
- The inner product operation for the final layer of the network.
- Control and synchronize all the operations both intra- & inter- layer.

A generic version of one layer is illustrated in figure 5.1. This is a unified Layer, implementing the Convolution, the ReLU and the Max Pooling operations. This unified Layer makes up most of the design. Three of these unified Layers are stacked up one after the other to complete the convolution phase of the network, followed by a fully connected layer. We shall now describe its components.

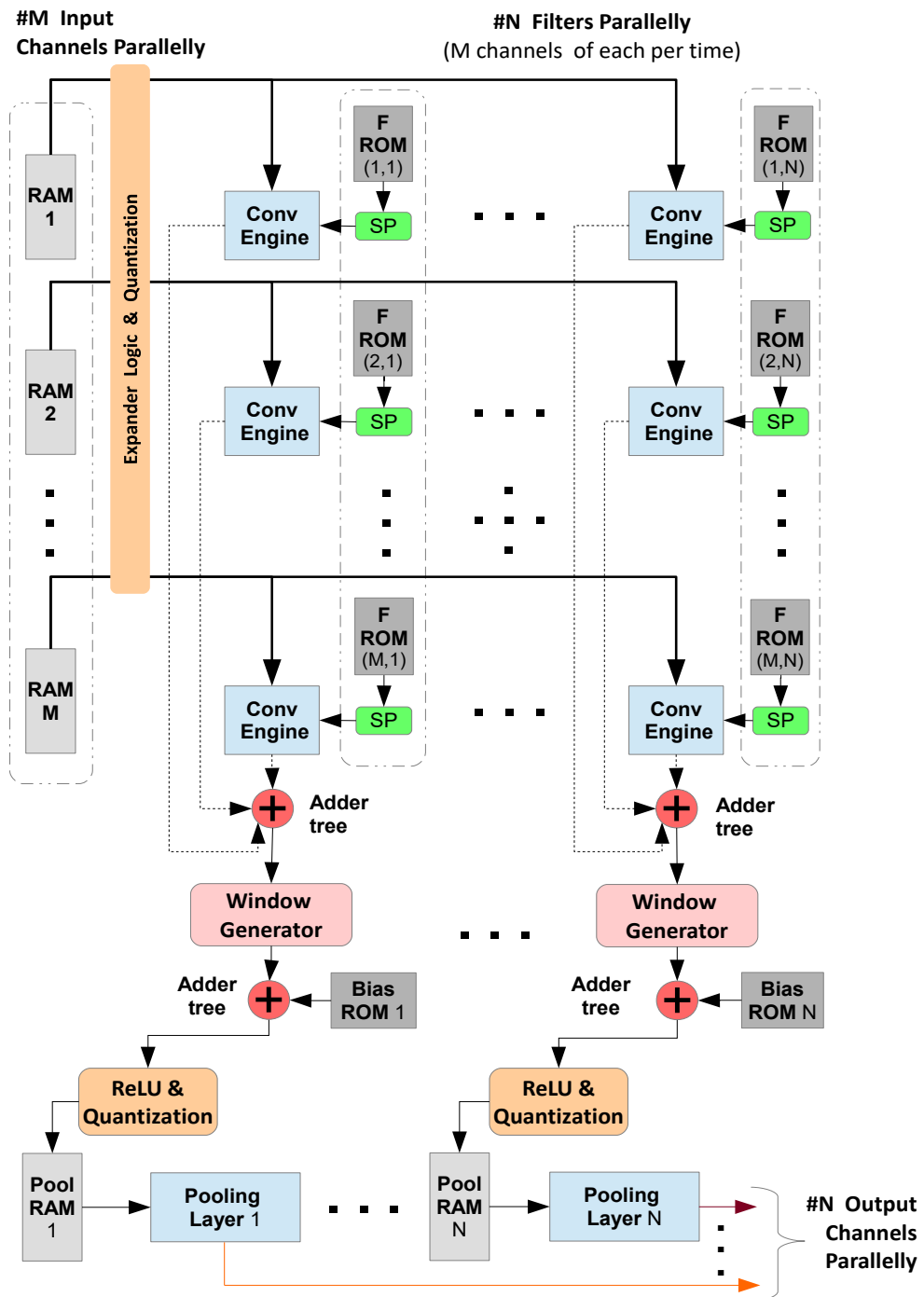


Figure 5.1: A generic version of the unified Layer's structure. Performs the zero-padding of its input, the truncation of its inputs & outputs, the convolution, the ReLU and the max pooling operations.

5.2.1 The "Expander" component

Multiplicity per Layer : 1

The Expander component is responsible for symmetrically zero-padding the 2D input matrices at each layer. The 3D operation is realized by repeating the operation for every channel of the input. We suppose that every channel is stored in row-major order and that each pixel is stored at each own address. The Expander keeps track of which pixel is being processed and sets the RAM address accordingly. When reaching the edges of each input channel, it points to a specific address, where the default value "0" is stored. A "padding" variable in the configuration file determines by how many zero-pad values will the input array be expanded.

5.2.2 The "2D Convolution Engine" component

Multiplicity per Layer : $M \times N$

At the core of the convolution layers lies a component that performs a 2D cross-correlation operation, between a filter and an input 2D array. By storing the filter's weights pre-flipped in the memory, or accessing them in reverse row-major order, the 2D convolution operation is performed.¹ The 3D operation is realized by repeating the 2D operation for every input channel and aggregating the intermediate results.

Our component is based on the work described at [Sho93]. Its structure can be seen in figure 5.2. We suppose that both the filter and the input array are square. The input array is of size $(Image_{width} + 2 \cdot Padding)^2$. At the input of the component arrives one pixel per cycle, in raster-scan order. Each pixel is simultaneously multiplied by all the 5x5 weights (the filter's coefficients) and then partial sums are accumulated by the adder chain. The delays between the adders allow the proper alignment of the kernel to the input array. This architecture creates bursts of "E" correct results, one per cycle, interleaved by "X" wrong results. "E" is the number of times that the kernel fits inside the given input, in one dimension, calculated as

$$E = \frac{Image_{width} + 2 \cdot Padding - Kernel_{width} + Stride}{Stride}$$

"X" is the number of times that the kernel won't be correctly aligned to the input array, calculated as $X = Kernel_{width} - 1$. This situation arises when we reach the edge of the input array and the kernel will have to move to the next line.

The first valid result will be ready after $b = (Kernel_{width} - 1) \cdot (Image_{width} + 2 \cdot Padding) + Kernel_{width}$ pixels have been streamed into the Convolution Engine. One

¹ This is also the way that the Caffe framework performs the convolution: by storing the weights pre-flipped.

pass of the whole 2D-input array will be completed after $E^2 + E * X + b$ cycles, where E^2 are the times that the kernel fits inside the input array and $E * X$ are the total wrong results between the bursts of valid ones.¹

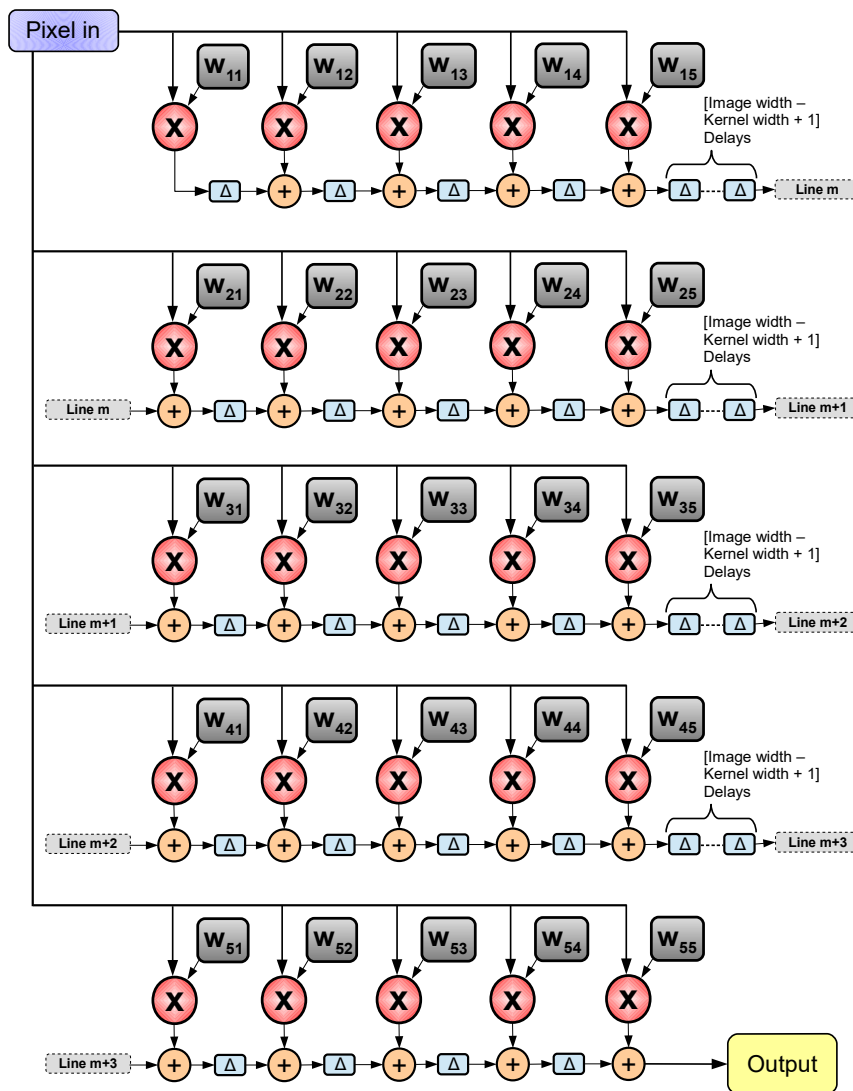


Figure 5.2: The 2D Convolution Engine. Illustrated for a 5x5 Kernel.

¹ Actually, this could be $E^2 + (E - 1) * X + b$, since we don't have to wait for the last "X" cycles.

5.2.3 The "SP" component

Multiplicity per Layer : $M \times N$

Each one of the Convolution Engines gets loaded with filters from its own ROM. The SP component is a Serial-to-Parallel converter, responsible for reading the filters from the corresponding ROM and feed them to the Convolution Engine. Each filter consists of 25 values of weights (5x5), with each weight stored at each own memory address. The Convolution Engine component needs all of these 25 values simultaneously, thus the SP component shifts these values on registers for immediate, parallel access. The control that is responsible for the SP component makes sure that the next set of weights is always pre-loaded on registers, so that the Convolution Engines can run without being paused.¹

5.2.4 The "Window-Gen" component

Multiplicity per Layer : N

The Window-Gen component is a sliding window generator and it is a component that wasn't created during this thesis. We borrowed it from the work described at [Sti16] and its source code can be found [here](#) ².

Let $M_{parallel}$ be the number of input channels processed in parallel and M_{total} the total number of input channels at a given convolution layer. When $M_{parallel} < M_{total}$, we process a 3D partition of the input plane. Thus, to complete the 3D convolution operation, we need to aggregate these partial 3D-convolution results.³

The Window-Gen component stores those intermediate, partial 3D-convolution results, aligns them, and outputs them in a window of parallel registers. We use the Window-Gen module to feed these partial 3D-convolution results to an adder tree and ensure that the n_{th} element produced by processing the $M_{parallel}^i$ channels, $i = 1 \dots \frac{M_{total}}{M_{parallel}}$, $n = 1 \dots (Output_{width})^2$, will eventually be summed with the n_{th} elements produced by the $M_{parallel}^j$ channels, $j = 1 \dots \frac{M_{total}}{M_{parallel}}$ and $j \neq i$.⁴ The process is illustrated in figure 5.3.

The Window-Gen can handle the interrupting way that the 2D Convolution Engines produce valid results, with a corresponding "input_valid" signal, which enables us to

1 Otherwise, a pause of $k \cdot k = 25$ cycles would be needed to shift each new filter from the ROM to the registers.

2 https://github.com/ARC-Lab-UF/window_gen

3 Reminder: A 3D convolution of a 3D input with one 3D filter, results in a 2D output. M_{total} 2D convolution operations must be performed and their results at each position have to be aggregated. M_{total} is the depth of the 3D input.

4 i & j are indices.

pipeline the creation of the windows to the Convolution Engines, reading the results at the rate at which they are created. To achieve this, the Window-Gen module uses complex control between its internal RAMs.

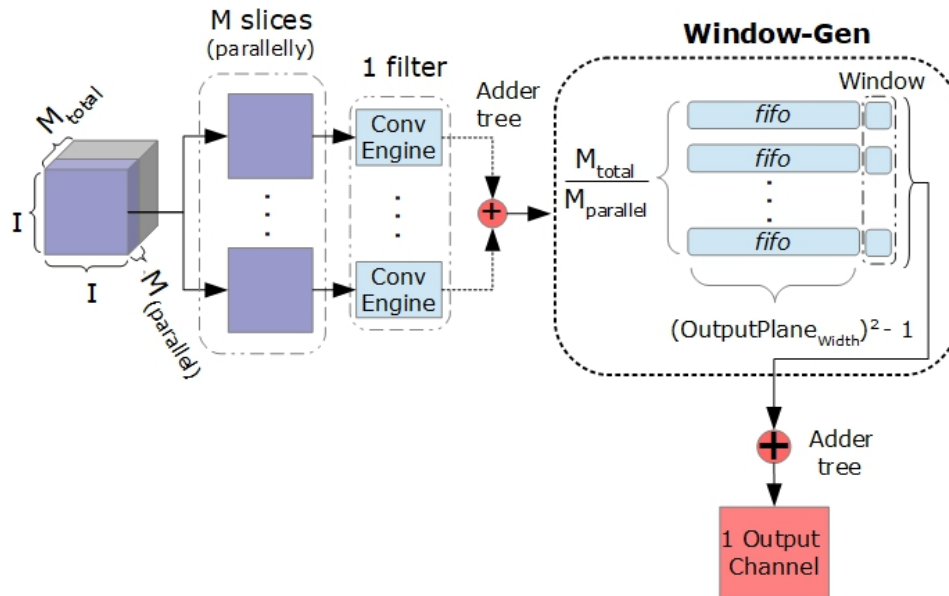


Figure 5.3: The Window-Gen component. Eventually each fifo i will contain the partial results generated by processing the $M_{parallel}^i$ parallel channels, $i = 1 \dots \frac{M_{total}}{M_{parallel}}$.

5.2.5 The "Pooling Layer" component

Multiplicity per Layer : N

The Pooling Layer consists of several sub-components itself. It reads a single channel of the 3D-convolution's output, one pixel per cycle, and creates a window that "slides" over the whole 2D input array. The window is a set of registers, that at each time contain a sub-matrix of the given 2D input array. The elements of the window are parallelly input into a comparator-tree. This structure resembles an adder-tree, but

performs a comparison operation between its nodes instead. The Pooling Layer outputs one pixel per cycle, the one that has the greatest value at each position of the window. Its operation is pipelined to the operation of the Convolution Engines.

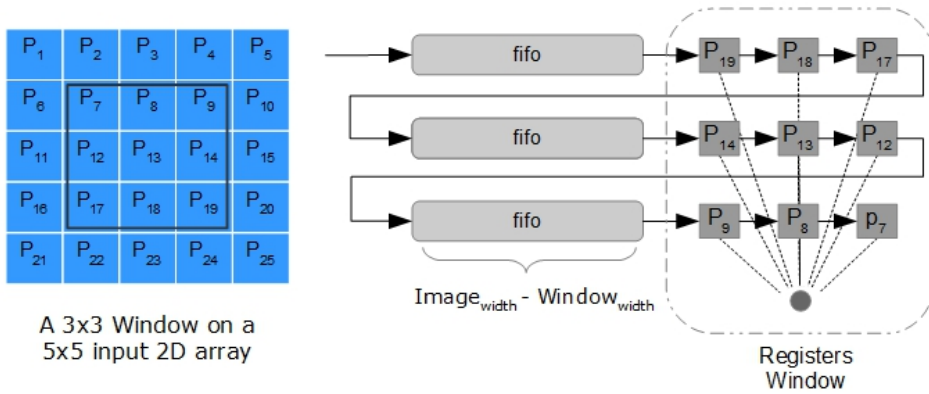


Figure 5.4: Pooling Window

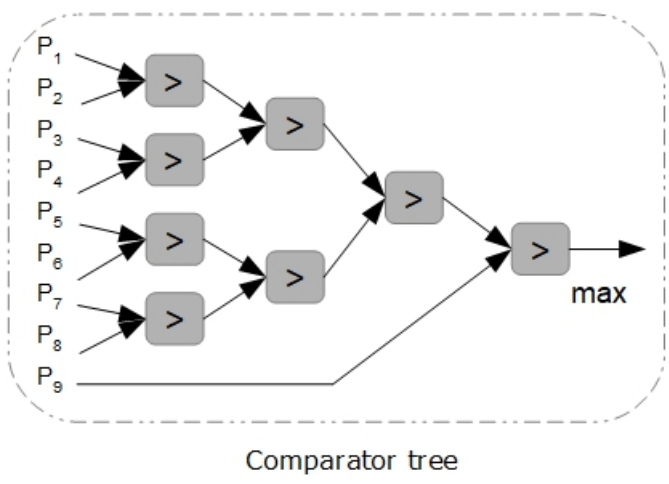


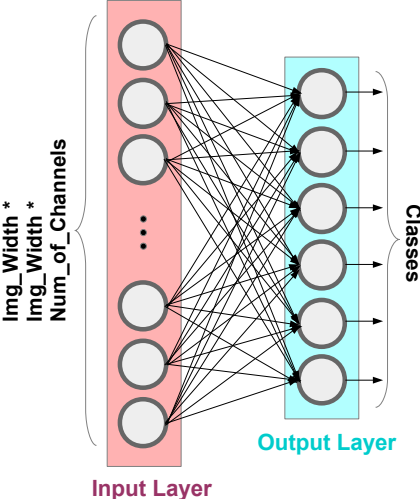
Figure 5.5: Comparator Tree

5.2.6 The "Fully Connected Layer" component

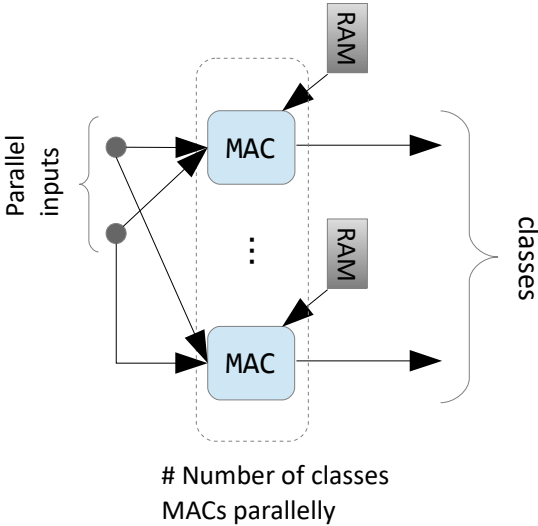
The Fully Connected layer is the final stage of the network. Its topology is the same as seen in classical neural networks. Since there is only one layer of fully connected neurons and no hidden layers, this operation is actually a multiplication of a matrix with a vector. The matrix is the weights of the layer with size (Number of classes) x (Input $Image_{width}^2 \cdot$ Total number of input channels) and the vector is the input of size (Input $Image_{width}^2 \cdot$ Total number of input channels) x 1. The output is a vector of size (Number of classes) x 1 (figure 5.6(a)), containing the scores for each class.

In the above description we have assumed a point in time where all the inputs are simultaneously available. However, the inputs are generated from the previous layer, $N_{par}^{L=3}$ of them per cycle. Thus all the inputs will be available (FC's Input $Image_{width}^2 \cdot \frac{\text{Total number of input channels}}{\text{Number of parallel input channels}} - 1$) cycles after the 1st input is generated. We can observe that we can pipeline the computation and use the inputs at the rate at which they are generated, without waiting for the full input vector to be formed. This can be achieved by using Multiply-Accumulate components, one per output, meaning one per classification class. Each of these components will take at its input the $N_{par}^{L=3}$ parallelly available values and gradually perform the full multiplication. The final, per class, results will be available shortly after the last inputs will become available. The component's diagram can be seen in figure 5.6(b).

In the general case, a serial-to-parallel converter would also be needed here, in order to simultaneously provide the MAC components with as many weights, as the number of their inputs. However, due to the quantization scheme, the weights in the Fully Connected Layer are pretty small -just 2 bits wide- and thus we decided to pack M_{par}^{FC} of them at the same RAM address, where $M_{par}^{FC} = N_{par}^{L=3}$ is the number of parallel inputs at the FC layer.



(a) The topology of the Fully Connected layer.



(b) The "Fully Connected Layer" component.

5.3 Bit-width Calculations

In this section we briefly discuss the bit-widths of some of the signals involved in a the convolution operation of a single layer. We assume that an image is square, of size $I \times I$, each element of each is BW_{in} bits, with the sign bit included. Every kernel is of size $k \times k$ and its elements are of BW_{param} bits size each, with the sign bit included. For each Convolution Engine we perform $k \times k$ multiplications and we sum the results. We want to calculate these results in full precision, so in order to prevent them from overflowing, each `Convolve_Engine_Result` is a signal $BW_{in} + BW_{param} + \lceil \log_2(k \cdot k) \rceil$ bits wide. Moreover, in order to produce one output channel, we sum the results of multiple Convolution Engines and also add the respective bias value. If there are M_{total} input channels to be processed, then the signal that carries the addition of all these values should be $BW_{in} + BW_{param} + \lceil \log_2(k \cdot k) \rceil + \lceil \log_2(M_{total}) \rceil + 1$ bits wide. Before adding the bias value, the result from the Convolution Engines would have FL_{Conv} bits fractional length, $FL_{Conv} = FL_{in} + FL_{param}$. However, special care must be taken when adding numbers of different fractional lengths, in order for the bits of the two operands to be aligned. The bias is of size BW_{param} bits and has FL_{param} bits fractional length. The signal with the smaller fractional length is shifted left in such a way that the two numbers will be aligned and their addition can be performed. In the current configuration for the bit-widths, as produced by the Ristretto tool (see section 4.4), the result of the Convolution Engines has always smaller floating length than the floating length of the bias.

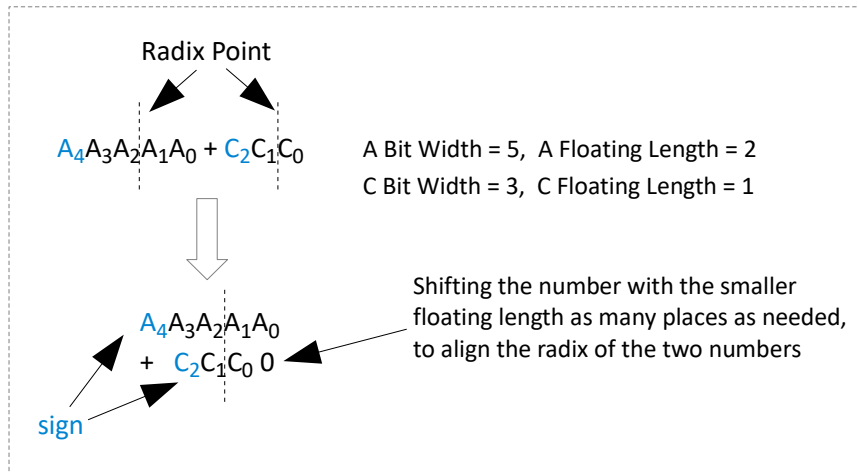


Figure 5.7: Simplified illustration of adding two numbers with different floating lengths.

Truncation: The signal that carries the sum of the results of the Convolution Engines and the bias is BW_{SConv} bits wide and has a fractional length $FL_{SConv} = FL_{param}$. We want to truncate this signal and the result should be $Desired_out_BW$ bits wide with $Desired_out_FL$ floating length, to match the results of the Ristretto tools (section

4.4).

Depth of RAMs

- The RAMs where the input feature maps of the convolution layer are stored, are of depth $I \cdot I \cdot \lceil \frac{M_{total}}{M_{parallel}} \rceil$ each¹, where I is the width of the input feature map, M_{total} the total number of input feature maps (input channels) and $M_{parallel}$ the number of input features maps that are parallelly input in the layer.
- The ROMs where the weights of the convolution kernels are stored, are of depth $\lceil \frac{N_{total}}{N_{parallel}} \rceil \cdot \lceil \frac{M_{total}}{M_{parallel}} \rceil \cdot k \cdot k$ each. N_{total} are the total number of output feature maps (channels) that the layer has to generate, $N_{parallel}$ is the number of output channels that the layer generates simultaneously and $k \times k$ are the dimensions of a single kernel. A single convolution layer has a total of $M_{total} \cdot N_{total} \cdot k \cdot k$ weights which are distributed between these ROMs.
- The ROMs where the bias of each filter is stored are of depth $\lceil \frac{N_{total}}{N_{parallel}} \rceil$ each.

In order to address each one of these RAMs, the respective address signal needs to be $\lceil \log_2(RAM_Depth) \rceil$ bits wide.

5.4 On-Chip Memory Organization

The organization of the memory for a specific layer is dependent on the layer's number of parallel input & output channels. For M_{par} parallel inputs and N_{par} parallel outputs, we have: M_{par} input RAMs, N_{par} Pooling RAMs, $M_{par} \times N_{par}$ Filter ROMs and N_{par} Bias ROMs (see figure 5.1).

For $i = 1 \dots M_{par}$ and $j = 1 \dots N_{par}$ and M_{total}, N_{total} the total number of input & output channels respectively, to be processed, we will have ²:

- the Filter ROM_{ij} will contain the filters in following order:

$$ROM_{ij} \leq \text{for } k=1 \text{ to } \frac{N_{total}}{N_{par}}$$

$$\text{for } v=1 \text{ to } \frac{M_{total}}{M_{par}}$$

$$Filter[j + (k - 1) \cdot N_{par}] : Channel[i + (v - 1) \cdot M_{par}]$$

$$\text{end}$$

$$\text{end}$$
- the Bias ROM_j 's contents should have the following order:

$$ROM_j \leq \text{for } k=1 \text{ to } \frac{N_{total}}{N_{par}}$$

$$Filter[j + (k - 1) \cdot N_{par}]$$

$$\text{end}$$

¹ $\text{ceil}(A) = \lceil A \rceil$

² M_{par} and N_{par} should be divisors of M_{total}, N_{total} respectively. For example, if we have $M_{total} = 32$ and we want $M_{par} = 7$, we shall increase the $M_{total} = 35$ with dummy RAMs/ROMs. More on this matter in the next section.

- the Pooling RAM_j 's contents should have the following order:
 $ROM_j \leq$ for $k=1$ to $\frac{N_{total}}{N_{par}}$
 $Channel[j + (k - 1) \cdot N_{par}]$
end
- the input RAM_i 's contents should have the following order:
 $RAM_i \leq$ for $v=1$ to $\frac{M_{total}}{M_{par}}$
 $Channel[i + (v - 1) \cdot M_{par}]$
end

An example is illustrated in table 5.1.

Table 5.1: Memory Organization of 32 filters, with 3 channels each, for a 3x2 grid of Convolution Engines ($M_{par} \times N_{par}$).

ROM (1,1)	ROM (1,2)
Filter 1, Channel 1	Filter 2, Channel 1
Filter 3, Channel 1	Filter 4, Channel 1
...	...
Filter 31, Channel 1	Filter 32, Channel 1
ROM (2,1)	ROM (2,2)
Filter 1, Channel 2	Filter 2, Channel 2
Filter 3, Channel 2	Filter 4, Channel 2
...	...
Filter 31, Channel 2	Filter 32, Channel 2
ROM (3,1)	ROM (3,2)
Filter 1, Channel 3	Filter 2, Channel 3
Filter 3, Channel 3	Filter 4, Channel 3
...	...
Filter 31, Channel 3	Filter 32, Channel 3

5.5 Intra-Layer Control

One thing that became apparent during this thesis is that, when designing a system in VHDL, more effort is needed in order to control and synchronize the various components, than to design the components themselves. We shall now briefly describe how the components within the layers are being synchronized. We will examine the unified Layer, which performs the zero-pad expanding of its input, the truncating of the inputs & outputs, the convolution operation, the ReLU and the max pooling operations (illustrated in figure 5.1). The Fully Connected layer does not have the same complexity and is sufficiently described in the according section.

The control is based upon signals between the components -thus internal to the Layer's architecture-, or signals that are input in the Layer -external to the Layer's architecture. These signals are altering the state of the components. That can be an on-off switch, a reset of the component to its initial state, ordering a transition to another state in their FSM, or a simpler straight-forward boolean operation. All the circuits in this thesis are designed to be synchronous to the clocks. Meticulous attention must be given for all the operations in the design to be exactly and correctly synchronized. A common practice is to observe the operation of a block and manually insert the appropriate amount of registers to delay several of the control signals included. Delaying the data is generally discouraged, because that would translate to more resources of the design, whereas the control signals are usually one bit each.

Suppose that we start the function of a Layer having all the data inputs ready and stored in the respective RAMs and ROMs and that the design is in its initial state. That can be due to a reset, external to the Layer's architecture, the FPGA's configuration ¹, or the Layer has returned to its initial state after some computations. At that point in time, the system is in an "IDLE" state. However, a pulse-signal has already been issued to the Serial-to-Parallel components and their associated ROMs to start working, in order to have the first set of weights ready for computation. After ~ 25 cycles ² the Layer will be ready to start reading the data from its input RAMs, will inform its surrounding environment about this with an "Initialization_ready" signal and the SP components will be paused. When an external "enable" signal arrives, the Layer will issue the Expander component to start working. The Expander will control which address of the input RAMs is read. Since all the Convolution Engines work simultaneously, only one address is needed for all the input RAMs, where the input channels are stored, and thus only one Expander component is needed per Layer. The Expander, in its turn, will inform with a pulse-signal when the first "pixel" of the input RAMs is available and the main, convolution state machine will transit to the "CONVOLUTION RUNNING" state.

1 configuration has the same effect as global reset: Xilinx's white paper "Get Smart About Reset: Think Local, Not Global", 2008

2 that is $kernel_{width}^2$ cycles, since each weight is stored at each own address

At that point in time, the SP components will kick in again. The weights needed for the convolution operation that has just started, are already in place and will remain static while the first set of 2D input planes is being read. The SP components will also get the next set of weights ready, so that they will be immediately available for the next set of 2D input planes. Also, a component that keeps track of the Convolution Engines' operation is being started. This component, the "Shoup Controller" is responsible for monitoring all the 2D convolution operations within the Layer and create a "valid_out" signal to accompany the Convolution Engine components' results. As stated in subsection 5.2.2, these Engines produce results in an interleaved manner, where bursts of valid results are followed by bursts of invalid results, so a "valid_out" signal is imperative here. The Shoup Controller also informs the main convolution state machine when a set of 2D convolutions has finished. Then the convolution operation will continue with the next set of weights and inputs, until we reach the ending address of the weights' ROMs. That signals that the last 2D convolution is about to be performed and thus that all the filters will have been convolved with the input plane and sets the main, convolution state machine to the "LAST CONVOLUTION" state. Mind that we check when we will reach the ending address of the weights' ROMs and not the ending address of the input RAMs. The input RAMs will be re-read multiple times in order to convolve the input channels with all the filters¹. After the last convolution is calculated, the main convolution state machine will transit to an "IMAGE DONE" state and wait till a new image is input, to start all over again.

The Convolution Engines produce one output per cycle and the proper addition of the valid outputs is taken care with the use of the "Window-Gen" components, paired with adder trees. When all the channels of a filter are processed, the respective Bias value is also added to the final results, which are then truncated. In the case that the output pixel is negative, it is set to zero, applying the non-linearity of the ReLU. These, final outputs, are the results of the 3D convolution operation, which consist the output plane before the pooling operation. Still accompanied by, an appropriately delayed, "valid_out" signal, these results of the 3D convolution operation are stored in dual-port RAMs, denoted as "Pool RAM" in figure 5.1. To perform the pooling operation we need "windows" of values, as already described in section 5.2.5. However, this time we do not use the Window-Gen component. Instead, we use a different and less resource-hungry approach. The Pooling Layer component will internally create the windows needed, in a similar manner to the Window-Gen component. However, it lacks the complex control to handle the interrupting rate in which the 3D convolution results are created. Since we do not want to wait for the entire plane of the convolved values to be generated and then start the max-pooling operation, nor can we directly connect the Pooling Layer to the preceding adder tree, we decided to store the valid convolution results in the

¹ That means that the contents of these RAMs must not be changed, until the whole 3D convolution operation of this Layer is completed. More about this in the next section.

intermediate "Pooling RAMs". The Pooling Layer will start reading these RAMs, after a certain delay in time. The delay before we start reading the RAMs' contents makes sure that we won't read an address that has not yet been written ¹. That's why these RAMs need to be dual-port, so that they can be both read by the Pooling Layer and written by the convolution operation at different addresses, at the same time. The Pooling Layer will complete its operation a few cycles after the last valid convolution result will have been generated and made available at its input. Its operation is pipelined to the operation of the Convolution Engines. The Pooling Layer resembles a bit the convolution layer, in the sense that it makes use of a "moving window" on top of its input plane, performing some operations on the data within the window. That implies that the Pooling layer component itself, contains several FSMs, counters and control signals. However, we will not describe it in any more detail.

¹ We give the writing process a head start of about $(k - 1) \cdot Img$ cycles before we start reading the results. K and Img refer to the convolution operation's parameters of $kernel_{width}$ and $Input_{width}$.

5.6 Inter-Layer Control

"Inter-Layer" control refers to how are the various Layers connected to one another, thus how and when will the data generated from one Layer be input to the next Layer.

In order to address this question, we shall first make some observations. Every unified Layer has M_{total} input channels to process and N_{total} output channels to generate for every image. Each of these Layers is designed to operate simultaneously on $M_{parallel}$ and $N_{parallel}$ of them respectively. In order to produce each set of the $N_{parallel}^L$ output channels, the layer L must combine the intermediate results produced by all the M_{total}^L channels at its input. Since the Layer L process $M_{parallel}^L$ input channels simultaneously, the next Layer $L + 1$ will have valid results at its input and can start its operation, only when the last input channels of Layer L will be under processing, for that particular set of $N_{parallel}^L$ output channels. When $N_{parallel}^{L+1} < N_{total}^{L+1}$, meaning, when the Layer $L + 1$ performs the convolution operation with a subset of the total number of its filters at a time, the Layer $L + 1$ will need to re-read the data at its input $\frac{N_{total}^{L+1}}{N_{parallel}^{L+1}}$ times, in order to produce all the output channels. That means, that until the Layer $L + 1$ has finished its operation, Layer L must not modify these data. Thus, when processing a single image, there aren't many opportunities to pipeline the various layers of the network, with the sole exception of the Fully Connected layer, which can be fully pipelined (section 5.2.6). However, when processing a stream of images, things can be more in our favor.

When processing a stream of images, we can pipeline the computation of different images. For that reason we double the buffers in between the Layers and use them in a ping-pong manner. Then, while the Layer $L + 1$ is operating, the Layer L no longer needs to be completely idle. It can operate on a new image and store its output in a RAM not currently being read. In an ideal scenario, the Layers would be fully pipelined and none of them would need to be idle at all. This is a property that depends on the latency of each Layer L and whether it can store its output without corrupting the data that the next Layer reads. In practice, the Layers cannot be fully pipelined and their operation will need to be stalled at times. A simplified example of this ping-pong style of buffering is illustrated in figure 5.8.

The network's specification for the Layers' output is that $N_{total}^{L=1} = 32$, $N_{total}^{L=2} = 32$ and $N_{total}^{L=3} = 64$. Deciding the exact configuration of $M_{parallel} \times N_{parallel}$ for each Layer is a matter that will be explored in the next section "Design Space Exploration" (section 5.7). Here we will examine how consecutive Layers should be interconnected, for a wide range of possible configurations. The easiest approach from the point of design is to force $N_{parallel}^L = M_{parallel}^{L+1}$, meaning that the $L + 1$ Layer has as many input channels in parallel as the previous Layer L generates. This makes the wiring pretty easy and reduces the amount of control needed between the Layers. Of course, the need for multiplexing the dual buffers to implement the Ping-Pong buffering remains. However, this is a restricting design approach which, as we will see in the next section 5.7, does not lead

to the best results in terms of performance. Therefore it is a choice that presents itself as a design time vs performance trade-off. If instead we let $N_{parallel}^L \neq M_{parallel}^{L+1}$, then the number of RAMs between the two Layers shall be $2 \cdot \max(N_{parallel}^L, M_{parallel}^{L+1})$, where the "times two" term is due to the ping-pong, double buffering. If $N_{parallel}^L < M_{parallel}^{L+1}$, then the L Layer must multiplex its writing, alternating between the $M_{parallel}^{L+1}$ RAMs. If $M_{parallel}^{L+1} < N_{parallel}^L$, then the $L + 1$ Layer must multiplex its reading, alternating between the $N_{parallel}^L$ RAMs. An example is illustrated in figure 5.8. In this example, $N_{parallel}^L = 1$, $M_{parallel}^{L+1} = 2$ and $N_{total}^L = M_{total}^{L+1} = 8$.

Although we let $N_{parallel}^L \neq M_{parallel}^{L+1}$, we do demand that the $M_{parallel}$ and $N_{parallel}$ are divisors of N_{total} and M_{total} respectively. The profound range of values then is $M_{parallel}^{L=1} \in \{1,3\}$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in \{1, 2, 4, 8, 16, 32\}$. However, we may also select values that are not included in the above sets, but are within $[1,32]$ interval instead. In that case, we increase the number of the Layer's total channels adequately, with "dummy" channels, where all values will be equal to zero, in order to satisfy the demand for the number of parallel channels to be divisors of the total number of channels. The extra, "dummy" channels will not affect the numerical results of the Layer. In these configurations the amount of RAMs needed is increased, but by allowing configurations like these to be inside our range of exploration, we can better control the latency of each Layer and possibly gain in performance¹. However, these choices significantly increase the design's complexity. An example is illustrated in figure 5.9. In this example we suppose that we have initial $N_{total}^L = M_{total}^{L+1} = 5$. We configure the Layer L to produce three output channels parallelly and the Layer $L + 1$ to read four input channels parallelly. This configuration will result in a need for some dummy channels and filters.

¹ For example consider a scenario where we have to process 32 channels in the input of a Layer. If we are allowed to only choose from divisors of 32, we can process either 1,2,4,8,16, or 32 of them parallelly. If we can't fit in our design the processing of 16 of them parallelly, then the next best choice is 8 channels parallelly, resulting in a $32/8 = 4$ factor of how many iterations will be needed to process all the channels. If instead we allow configurations of $M, N \in [1,32]$, we may choose $M=11$ and reduce the number of iterations needed to $32/11 = 3$.

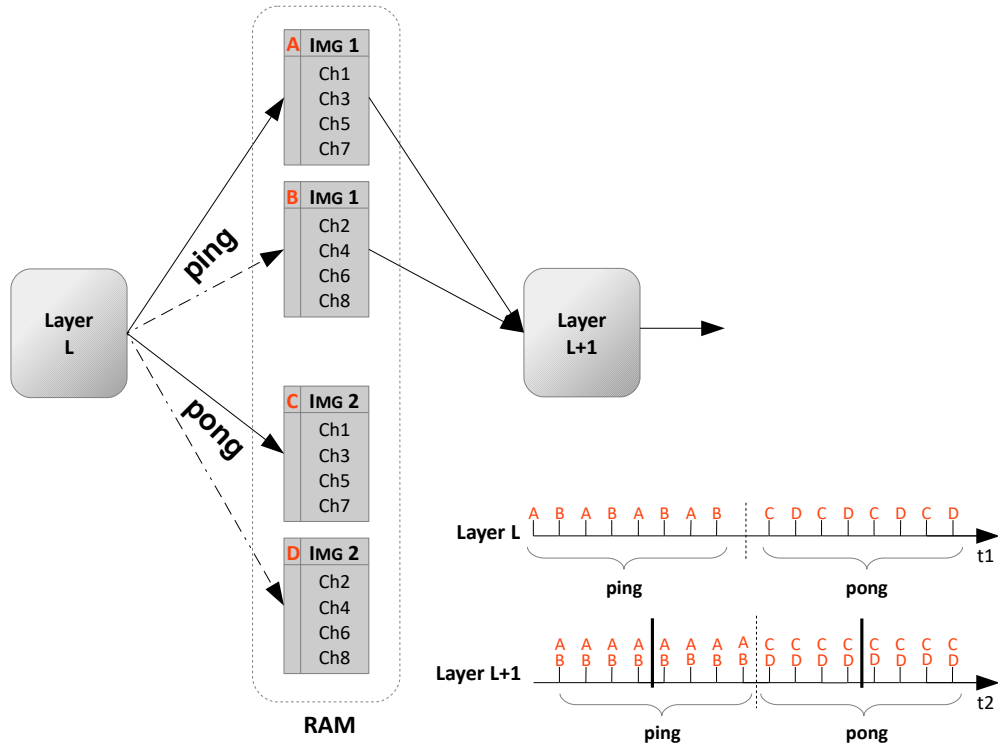


Figure 5.8: Example of connecting two Layers. $N_{parallel}^L = 1$ and $M_{parallel}^{L+1} = 2$. $N_{total}^L = M_{total}^{L+1} = 8$. Also $N_{parallel}^{L+1} = 1$ and $N_{total}^{L+1} = 2$. Layer L creates one output channel each time, while Layer $L + 1$ reads two channels at a time. For that reason Layer L alternates its writing between RAMs A&B. Because the $L + 1$ Layer has to create two output channels, but only generates one of them at a time, it will re-read the input data twice, one for each filter. Moreover, double buffers in Ping-Pong style, to pipeline the processing of a stream of images: At one point in time Layer L created the output of Img1 (ping) and then continues with generating the output of Img2 (pong). Layer $L + 1$ reads the output of Img1, while the output of Img2 is generated. When Layer $L + 1$ proceeds to reading the output of Img2, Layer L will continue in generating the output of Img3, etc. Note that the two time axis use a different scale and their relative position is unrelated. Also stalling is not illustrated.

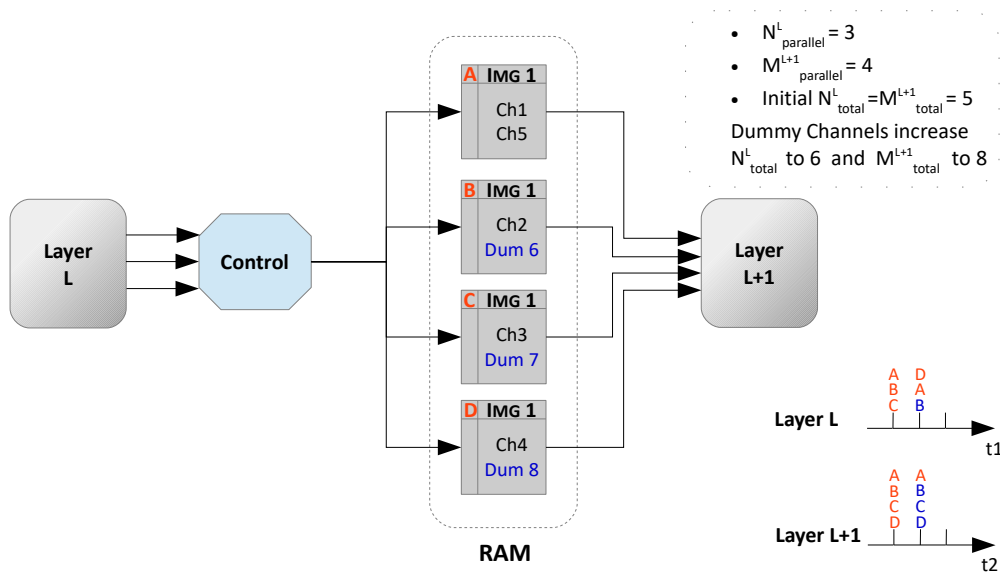


Figure 5.9: Layer L creates three output channels parallelly, while Layer $L + 1$ reads four input channels parallelly. The total amount of channels in between initially was equal to five. In order to preserve the symmetry, dummy channels are used, which increase the amount of channels in between to $N_{total}^L = ceil(\frac{5}{3}) \cdot 3 = 6$ and $M_{total}^{L+1} = ceil(\frac{5}{4}) \cdot 4 = 8$. Mind that the two time axis use a different scale and their relative position is unrelated.

5.7 Design Space Exploration

One question that emerges is how to decide the number of parallel input & output channels for each layer. When imposing no constraints, the design has a vast space of possible configurations = $\prod_{i=1}^3 (Layer_{in}^i \cdot Layer_{out}^i)$, which is $> 2^{27}$.¹ Different configurations of the design result in widely different performance of the system.

5.7.1 Approach

We can measure the performance of the design using multiple metrics:

Device Resource Utilization is a percentage indicating how many of the device's available resources we are using.

Latency is the amount of time needed for the system to process a single image input. For a pipelined system, this time interval is different from the time needed to process successive inputs.

Throughput is the rate at which the system can classify new images.

Hardware Time Utilization is a percentage indicating how effectively we are using the device resources that we have occupied, during the execution time, i.e whether and how much of our resources are continuously busy. This is a metric closely related with pipelining. A fully pipelined system would have Hardware Time Utilization = 100%.

Power Consumption is the amount of energy in time that the design needs to operate.

We decided to explore the design space and optimize the design based on two axes: **(i)** maximize the system's throughput and **(ii)** take into account the hardware time utilization. In this thesis we did not have specific constraints to satisfy in our design, i.e for the power consumption, etc. However, we perceive throughput as the most important metric of our design, since we expect that the FPGA will have a continuous stream of images at its input to classify and we want to perform the classification as fast as possible, delivering a design that can be meaningfully and successfully employed in real-time satellite image classification. Latency on the other hand is considered as less important for this application. Another important metric is the power consumption, however it is very computationally exhaustive to calculate the power consumption for every configuration. This would require the execution of both synthesis and implementation for each configuration, which are computationally heavy processes. Moreover, we rely on the fact that FPGAs typically have smaller energy consumption compared to their competitors. A comparison regarding the power consumption of different configurations can be estimated based on the respective hardware time utilization of these configurations.

¹ that is: $3 \cdot 32 \cdot 32 \cdot 32 \cdot 32 \cdot 64 = 201.326.592$ possible configurations.

We can narrow down the design space a lot, based on **(i)** the maximum number of Convolution Engines that can fit on the device in the best scenario, given the available resources and **(ii)** various constraints on the input & output of the Layers. We apply constraints on the I&O of the Layers based on the fact that for a constant throughput, certain configurations are more resource hungry and that certain configurations increase the inter-layer control complexity (as explained in section 5.6). More specifically, we observed that increasing the amount of the parallelly input channels for the convolution layers is preferable over increasing the amount of the parallelly output channels. This is justifiable: If we look at figure 5.1 we can see that by increasing the number of parallel outputs N_{par} , we also increase the number of "Window Generator" & "Pooling Layer" components. In fact, our experiments showed that a configuration of $M_{par} \times N_{par} = 2 \times 8$ needs almost x2 the LUT resources and x4.5 the Block RAM resources of the $M_{par} \times N_{par} = 8 \times 2$ configuration for a specific layer (figure 5.10). Moreover, when $M_{parallel}^L = M_{total}^L$, we no longer have to accumulate intermediate, partial 3D-convolution results and thus the Window-Gen component will no longer be needed, which frees some resources. Regarding the I&O constraints, we tried the following explorations:

- A space where $M_{parallel}^{L=1} \in [1,3]$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in [1,32]$. This is the case where we do not apply any constraint on the input & output of the Layers.
- A space where $N_{parallel}^L \leq M_{parallel}^L$ for all Layers, $M_{parallel}^{L=1} \in [1,3]$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in [1,32]$. Here we take into account that increasing the amount of the parallelly input channels for the convolution layers is preferable over increasing the amount of the parallelly output channels. Moreover, we allow the dimensions of the grid of Convolution Engines at each layer to not match completely the initial dimensions of the data that the Layer has to process (see section 5.6).
- A space where $N_{parallel}^L \leq M_{parallel}^L$ for all Layers, $M_{parallel}^{L=1} \in \{1,3\}$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in \{1,2,4,8,16,32\}$. Similar to the above, but we force the dimensions of the grid of Convolution Engines at each layer to be divisors of the initial dimensions of the data that they have to process.
- A space where $N_{parallel}^L = M_{parallel}^{L+1}$, $M_{parallel}^{L=1} \in \{1,3\}$ and $M_{parallel}^{L=2}, M_{parallel}^{L=3} \in \{1,2,4,8,16,32\}$, which leads to configurations with the easiest inter-Layer control.
- A space where $N_{parallel}^L \leq M_{parallel}^L$ for Layers 2 & 3 but not for Layer 1, $M_{parallel}^{L=1} \in \{1,2,3\}$, $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3} \in \{1,2,3,4,5,6,7,8,11,16,32\}$ and $N_{parallel}^{L=3} \in \{1,2,3,4,5,6,7,8,11,16,32,64\}$ which is the most fine-grained exploration between the ones listed. One can observe that due to the ceil function when calculating the latency, and the requirement that $M_{parallel}$ and $N_{parallel}$ should be divisors of M_{total} and N_{total} respectively, these are the only

configurations that have an impact on the throughput of the system, from within the $[1, 32]$ interval ¹.

As one can observe, the first space in the list above is a superset of the latter four. In each of the rest explorations we narrow the scope, to better screen for configurations of increased interest, due to the resource utilization and inter-layer control complexity to which they result. We search the design space by using Python scripts, calculating the throughput and the hardware time utilization for every configuration that lies within our constraints and then we manually evaluate the top candidates.

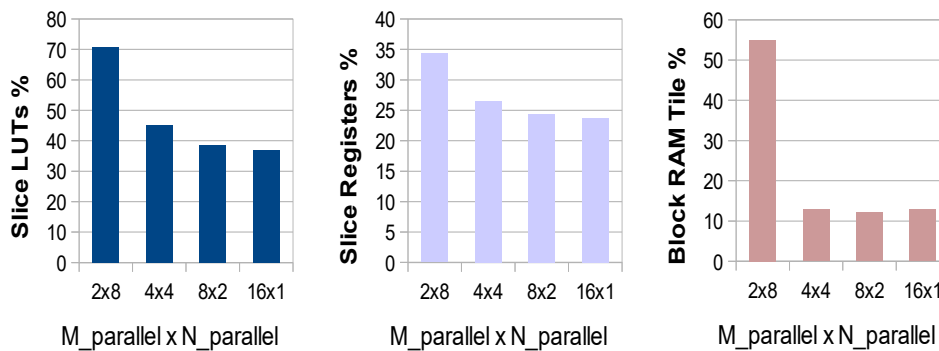


Figure 5.10: Device’s resource utilization, for different $M_{parallel} \times N_{parallel}$ configurations of the 2nd Layer, while keeping the number of Convolution Engines within the Layer constant. DSPs are not used in this example. Increasing the number of parallel inputs in the Layer is preferable over increasing the number of parallel outputs.

¹ For i from 1 to N , compute $a_i = \lceil \frac{N}{i} \rceil$. Keep a_1 as the first element. If $a_{i+1} \neq a_i$, keep a_{i+1} as the next element. For example, for $N=32$ that yields 32,16,11,8,7,6,5,4,3,2,1. For example, if we choose 9, which is not in the above list, then $\lceil \frac{M_{total}}{M_{parallel}} \rceil = \lceil \frac{32}{9} \rceil = 4$. We have increased the parallelism to more than 8, but we have not gained anything in terms of speed. We only spent more resources.

Per Layer Latency

Each 2D array needs a total of $C_1^L = E^2 + E \cdot X + b$ cycles to get convolved with one filter and produce one output channel (see section 5.2.2). Thus, for all the convolutions to be completed within a layer L , we need C_{total}^L cycles:

$$C_{total}^L = C_1^L \cdot \frac{M_{total}^L}{M_{parallel}^L} \cdot \frac{N_{total}^L}{N_{parallel}^L} \quad (5.1)$$

The equation 5.1 gives for each layer:

- **Layer 1** : $C_{total}^{L=1} = 1029 \cdot \text{ceil}(\frac{3}{M_{par}^{L=1}}) \cdot \text{ceil}(\frac{32}{N_{par}^{L=1}})$
- **Layer 2** : $C_{total}^{L=2} = 329 \cdot \text{ceil}(\frac{32}{M_{par}^{L=2}}) \cdot \text{ceil}(\frac{32}{N_{par}^{L=2}})$
- **Layer 3** : $C_{total}^{L=3} = 126 \cdot \text{ceil}(\frac{32}{M_{par}^{L=3}}) \cdot \text{ceil}(\frac{64}{N_{par}^{L=3}})$

This is an approximation of the layers' latency, which does not take into account the cycles needed for initialization and the control signals. However, it is rather accurate - $\sim 2\%$ off the true latency of the unified Layers- and we can still compare the latency of the different layers, identify which one acts as a potential bottleneck and perform the design space exploration. It is desirable that we find a configuration on $M_{parallel}$, $N_{parallel}$ for which $C_{total}^{L=1} \simeq C_{total}^{L=2} \simeq C_{total}^{L=3}$, so that all the layers will need about the same time to execute their operations.

Throughput

Throughput is measured in Images/sec. The images in this thesis consist of $28x28x3$ 8bit values each. We can classify one Image every Max Latency cycles, where $\text{Max Latency} = \max(\text{Latency}^{L1}, \text{Latency}^{L2}, \text{Latency}^{L3})$.

Since we use ping-pong buffers between the Layers, the Layers can work on different input images from each other simultaneously, thus throughput is dependent on the part of the design that needs the most time to complete its operation. If the operation of the Layers wasn't pipelined, throughput would be dependent on the sum of the Layers' latency instead. We want to find a configuration that will minimize the $\max(\text{Latency}^{L1}, \text{Latency}^{L2}, \text{Latency}^{L3})$ and thus increase the throughput.

$$\text{Throughput} = \frac{\text{Design's Clock Frequency}}{\text{Max Latency}} \left(\frac{\frac{\text{cycles}}{\text{sec}}}{\frac{\text{cycles}}{\text{Img}}} \right) \quad (5.2)$$

Hardware Time Utilization

Different $M_{parallel} \times N_{parallel}$ configurations result in a different distribution of the available resources among the Layers. We want to choose a configuration that will minimize the idle time of the Layers. The idle time is minimized when $Latency^{L=1} = Latency^{L=2} = Latency^{L=3}$.

Suppose that we map to Layer L , $\#V$ Convolution Engines. We are interested in whether these V Engines are working-non stop and thus these are resources well spent, or they are completing their load of work faster than needed, and thus we could map less Engines on that particular Layer. A metric to quantify this is the following:

$$\text{Working as} = \#V \text{ Engines} \cdot \frac{\text{Time spent Working}}{\text{Available Time}} \quad (5.3)$$

When Time spent working $<$ Available Time, these Engines are idle for a fraction of the execution time. By calculating equation 5.3 for every Layer, we can find the number of "Relative Engines". Ideally this would be equal to the number of Engines we actually mapped on the design. We then calculate the hardware time utilization. Total number of Convolution Engines $J = M_{par}^{L1} \cdot N_{par}^{L1} + M_{par}^{L2} \cdot N_{par}^{L2} + M_{par}^{L3} \cdot N_{par}^{L3}$

$$\text{Number of Relative Engines} = \sum_{L=1}^3 \frac{(M_{par}^L \cdot N_{par}^L) \cdot (Latency)^L}{(\text{Max Latency})}$$

$$\text{Hardware Time Utilization} = \frac{\text{Number of Relative Engines}}{J} \cdot 100\% \quad (5.4)$$

5.7.2 Results

While exploring the design space, we run multiple experiments. In all of these experiments we constrained the total number of Convolution Engines ≤ 44 , i.e. $M_{parallel}^{L=1} \cdot N_{parallel}^{L=1} + M_{parallel}^{L=2} \cdot N_{parallel}^{L=2} + M_{parallel}^{L=3} \cdot N_{parallel}^{L=3} \leq 44$, based on empirical observations of how the Layers synthesized and how many resources they consumed. The device that we target is a Xilinx Zynq Z-7020. Synthesis and Implementation were executed for each Layer separately, in Xilinx's Vivado Design Suite and both were run under the "RunTimeOptimized" directive to reduce the computational time needed for each run. We also forced the tool to use the available DSP resources to implement a part of the design, while the rest was implemented on the available LUTs. However we should note that the use of the available DSPs is not optimized. Our Layers use 4-bit inputs and 8-bit weights, as explained in section 4.4, meaning that each Convolution Engine performs multiple multiplications of 4 x 8 bit simultaneously¹. Each of the on-chip DSPs can perform 18 x 25 MACCs, thus more than one of the 4 x 8 bit multiplications could be mapped on the same DSP². This optimization technique is an option that we didn't make use of during this thesis and currently each 4 x 8 bit multiplication occupies a different DSP. Having in mind all of the above, in the best scenario, we managed to fit a total of 38 Convolution Engines on the Zynq Z-7020 device, leading to about 75% LUTs utilization and 90% DSPs utilization. We expect the rest of the available LUTs to be used for the routing of the design. To calculate the throughput, we used the same clock for the whole device, at 100MHz frequency. This clock is a bit slower than the clock that the implementation tool reported for the slowest part of the design, to take into account that the timing constraints will not be as easily met when the Layers will be interconnected and the full design placed and routed. We also examined whether allowing the different Layers to run with a different clock frequency from each other, could lead in a re-distribution of the Convolution Engines from the fastest Layer to the slowest in order to increase the throughput. However, with the amount of resources available on the Zynq Z-7020 device, this was not a fruitful path in terms of throughput performance.

Figure 5.11 illustrates the results from exploring the design space for three different numbers of Convolution Engines. This figure corresponds to an exploration where $N_{parallel}^L \leq M_{parallel}^L$ for all Layers, $M_{parallel}^{L=1} \in [1,3]$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in [1,32]$. We can see that the throughput is not a function of the hardware time utilization, given that there are configurations within the same group of total Convolution Engines that have the same throughput, but widely different hardware time utilization. Moreover, while we get the sense that for a constant number of total

1 Specifically, $k \times k = 5 \times 5 = 25$ multiplications at each Convolution Engine, where k is the width of one kernel.

2 Explained in Xilinx's Whitepaper "Deep Learning with INT8 Optimization on Xilinx Devices."

Convolution Engines, higher throughput comes with higher hardware time utilization, this is actually not the case for every configuration. At first, this might strike as a bit confusing, as one expects that when hardware time utilization is increased, we have distributed the available Convolution Engines in a more effective manner between the Layers and thus that the latency of the slowest Layer should be decreased. However it turns out that when we increase the amount of Convolution Engines in a Layer, we do not necessarily decrease its latency. This comes as a result of the fact that we have allowed the parallel inputs and outputs of each Layer to take values that are not divisors of the initial values of the respective total channels, which does not turn into our favour in every configuration. For example, if we assign to the third Layer, 18 Convolution Engines in a 9x2 configuration, this results in a latency= $329 \cdot \text{ceil}(\frac{32}{9}) \cdot \text{ceil}(\frac{32}{2}) = 329 \cdot 64$, whereas a 8x2 configuration of 16 Convolution Engines also results in latency= $329 \cdot \text{ceil}(\frac{32}{8}) \cdot \frac{32}{2} = 329 \cdot 64$. In that case, we have spent more resources but did not gain in performance. On the other hand, when we force $M_{parallel}^{L=1} \in \{1,3\}$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in \{1, 2, 4, 8, 16, 32\}$, the ceil function does not affect the calculation of the Layers' latency and the aforementioned situation does not emerge. Then, within a constant number of Convolution Engines, an increase in the hardware time utilization signifies an increase in the throughput of the system. Also, an increase in the number of Convolution Engines for a particular Layer signifies a decrease in its latency, as expected. However, such a restriction in the parallel inputs and outputs, may lead in sub-optimal solutions (see figure 5.11). It turns out ¹ that the most fine-grained space regarding the throughput performance, from which one should choose the parallelism of the Convolution Engines is when $M_{parallel}^{L=1} \in \{1,2,3\}$, $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in \{1, 2, 3, 4, 5, 6, 7, 8, 11, 16, 32\}$ and $N_{parallel}^{L=3} \in \{1,2,3,4,5,6,7,8,11,16,32,64\}$. In any case, one can observe that the same throughput may be obtained with less Convolution Engines, when these Engines are better distributed between the Layers and thus better used in time. It is evident that the various configurations result in widely different performance of the system. In the results of the exploration illustrated in figure 5.11, when using 38 Convolution Engines, the throughput jumps from ~ 295 images/sec in the worst case scenario to ~ 4749 images/sec in the best.

When the throughput is the same, the various configurations can be evaluated based on the device's resources utilization to which they will lead and the complexity of the inter-Layer control that they will impose. Hardware time utilization is considered in that case as less important. For example, for the results of the exploration illustrated in figure

1 Given a specification of L identical operations to be performed and searching for the number of J identical parallel machines in which to distribute them: Each machine can perform a single operation at a time and the single operation itself cannot be split. All machines start computing at the same time. The set of numbers from which one should choose the parallelism of the machines can be calculated as follows: For i from 1 to L , compute $a_i = \lceil \frac{L}{i} \rceil$. Keep a_1 as the first element of the set. If $a_{i+1} \neq a_i$, keep a_{i+1} as the next element. This can be used to separately calculate each dimension of the grid at each Layer.

5.11 the best throughput is achieved with twelve different configurations of $(M_{parallel}^{L=1} \times N_{parallel}^{L=1}, M_{parallel}^{L=2} \times N_{parallel}^{L=2}, M_{parallel}^{L=3} \times N_{parallel}^{L=3})$. Some of these significantly increase the inter-layer control complexity ($[3 \times 2, 9 \times 2, 7 \times 2]$) and present the highest hardware time utilization, others exceed the available LUT resources ($[3 \times 2, 4 \times 4, 4 \times 4]$) and some satisfy the resources' constraints, while they present relatively easy inter-layer control ($[3 \times 2, 16 \times 1, 16 \times 1]$) with slightly lower hardware time utilization than the highest observed. We decided to prefer the latter one over the first. As long as the routing of the design is not a problem, we decided to cling on the configurations with the least complex inter-Layer control, despite occupying a bit more of the device's resources. Although we treated the hardware time utilization as less important than the other parameters in the discussion above, this metric can be used to find configurations of increased interest in terms of power: If there is the possibility to temporarily, completely switch off parts of the design, configurations with lower hardware time utilization should be preferred, since this would lead to less switching activity and thus to reduced power consumption.

In the exploration where we did not apply any other constraint in the inputs and outputs of the Layers - other than the total number of Convolution Engines ≤ 38 - the highest throughput was still at ~ 4749 images/sec. Thus, constraining $N_{parallel}^L \leq M_{parallel}^L$ for all Layers is not only necessary in order to reduce the amount of the device's resources needed and fit our design on the Zynq Z-7020, but it also doesn't have any negative effect on the achieved throughput. When examining the exploration where $N_{parallel}^L = M_{parallel}^{L+1}$ and the Convolution Engines are equal to 38, the design space consists of only two points, one of which corresponds to a configuration that can run with the maximum throughput ($[3 \times 2, 2 \times 8, 8 \times 2]$). This configuration is of particular interest due to the fact that it leads to the least complex inter-Layer control. Unfortunately it is also a configuration that exceeds the device's available LUT resources and thus it is not a feasible one.

To conclude this section, we turn our focus again to the relation between the hardware time utilization and the throughput, but this time we do not take into account the inter-Layer control complexity (figure 5.11). It is not profound which configuration should one choose in each case. We may observe the following:

- Given a specification for the throughput, the threshold throughput may be achieved with less Convolution Engines, if these engines are distributed between the Layers in an effective manner. That translates to less resources needed, thus the possibility of using a smaller device, or having enough resources to also implement some other function on the same device. We search for configurations with high hardware time utilization.
- Yet, for a constant throughput and a constant number of total Convolution Engines, we prefer configurations with smaller hardware time utilization. These configurations will lead to reduced power consumption, especially if the technology and the available developing time offer the possibility of designing the circuit of

the Layers to switch off, for the fraction of time that they would otherwise be idle.

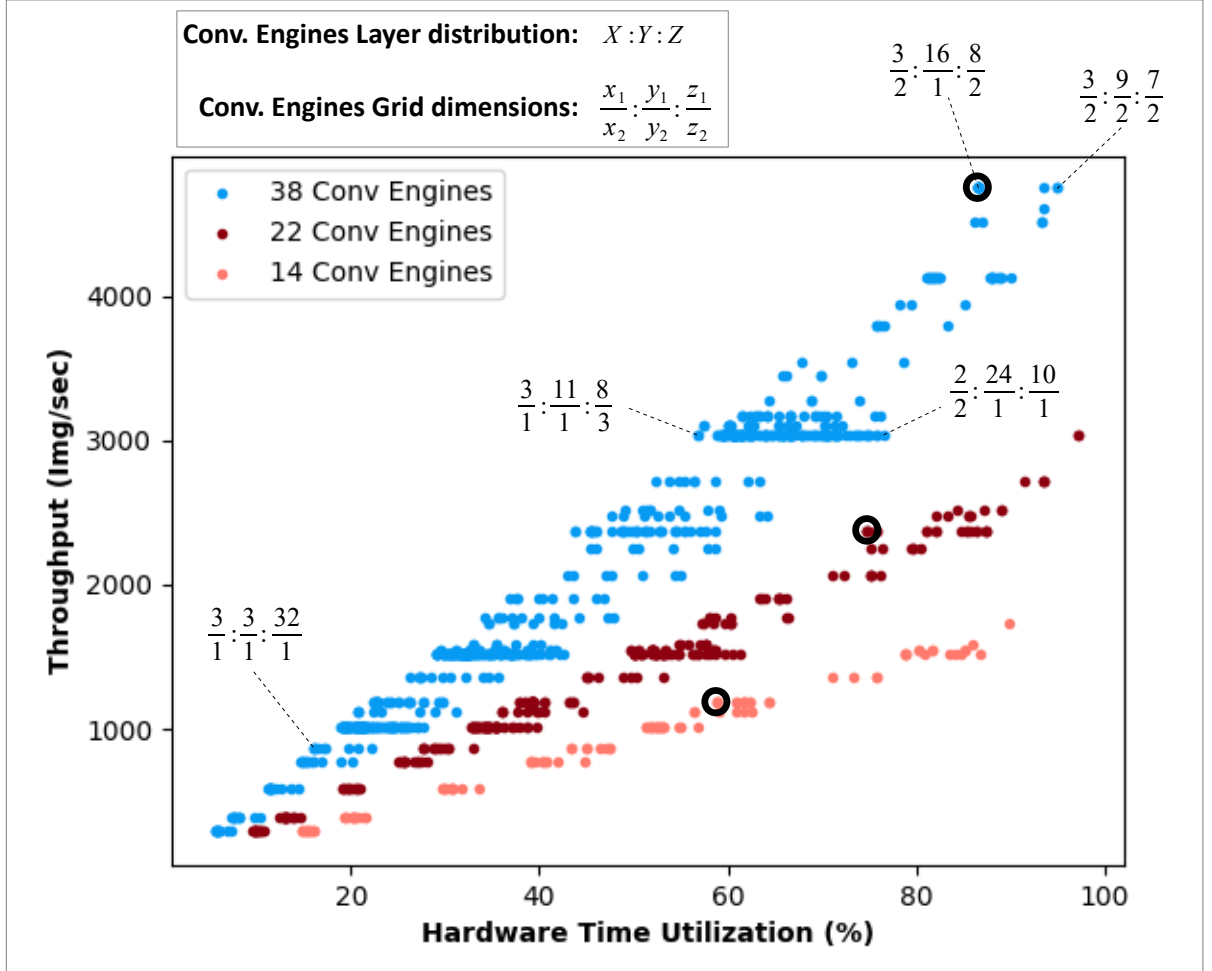


Figure 5.11: Results from exploring the design space for three different number of total Convolution Engines in the design, when $N_{parallel}^L \leq M_{parallel}^L$ for all the Layers, $M_{parallel}^{L=1} \in [1, 3]$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in [1, 32]$. Total Convolution Engines = $X + Y + Z$ and $X = x_1 \cdot x_2, Y = y_1 \cdot y_2, Z = z_1 \cdot z_2$.

With black circles denoted the configurations in which $M_{parallel}$ and $N_{parallel}$ are divisors of the initial M_{total}, N_{total} , i.e. $M_{parallel}^{L=1} \in \{1, 3\}$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in \{1, 2, 4, 8, 16, 32\}$. This proves that it is legitimate to not restrict the parallel inputs and outputs in such a way, as it may lead to sub-optimal solutions.

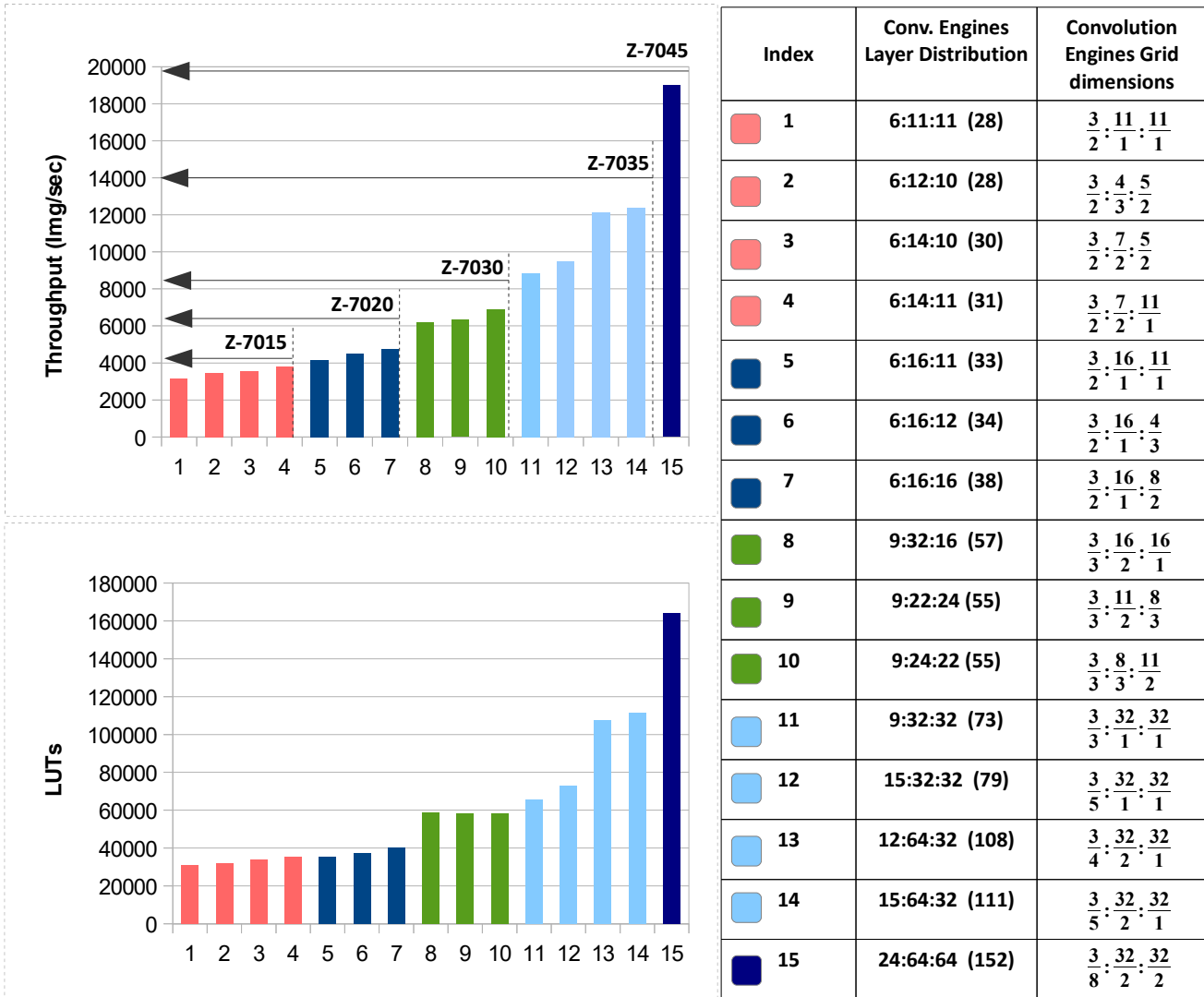


Figure 5.12: Results of extending the Design Space Exploration to other devices of the Zynq-7000 SoC family. The achieved throughput (Img/sec) and the amount of LUTs needed for each configuration is illustrated. The smallest device possible in which the "Modified Cifar-10 Full" CNN can be implemented is the Z-7015, due to the on-chip memory requirements. To compute all the convolutions of this particular CNN parallelly, a total of $3 \times 32 + 32 \times 32 + 32 \times 64 = 3168$ Convolution Engines would be needed and a huge amount of Block RAMs.

5.8 Final Configuration and FPGA Implementation Results

Through the exploration of the design space we determined that the configuration [3x2,16x1,8x2], presented in figure 5.14, is one of the configurations that yield the maximum throughput, for the amount of resources that we have occupied on the device. A detailed overview of the deliverable design's properties can be seen in figure 5.15. Unlike the Caffe framework, our design does not support batch processing. Instead it can classify only one image at a time. In order to achieve the throughput of 4650 classified images per second, the FPGA device must also be fed with images at its input at this rate. This means that we need an input bandwidth = $4650 \cdot 28 \cdot 28 \cdot 3 \cdot 8 = 87.5 \text{ Mbit/sec}^1$. In the Microlab-NTUA we have measured that the programmable logic on the SoC can communicate with the processing system at a rate of $\sim 3 \text{ Gbit/sec}$. Moreover, the Zynq Z-7020 SoC can be fed with the images using a 100Mbit/sec Ethernet cable to connect it to an external camera, etc. Thus the desired input bandwidth for the FPGA is achievable and the time needed for data transfers does not pose a bottleneck. The throughput of the design is currently limited only by the clock frequency at which it operates and the number of the device's available resources.

For the purposes of this thesis it was not deemed necessary to perform in-circuit validation of the design and integrate it to a working, demonstrator system. Rather, we aimed to accurately estimate the cost and the performance of a Convolutional Neural Network implemented on an FPGA device. With minimal effect on the conclusions of this thesis, the final state machine that coordinates the different layers of the network has not been designed, although their interconnection has been outlined in section 5.6. Moreover, the Softmax Classifier was not mapped on the design, as it was not deemed necessary².

-
- 1 The image is represented with 8-bit values, but our network will need only 4bits of each value at its input. Currently we have configured this truncation to happen on the FPGA chip. Although half of these bits are unused and their transfer & storage is pointless and expensive, this choice makes our design more adjustable as far as the arithmetic precision and the dynamic fixed point strategy is concerned. Moreover, we regard the quantization phase as part of the algorithm. We would like this design to be an "all-included" CNN inference accelerator, without the design posing a list of special needs to its surroundings in order to be interconnected.
 - 2 The Softmax Classifier takes the class scores produced from the rest of the network and normalizes them in the [0,1] range, to interpret them as probabilities. This does not change the "winning" class, but serves as a way to comprehend the algorithm's results.

FPGA chip on Xilinx Zynq Z-7020 SoC	
7 Series PL Equivalent	Artix-7
Logic Cells	85K
Look-Up Tables (LUTs)	53200
Flip-flops	106400
Total Block RAM	4.9 Mb
DSP Slices	220

Figure 5.13: Characteristics of the Programmable Logic on the Xilinx Zynq Z-7020 SoC.

	L1	L2	L3	FL
Engines	3x2	16x1	8x2	6 (2 inputs each)
Slice LUTs %	13.33	35.97	26.49	0.08
Slice Registers %	8.79	23.75	19.8	0.02
Block RAM Tile %	15.71	18.56	20.71	2.14
DSPs %	0	0	90.91	8.18
Latency (cycles)	16574	21502	17022	288 (as stand alone) 4 (attached to L3)
Possible Clock (ns)	6	9	8	-
Dynamic Power (W)	0.412	0.46	0.752	0.026
Device Static Power (W)	0.106	0.107	0.11	0.101

Figure 5.14: The configuration of the deliverable design's Layers. Results obtained through synthesis, implementation & simulation with Xilinx's Vivado Design Suite, for the Zynq Z-7020 SoC.

Deliverable Design			
Algorithm : "Modified Cifar-10 Full" Convolutional Neural Network			
Target Platform	Xilinx Zynq Z-7020	Convolution Engines	38
Slice LUTs %	75.81	Latency (ms)	0.5510
Slice Registers %	52.36	Throughput (Images/sec)	4650
Block RAM Tile %	57.12	Hardware Time Utilization %	87.6
DSPs %	99.09	Image size (RGB)	28x28x3, 8-bit values
Clock (ns)	10 (100 MHz)	Word Length	Inputs & Outputs : 4 bit Conv Weights : 8 bit FC Weights: 2 bit
Total On-chip Power (W)	1.76	CNN's Classification Accuracy	94.89%
Designed with	VHDL	Workload per Image (Million OPs)	18.864

Figure 5.15: The properties of the Deliverable Design on the Zynq Z-7020.

Since the deliverable design uses only the on-chip memory, it cannot process images of any size. Given the available RAM of the Programmable Logic on the Zynq Z-7020 SoC, this design can handle images of at least up to three times the size of the SAT-6 airborne dataset's images¹, if needed. The execution time is quadratically related to the input size, as we already expected by equation 5.1 (figure 5.16).

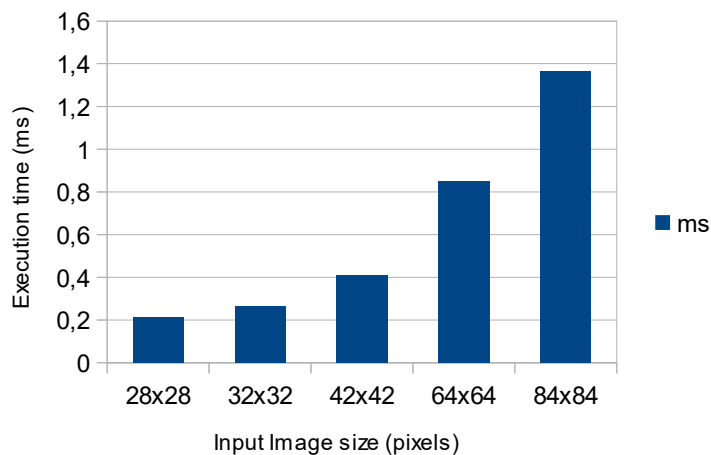


Figure 5.16: Execution time of the "Modified Cifar-10 Full" CNN on the proposed configuration for Zynq Z-7020, in relation to the size of the input image. Zynq Z-7020 can handle images of up to $\sim 50 \times 50$ pixels.

In figure 5.17 we present how the choice of the data word-length affects the amount of resources needed by the deliverable design (configuration [3x2,16x1,8x2]). Five different scenarios are taken into account: **(i)** 4-bit activations & 8-bit weights, which is the proposed word-length. **(ii)** 8-bit activations & 8-bit weights, **(iii)** 16-bit activations & 8-bit weights, **(iv)** 8-bit activations & 16-bit weights, **(v)** 16-bit activations & 16-bit weights. Both the amount of registers and Block RAM needed is within the resources of the Z-7020 for all the different word-length scenarios, however the amount of LUTs needed quickly exceeds the available resources of the Xilinx Zynq Z-7020. The characteristics of the Programmable Logic in the Xilinx Zynq Z-7020 SoC can be seen in figure 5.13. We remind that the proposed word length is based on exploring the relation of the word-length to the accuracy of the specific CNN, on the specific dataset, in section 4.4.

¹ Taking into account only the RGB channels, as explained in section 4.3.3.

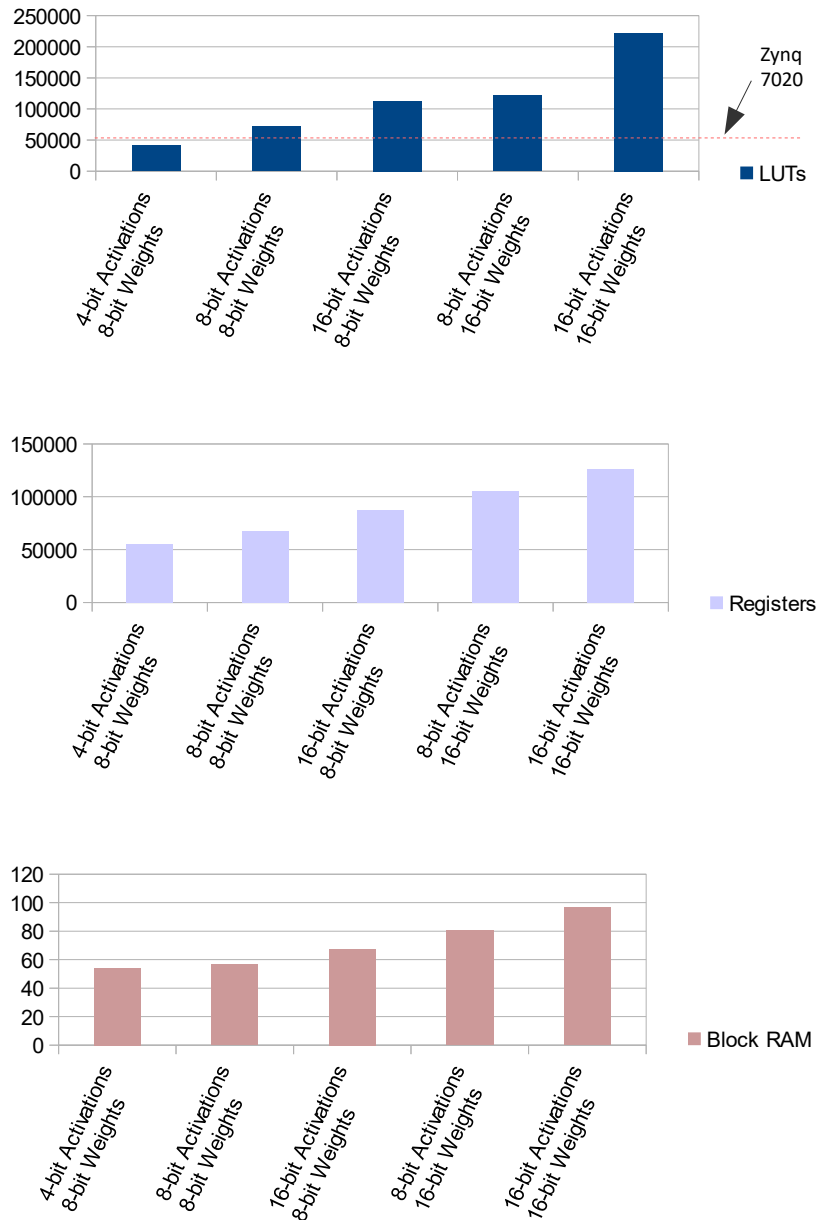


Figure 5.17: The amount of FPGA resources needed in relation to the word-length of the data. Design configuration: [3x2, 16x1, 8x2]. Both the amount of registers and Block RAM needed is within the resources of the Z-7020 for all the different word-length scenarios presented. However, the LUTs needed quickly exceed the available resources of Zynq Z-7020. The proposed word-length is to use 4-bits for the activations and 8-bits for the weights of the "Modified Cifar-10 Full" CNN.

5.8.1 Comparison to relevant works

In this section we compare the estimated performance of our design to other implementations and technologies. To the best of our knowledge there hasn't been another FPGA implementation of a CNN that targets the SAT-6 airborne dataset. However, the CNN model that is used in this thesis originated from the "Cifar-10 Full" model in the Caffe framework's source code and resembles the original model a lot. We compare our FPGA architecture to implementations that use either exactly the same, or a close variation of this CNN model. While these CNN models resemble each other greatly and thus we believe that the comparison is valid, the properties of our design are also related to the specific dataset that we used. In particular, the bit-width precision that we decided to use in our design, was chosen based on its effect to the accuracy of the "Modified Cifar-10 Full" CNN model on the SAT-6 airborne dataset. The particular application of a CNN and the underlying dataset is often treated as indifferent when the primary target is the acceleration of the CNN's inference. However, it is the target application that dictates which CNN model is adequate and a key factor of whether and to what extent low bit-width approximations can be used.

We compare our design against: **(i)** an implementation of the "Cifar-10 Quick" CNN, when the Caffe framework is executed on the Fathom Neural Compute Stick **(ii)** an implementation of the "Cifar-10 Quick" CNN on the Movidius Myriad2 platform [Xyg17], **(iii)** an implementation of the "Cifar-10 Quick" CNN, when the Caffe framework is executed on the Intel Xeon E5-2650 v2 CPU, **(iv)** an implementation of the "Cifar-10 Full" CNN when the Caffe framework is executed solely on the dual core, Cortex-A9 MPCore ARM processor, located on the Zynq Z-7020 SoC, **(v)** an implementation of the "Cifar-10" CNN on the Xilinx Zynq-7000 XC7Z045, produced by the FpgaConvNet framework [Ven17] and **(vi)** An implementation of the "Cifar-10" CNN on the Xilinx Zynq-7000 XC7Z045, produced by the Haddoc2 framework [Abd17]. Since most of these implementations use images of size 32x3x3, we also use measurements of our design's performance for that size of input.

The first four implementations above (i, ii, iii & iv) are all software implementations and thus they are not bound to the specific architecture of a CNN model. They can be flexible and execute the inference of various CNNs topologies. The device on the first implementation is a Fathom NCS. This is a commercial Neural Network accelerator by Movidius, that utilizes the Myriad2 vision processor. The second implementation is also on the Myriad2 platform (model MA2150) and was produced in the Microlab-NTUA as part of a diploma thesis. It uses 16-bit floating point arithmetic and it is optimized on the assembly level. The implementations on the Intel Xeon E5-2650 v2 and on the Cortex-A9 ARM processor, use the standard 32-bit floating point arithmetic that comes with the main fork of the Caffe framework. The Intel Xeon E5-2650 v2 is a processor intended for server applications, with 8 cores, 16 threads and 2.60 GHz of base processor frequency. The "Cifar-10 Quick" CNN used in the above implementations

has one more fully connected layer than the "Modified Cifar-10 Full" used in this thesis. This extra layer would increase the latency of our deliverable design, but would not affect its throughput. Thus we do not regard this difference as an important one in our comparison. The FpgaConvNet and the Haddoc2 are frameworks that automate the mapping of a CNN topology on an FPGA device. Both are compatible with the Caffe framework. FpgaConvNet creates synthesizable HLS code taking into account the target platform, while the Haddoc2 generates platform-independent VHDL synthesizable code. In the current comparison we have metrics of these frameworks when they are implemented on the Xilinx Zynq-7000 XC7Z045. However, this device is much bigger and resource rich than the Zynq Z-7020 that we targeted. In fact, it has ~ 4 times the LUT resources and ~ 4 times the DSP resources of the Z-7020 device. Moreover, these frameworks are very different from each other: The FpgaConvNet is a framework that can map a huge number of different CNNs topologies. It utilizes partial reconfiguration of the FPGA device, by creating different bit-streams for the different layers of the network and does not rely solely on the on-chip memory. In this particular implementation of the "Cifar-10" CNN, it uses 16-bit fixed point precision and runs at 125MHz. Haddoc2 on the other hand, can only map small CNN topologies, like the Cifar-10. That is because, it relies solely on the on-chip memory of the FPGA. Moreover, Haddoc2 is fully unrolling all the computations of the CNN, by physically mapping every single multiplier needed for the convolution layers to its own resources on the device. By fully unrolling all the computations, Haddoc2 can fully pipeline the execution of consecutive layers. Moreover, the multipliers can be specialized to their constants (e.g zero, one, powers of two). Haddoc2 uses 6-bit fixed point precision and runs at 54.17MHz. In a sense, the delivered design of this thesis, is closer to the implementation produced by the Haddoc2 framework, than to the rest implementations discussed above -but for a significantly smaller target device than that of the Haddoc2. We compare these implementations regarding their throughput, for consecutive images (batch=1). The "Cifar-10" consists of ~ 24.8 Million Operations (1 MAC Operation = 2 Operations) for each image. The results of the comparison discussed above are illustrated in figures 5.18 and 5.19.

Implementation Device	CNN Model	Input Image Size	Programming	Bitwidth	Freq	Power	Throughput (lmg/sec)
VPU: Fathom NCS [Xyg17]	"Cifar-10 Quick"	32x3x3	Caffe software	16 fixed	N/a	N/a	232
VPU: Myriad2 [Xyg17]	"Cifar-10 Quick"	32x3x3	C & Assembly	16 fixed	600 MHz	1,09 W	909
CPU: Intel Xeon E5-2650 v2, 4 cores [Xyg17]	"Cifar-10 Quick"	32x3x3	Caffe software	32 float	2.6 GHz	N/a	1000
CPU: Cortex-A9 ARM [Dan18]	"Cifar-10"	32x3x3	Caffe software	32 float	N/a	N/a	10
FPGA: Xilinx Zynq Z-7020	"Cifar-10"	32x3x3	Automatically generated HLS	16 fixed	125 MHz	N/a	6080
FPGA: Haddloc2 [Abd17]	"Cifar-10"	32x3x3	Automatically generated VHDL	6 fixed	54.17 MHz	N/a	17633
This Thesis	"Modified Cifar-10 Full"	32x3x3	VHDL	Dynamic Fixed Point Inputs & Outputs : 4 Conv Weights : 8 FC Weights : 2	100 MHz	1.76 W	3778

Figure 5.18: Comparison of different implementations, executing the inference of the "Cifar-10" CNN.

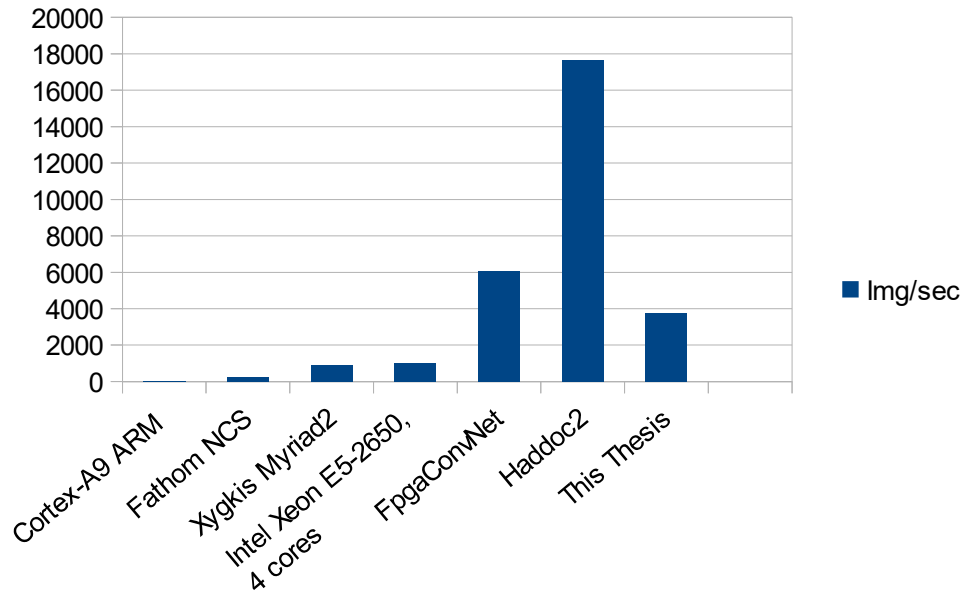


Figure 5.19: Comparison of different implementations executing the inference of the "Cifar-10" CNN. FpgaConvNet and Haddoc2 implemented on Xilinx Zynq Z-7045 device. This thesis implemented on the smaller Zynq Z-7020 device.

Looking at figures 5.18 and 5.19 it is clear that the hardware implementations dominate the software ones in terms of throughput, which is expected. Our deliverable design on the Xilinx Zynq Z-7020, achieves a x4 speedup on throughput over the Myriad2 platform and a x377 speedup on throughput over the Cortex-A9 ARM processor. As far as the FPGA implementations are concerned, our design is out-performed by the implementations of the FpgaConvNet and the Haddoc2 frameworks. However, these implementations are using a device that has x4 times the resources of the Zynq Z-7020. In order to get a better feeling of how our design competes against the implementations that these two frameworks produce, we synthesized a version of our design on the Zynq Z-7045, without performing a meticulous design space exploration like before. The next figure 5.20 illustrates a comparison of the three implementations, when targeting the same device.

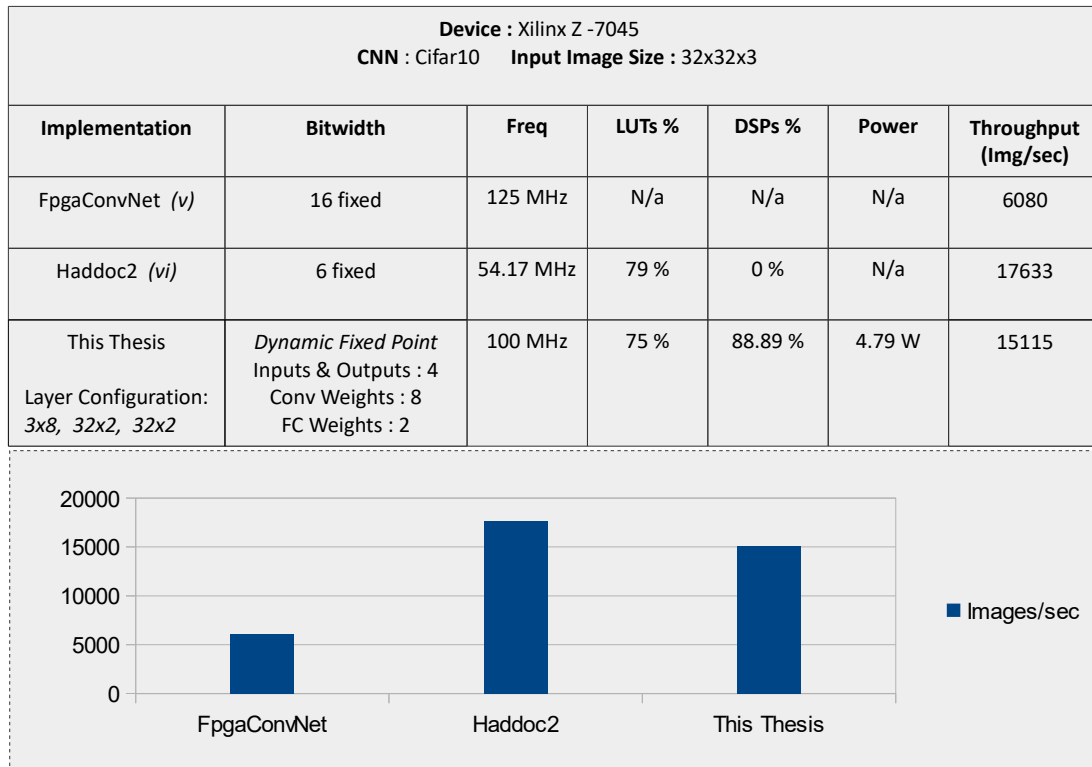


Figure 5.20: Thesis design vs designs generated from the FpgaConvNet and the Haddoc2 frameworks, when targeting the same FPGA platform.

When targeting the same device, Xilinx’s Zynq Z-7045, Haddoc2 still outperforms our design, by a $\times 1.17$ factor. Moreover, while Haddoc2 reports that it completely unrolls all the operations of the CNN, our design could not reach the full parallelization state, as it didn’t fit on the device. The optimal target in terms of throughput for our design would be a configuration of $\{3 \times 32, 32 \times 32, 32 \times 64\}$, where all the input planes of a Layer would be simultaneously processed and all the output planes simultaneously generated. Instead, the configuration presented in figure 5.20, is a $\{3 \times 8, 32 \times 2, 32 \times 2\}$ configuration. The reason we achieve a throughput close to that of the Haddoc2 implementation, is because our design can run with a much faster clock. One thing to consider here is that Haddoc2 uses 6-bit weights for the convolution filters, of which 33.78% are zero parameters, 45.32% one parameters, 16.4% power-of-2 parameters and only 4.5% are parameters of other values. That means that the need for actual multipliers on their implementation is drastically reduced, which frees a lot of the device’s resources. On the contrary, in the set of weights that the deliverable design of this thesis uses, we have 14.5% zero parameters, 12.5 % one parameters, 14.4% power-of-2 parameters and 58.5% parameters of other values. This results in an increased need for resources in our design, comparatively to Haddoc2’s implementation.

Compared to the HLS implementation of the FpgaConvNet [Ven17] and an HLS implementation on the Zynq Z-7020 reported by [Dan18], the design in this thesis achieves $\times 2.4$ times their throughput when targeting the same device. We keep in mind that these implementations use 16-bit fixed point arithmetic.

Shifting away our focus from the throughput performance, and to conclude this Chapter, we have to observe that frameworks that automatically map CNN models to FPGA, like FpgaConvNet and Haddoc2, truly dominate our design in some other aspects: (i) the developing time and (ii) the flexibility to implement various CNN models.

CHAPTER 6

Conclusions

This thesis delivers FPGA architectures designed in VHDL for the inference of Deep Convolutional Neural Networks classifying images, using only the on-chip memory of the Programmable Logic. It also delivers the "Modified Cifar-10 Full" CNN, a low bit-width customized CNN model, created and trained with the Caffe framework on images of the SAT-6 airborne dataset¹. The SAT-6 airborne dataset consists of image patches, each one of size 28x28 pixels with 1m spatial resolution, covering six land cover classes: barren land, trees, grassland, roads, buildings and water bodies. Our customized, "Modified Cifar-10 Full" CNN achieves 94.89% top-1 accuracy on the SAT-6 airborne dataset, using 8-bit weights and 4-bit feature maps. The size of the "Modified Cifar-10 Full" CNN allows us to perform the whole processing and control of the algorithm on the FPGA chip, without a need for an external memory to store the intermediate results, or a CPU to coordinate and monitor the algorithm's execution. When mapped on the Xilinx Zynq Z-7020 SoC, the design operates at 100MHz, can classify 4650 Images per second and consumes 1.76 Watt ² (figure 5.15). Our design on the Zynq Z-7020 achieves a x377 speedup on throughput over the embedded Cortex A9-Arm processor. Compared to HLS implementations from relevant work our design achieves a x2.4 speedup on throughput when targeting the same FPGA device. This is a high accuracy, high throughput, low power FPGA design of Convolutional Neural Networks, suitable for embedded systems placed on UAVs-drones/satellites, classifying images at the edge of the computing cloud, without the need of transmitting the data of every image to a server.

1 Saikat Basu, et al. "DeepSat - A Learning framework for Satellite Imagery, ACM SIGSPATIAL 2015", [Bas15]

2 Without counting the energy needed for the data transfers to and from the FPGA chip.

Reviewing the recent developments in the field of Machine Learning, regarding Convolutional Neural Networks and the prior work in the approaches of designing hardware CNN implementations were among this thesis goals. Comprehensive reviews examining the exact same issues were made available at the time of concluding this thesis [Abd18a], [Guo17], [Zha18]. Therefore, extensively reporting the results of our preliminary literature review became redundant. Instead a general overview of the field is outlined in chapter 3.

Concluding this thesis, we discuss several possible improvements and directions for future work:

- **Create a fully functional prototype of the system designed.** In this thesis we accurately estimated the cost and the performance of a Convolutional Neural Network for satellite image classification, implemented on an FPGA device. It was not deemed necessary to perform in-circuit validation of the design and integrate it to a working, demonstrator system, which remains to be addressed in the future.
- **Optimize RTL-level pipeline.** Currently there is plenty of room for optimization in the RTL code. Long paths with multiple levels of logic have been identified. The target is to reduce the levels of logic in the critical paths, by inserting flip-flops in between, trading-off resources and increasing the latency, for an increase in the maximum clock frequency of the design. This way the throughput will be drastically increased. We expect that through optimization, an x2 increase in the frequency of the clock is possible.
- **Reconsider the architecture of the Convolution Engines.** Although it is not profound, the architecture of the Convolution Engines that we used results in every Convolution Engine creating its own window of input elements (figure 5.2). However, multiple Convolution Engines work simultaneously, while reading the same input. We should consider exploring the trade-off of extracting the memory elements outside the Convolution Engines and use a component external to the Engines to create the window of the parallel registers that will feed all the Convolution Engines within a layer. This will result in an increased fan-out for the window's registers, but will also reduce the amount of resources needed for each Convolution Engine. Moreover, with such an approach, every Convolution Engine will essentially be a MAC component, which the design tools are very good at optimizing. We believe this is a direction worth exploring.
- **Extend the library of our components.** Currently we have designed components that can perform the convolution operation, the rectified linear unit (ReLU), the max pooling, the inner product (fully connected layer) and the zero padding of the feature maps. However, new types of layers are constantly being created. Even more importantly, new types of interconnections have also been proposed. Of particular interest is the "Residual block" used in variations of the ResNet network, which forwards the output of a convolution layer not only to the layer

after it, but also to layers that are several steps ahead. This poses an interesting problem regarding the pipelined operation of these layers.

- **Advance the Design Space Exploration.** When searching the design space, we set two optimization goals: throughput and hardware time utilization. We used empirically observed constraints and python scripts to calculate the throughput and hardware time utilization for all the configurations that were inside the space that our constraints defined. Then, we manually evaluated the top candidates and synthesized several different versions of the design, examining how the dimensions of the grid of Convolution Engines affects the resource utilization, in order to identify the best configuration within the resources of each device. There is plenty of room for constructing a more well formulated method and an automated tool that will take into account the multiple constraints of both the design and the target device and search for configurations under multiple optimization goals.
- **Further explore techniques of compressing the CNN models' size.** In this thesis we decided to compress the size of the network, by exploring low bit-width approximations of its weights, in relation to the network's accuracy. The *Dynamic Fixed Point Approximation* method was applied, using the "Ristretto" tool [Gys16], which calculates how the accuracy of the CNN model is affected by the numerical precision. Hardware implementations benefit greatly from using lower bit-width, while CNN models can work with limited numerical precision, without a great loss in accuracy. However this is not the only option to compress a network: Pruning some of the network connections, encoding the parameters of the model to reduce their memory and several other techniques have been developed [Han15]. This is an exciting path and a very useful one, in order to implement CNN models on low-power embedded systems.
- **Explore algorithmic optimizations of the convolution operation.** In order to accelerate the execution of the convolution layers, several approaches have been proposed, including Fast Fourier Transform, Winograd Transform and more [Abd18b]. During this thesis we did not take them into account, so this is an area of future study.
- **Study the problems that emerge when accelerating the inference of larger CNNs.** In this thesis, the network we used could fit entirely on the FPGA's on-chip memory. With larger networks this is usually not the case and the memory bandwidth becomes the main bottleneck in the execution time. Moreover, Convolutional Neural Networks nowadays are becoming increasingly deeper and bigger. Thus, studying the questions that arise -and the solutions that have been proposed- regarding when and which data to move, is essential for designing hardware implementations of CNNs.
- **Move towards the automation of CNN to FPGA mappings.** Lately, several toolflows for mapping Convolutional Neural Networks on FPGAs were

published [Ven18]. These tools enable the fast deployment of CNN models on FPGA devices, minimizing the developing time, and can work for a variety of CNN architectures. Considering that new architectures of CNNs are continuously proposed, the adaptability of these tools is very important. In contrast, the deliverable design of this thesis, although not bound to the specific architecture of the "Modified Cifar-10 Full" CNN, exhibits limited flexibility, mainly due to the constraint of the on-chip memory. Moreover, it requires re-configuration from specialized design engineers and significantly increased developing time. Nevertheless, working on the level of Hardware Description Languages to map CNNs on FPGAs is neither outdated, nor in vain: Many of these frameworks are based on hand-crafted, highly optimized computations engines, in order to achieve increased performance. Moving towards the creation of tools that can take the high-level description of a CNN and efficiently map it on a target FPGA device, is a direction that the author of this thesis is very keen about.

- **Approach the design of CNN architectures in relation to the complexity of the classification task.** Currently most of the work in the deep learning field is towards creating new architectures of CNNs that will out-perform the previous generations in terms of accuracy. There is little understanding of how deep a neural network should be and how many parameters it should have when targeting a specific problem. Several research studies are in the direction of establishing the theoretical foundation of deep artificial neural networks, which is imperative in order to effectively harness these powerful computational tools.
- **The problem of nomenclature.** Although this is not a direction for future work, we have to acknowledge it as a problem: Currently there is not a unified nomenclature in the field of Machine Learning and several concepts and techniques keep re-appearing with different names, while several terms are used interchangeably despite describing different things. What is Machine Learning, how it is distinct from Artificial Intelligence and what is Deep Learning? Are they called Multilayer Perceptrons, regular Artificial Neural Networks, or Deep Feed Forward Neural Networks? How does one calculate the depth of a network? Do only the convolution layers count for CNNs, or are the pooling layers also taken into account? Even for the input and output of the convolution layers we have plenty of terms used: "feature maps", "channels", "activations", "activation volumes", "planes", "tensors", etc. This is by far a non-exhaustive list of overlapping terminology, which impedes our understanding.

Bibliography

- [Abd18a] KAMEL ABDELOUAHAB, MAXIME PELCAT, JOCELYN SEROT, and FRANÇOIS BERRY: *Accelerating CNN inference on FPGAs: A Survey*. 2018 (cit. on pp. 80, 146).
- [Abd18b] KAMEL ABDELOUAHAB, MAXIME PELCAT, JOCELYN SÉROT, and FRANÇOIS BERRY: ‘Accelerating CNN inference on FPGAs: A Survey’. In *CoRR* (2018), vol. abs/1806.01683 (cit. on p. 147).
- [Abd17] KAMEL ABDELOUAHAB, MAXIME PELCAT, J. SX00E9ROT, CÉDRIC BOURRASSET, and F. BERRY: ‘Tactics to Directly Map CNN Graphs on Embedded FPGAs’. In *IEEE Embedded Systems Letters* (2017), vol. 9: pp. 113–116 (cit. on p. 137).
- [Bas15] SAIKAT BASU, SANGRAM GANGULY, SUPRATIK MUKHOPADHYAY, ROBERT DIBIANO, MANOHAR KARKI, and RAMAKRISHNA R. NEMANI: ‘DeepSat - A Learning framework for Satellite Imagery’. In (2015), vol. (cit. on pp. 85, 94, 145).
- [Bera] BERKELEY: *Why Caffe?* URL: <http://caffe.berkeleyvision.org/> (visited on) (cit. on p. 83).
- [Berb] BERTENDSP: *Whitepaper: GPU vs FPGA Performance Comparison*. URL: http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf (cit. on pp. 73, 74, 80).
- [Che16a] YU-HSIN CHEN, JOEL EMER, and VIVIENNE SZE: ‘Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks’. In *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016: pp. 367–379 (cit. on p. 89).
- [Che16b] YU-HSIN CHEN, JOEL EMER, and VIVIENNE SZE: ‘Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks’. In *SIGARCH Comput. Archit. News* (June 2016), vol. 44(3): pp. 367–379 (cit. on p. 101).

- [Cou14] MATTHIEU COURBARIAUX, YOSHUA BENGIO, and JEAN-PIERRE DAVID: ‘Low precision arithmetic for deep learning’. In *CoRR* (2014), vol. abs/1412.7024 (cit. on p. 95).
- [Dan18] DIMITRIOS DANOPOULOS: ‘Acceleration of Image Recognition on Caffe framework using FPGAs’. 2018 (cit. on p. 142).
- [Dum16] VINCENT DUMOULIN and FRANCESCO VISIN: ‘A guide to convolution arithmetic for deep learning’. In *CoRR* (2016), vol. abs/1603.07285 (cit. on p. 65).
- [Goo16] IAN GOODFELLOW, YOSHUA BENGIO, and AARON COURVILLE: *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016 (cit. on pp. 53, 55, 56, 62).
- [Gsc16] DAVID GSCHWEND: ‘ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network’. 2016 (cit. on pp. 68, 71, 88).
- [Guo17] KAIYUAN GUO, SHULIN ZENG, JINCHENG YU, YU WANG, and HUAZHONG YANG: *A Survey of FPGA Based Neural Network Accelerator*. 2017 (cit. on pp. 80, 146).
- [Gys16] PHILIPP GYSEL, MOHAMMAD MOTAMED, and SOHEIL GHIASI: ‘Hardware-oriented Approximation of Convolutional Neural Networks’. In *CoRR* (2016), vol. abs/1604.03168 (cit. on pp. 95, 147).
- [Han15] SONG HAN, HUIZI MAO, and WILLIAM J. DALLY: ‘Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding’. In *CoRR* (2015), vol. abs/1510.00149 (cit. on p. 147).
- [Hay09] SIMON S. HAYKIN: *Neural networks and learning machines*. Third. Pearson Education, 2009 (cit. on p. 58).
- [Jia14] YANGQING JIA, EVAN SELHAMER, JEFF DONAHUE, SERGEY KARAYEV, JONATHAN LONG, ROSS GIRSHICK, SERGIO GUADARRAMA, and TREVOR DARRELL: ‘Caffe: Convolutional Architecture for Fast Feature Embedding’. In *arXiv preprint arXiv:1408.5093* (2014), vol. (cit. on p. 83).
- [Kae12] HUBERT KAESLIN: *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 2012 (cit. on p. 73).
- [Kara] ANDREJ KARPATY: *CS231n: Convolutional Neural Networks for Visual Recognition - Image Classification*. URL: <http://cs231n.github.io/classification/> (visited on) (cit. on p. 56).
- [Karb] ANDREJ KARPATY: *CS231n: Convolutional Neural Networks for Visual Recognition - Neural Networks*. URL: <http://cs231n.github.io/neural-networks-1/> (cit. on pp. 58, 59, 66, 67).

- [Karc] ANDREJ KARPATHY: *CS231n: Convolutional Neural Networks for Visual Recognition - Normalization Layer*. URL: <http://cs231n.github.io/convolutional-networks/#norm> (visited on) (cit. on pp. 68, 89).
- [Kri12] ALEX KRIZHEVSKY, ILYA SUTSKEVER, and GEOFFREY E HINTON: ‘ImageNet Classification with Deep Convolutional Neural Networks’. In (2012), vol. (cit. on p. 89).
- [Lec89] YANN LECUN: ‘Generalization and network design strategies’. English (US). In. *Connectionism in perspective*. Ed. by R. PFEIFER, Z. SCHRETER, F. FOGELMAN, and L. STEELS. Elsevier, 1989 (cit. on p. 69).
- [Lei] RICHARD LEIBRANDT: *Commercial Applications of Neural Networks*. URL: <https://www.nst.ei.tum.de/fileadmin/w00bqs/www/publications/as/2012SS-HS-CommercialApplicationsOfNeuralNetworks.pdf> (cit. on p. 45).
- [Pap16] M. PAPADOMANOLAKI, M. VAKALOPOULOU, S. ZAGORUYKO, and K. KARANTZALOS: ‘Benchmarking Deep Learning Frameworks for the Classification of Very High Resolution Satellite Multispectral Data’. In *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* (2016), vol. (cit. on pp. 88, 94).
- [Qiu16] JIANTAO QIU, JIE WANG, SONG YAO, KAIYUAN GUO, BOXUN LI, ERJIN ZHOU, JINCHENG YU, TIANQI TANG, NINGYI XU, SEN SONG, YU WANG, and HUAZHONG YANG: ‘Going Deeper with Embedded FPGA Platform for Convolutional Neural Network’. In *FPGA*. 2016 (cit. on p. 101).
- [Rus15] OLGA RUSSAKOVSKY, JIA DENG, HAO SU, JONATHAN KRAUSE, SANJEEV SATHEESH, SEAN MA, ZHIHENG HUANG, ANDREJ KARPATHY, ADITYA KHOSLA, MICHAEL BERNSTEIN, ALEXANDER C. BERG, and LI FEI-FEI: ‘ImageNet Large Scale Visual Recognition Challenge’. In *International Journal of Computer Vision (IJCV)* (2015), vol. 115(3): pp. 211–252 (cit. on p. 69).
- [Sho93] RICHARD G. SHOUP: *Parameterized Convolution Filtering in a Field Programmable Gate Array Interval*. Tech. rep. 1993 (cit. on p. 105).
- [Sti16] GREG STITT, ERIC SCHWARTZ, and PATRICK COOKE: ‘A Parallel Sliding-Window Generator for High-Performance Digital-Signal Processing on FPGAs’. In *ACM Trans. Reconfigurable Technol. Syst.* (May 2016), vol. 9(3): 23:1–23:22 (cit. on p. 107).
- [Vak15] MARIA VAKALOPOULOU, KONSTANTINOS KARANTZALOS, NIKOS KOMODAKIS, and NIKOS PARAGIOS: ‘Building detection in very high resolution multispectral data with deep learning features’. In *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)* (2015), vol.: pp. 1873–1876 (cit. on p. 94).

- [Ven17] STYLIANOS I. VENIERIS and CHRISTOS-SAVVAS BOUGANIS: ‘Latency-driven design for FPGA-based convolutional neural networks’. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (2017), vol.: pp. 1–8 (cit. on pp. 137, 142).
- [Ven18] STYLIANOS I. VENIERIS, ALEXANDROS KOURIS, and CHRISTOS-SAVVAS BOUGANIS: ‘Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions’. In *ACM Comput. Surv.* (2018), vol. 51: 56:1–56:39 (cit. on p. 148).
- [Wid94] BERNARD WIDROW, DAVID E. RUMELHART, and MICHAEL A. LEHR: ‘Neural Networks: Applications in Industry, Business and Science’. In *Commun. ACM* (Mar. 1994), vol. 37(3): pp. 93–105 (cit. on p. 45).
- [Woo17] R. WOODS, J. MCALLISTER, G. LIGHTBODY, and Y. YI: *FPGA-based Implementation of Signal Processing Systems*. Wiley, 2017 (cit. on p. 73).
- [Xyg17] ATHANASIOS XYGKIS: ‘Implementation of Convolutional Neural Networks on Embedded Architectures’. 2017 (cit. on p. 137).
- [Yan98] LECUN YANN and BENGIO YOSHUA: ‘The Handbook of Brain Theory and Neural Networks’. In. Ed. by MICHAEL A. ARBIB. Cambridge, MA, USA: MIT Press, 1998. Chap. Convolutional Networks for Images, Speech, and Time Series: pp. 255–258 (cit. on p. 62).
- [Zha15] CHEN ZHANG, PENG LI, GUANGYU SUN, YIJIN GUAN, BINGJUN XIAO, and JASON CONG: ‘Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks’. In *FPGA*. 2015 (cit. on p. 101).
- [Zha18] QIANRU ZHANG, MENG ZHANG, TINGHUAN CHEN, ZHIFEI SUN, YUZHE MA, and BEI YU: *Recent Advances in Convolutional Neural Network Acceleration*. 2018 (cit. on pp. 80, 146).

List of Figures

2.1	Example of a System-on-Module that can be used for embedded computing solutions: the Xilinx Zynq MMP, which contains a Zynq-Z7045 SoC. This board costs about 3000€. Lower cost solutions do exist, like the Zybo Zynq-7000, which features a smaller SoC and costs approximately 300€.	46
2.2	A land cover map from the area of Portland, Oregon, in the USA (NASA Earth Observatory).	47
3.1	Machine Learning's relation to other fields	52
3.2	Supervised Learning	53
3.3	Relation of Training Error and Generalization Error to Under-fitting & Over-fitting. Source: "Deep Learning", MIT Press, p.113	54
3.4	Learning Rate in Gradient Descent.	55
3.5	Under-fitting, Fitting and Over-fitting. Models' Capacity. Source:[Goo16]	56
3.6	Performance of traditional Computer Vision and Deep learning algorithms in the Imagenet competition, 2010-2015. Top-5 accuracy illustrated. Source: openpowerfoundation.org	57
3.7	A simplified model of a biological neuron (left) and its mathematical model (right).	58
3.8	Example of a Feed-Forward Neural Network with three hidden layers, three inputs and two outputs. The weights of the network are illustrated with the arrows.	59
3.9	Example of a Convolutional Neural Network: Several layers are placed one after the other and the data flow in one direction (to the right). Network "VGG-16", introduced in 2014. This network achieved 92.7% top-5 accuracy in the ImageNet competition.	61
3.10	Example of an RGB image: A 3D matrix of size 4x4x3. 8 bit for each value (24 bit per pixel).	63
3.11	2D Cross-correlation: a 3x3 kernel slides over a 4x4 input with unit stride and no padding (i.e $H_{in} = 4, K = 3, S = 1, P = 0$). Results to an output of size 2x2. Source: [Dum16]	65

3.12	Convolution of an image with known kernels, from the traditional Computer Vision field: Blur, sharpen, edge detection. In Convolutional Neural Networks the kernels are the variables that will be calibrated in order to approximate a function that maps the input image to a class.	65
3.13	Example of 3D Convolution operation: Three input feature maps, getting convolved with two filters, generating two output feature maps. Padding=1, Stride=1. The marked "pixel" on the output feature map is a sum of all the dot products between the marked area of the input feature maps and the W0 filter's kernels. Source: [Karb]	66
3.14	Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square). Source: [Karb]	67
3.15	The Non-Linear Activation Functions Sigmoid, tanh, ReLU, PReLU and ELU. Source: [Gsc16]	68
3.16	Well-known Convolutional Neural Network Architectures in chronological order.	70
3.17	Visualization of well-known CNN Architectures. Left to right: AlexNet, Network-in-Network, VGG-16, GoogLeNet, ResNet-50, Inception v3, Inception-ResNet-v2, SqueezeNet. Data flows from top to bottom. Convolution layers are presented with brown. Source: [Gsc16]	71
3.18	FPGA Fabric	72
3.19	Flexibility vs Efficiency: CPU, GPU, FPGA, ASIC technologies.	73
3.20	GPU vs FPGA qualitative comparison. Source: [Berb]	74
3.21	FPGA mapping flow	75
3.22	Row and Column Relationship between CLBs and Slices. Source: Xilinx's User Guide	77
3.23	Simplified diagram of an FPGA slice. Since the Xilinx Virtex-5 generation introduced in 2006, the slices contain 6-input LUTs.	77
3.24	Internal structure of a single FPGA slice.	79
3.25	Main approaches to accelerate CNN inference on FPGAs.	81
4.1	Sample images from the SAT-6 dataset. Source: [Bas15]	85
4.2	Original "Cifar-10 Full" CNN Model structure	86
4.3	Half zero-padding the input plane. Ceiling pooling layer's output size.	90
4.4	The "Modified Cifar-10 Full" CNN model: based on the "Cifar-10 Full".	91
4.5	Number of Operations & Weights per layer	92
4.6	Training loss and accuracy of the "Modified Cifar-10 Full" CNN model, for the SAT-6 dataset after 4 epochs of training.	94

4.7	Fixed point arithmetic is adequate for neural network computation: Comparative presentation of the achieved accuracy between the full precision and a quantized version of various networks. These are fine-tuned networks with dynamic fixed point parameters & outputs. The numbers in parentheses indicate accuracy without fine-tuning. [Gys16] .	95
4.8	Examples of dynamic fixed point numbers. Note that the fractional length may be negative or greater than the word length.	96
4.9	"Modified Cifar-10 Full" CNN model's Weights after quantization: Per layer percentages of how many weights are equal to zero, one, or a power-of-two number.	99
5.1	A generic version of the unified Layer's structure. Performs the zero-padding of its input, the truncation of its inputs & outputs, the convolution, the ReLU and the max pooling operations.	104
5.2	The 2D Convolution Engine. Illustrated for a 5x5 Kernel.	106
5.3	The Window-Gen component. Eventually each fifo i will contain the partial results generated by processing the $M_{parallel}^i$ parallel channels, $i = 1 \dots \frac{M_{total}}{M_{parallel}}$	108
5.4	Pooling Window	109
5.5	Comparator Tree	109
5.7	Simplified illustration of adding two numbers with different floating lengths.	112
5.8	Example of connecting two Layers. $N_{parallel}^L = 1$ and $M_{parallel}^{L+1} = 2$. $N_{total}^L = M_{total}^{L+1} = 8$. Also $N_{parallel}^{L+1} = 1$ and $N_{total}^{L+1} = 2$. Layer L creates one output channel each time, while Layer $L + 1$ reads two channels at a time. For that reason Layer L alternates its writing between RAMs A&B. Because the $L + 1$ Layer has to create two output channels, but only generates one of them at a time, it will re-read the input data twice, one for each filter. Moreover, double buffers in Ping-Pong style, to pipeline the processing of a stream of images: At one point in time Layer L created the output of Img1 (ping) and then continues with generating the output of Img2 (pong). Layer $L + 1$ reads the output of Img1, while the output of Img2 is generated. When Layer $L + 1$ proceeds to reading the output of Img2, Layer L will continue in generating the output of Img3, etc. Note that the two time axis use a different scale and their relative position is unrelated. Also stalling is not illustrated.	120
5.9	Layer L creates three output channels parallelly, while Layer $L + 1$ reads four input channels parallelly. The total amount of channels in between initially was equal to five. In order to preserve the symmetry, dummy channels are used, which increase the amount of channels in between to $N_{total}^L = ceil(\frac{5}{3}) \cdot 3 = 6$ and $M_{total}^{L+1} = ceil(\frac{5}{4}) \cdot 4 = 8$. Mind that the two time axis use a different scale and their relative position is unrelated. . .	121

- 5.10 Device's resource utilization, for different $M_{parallel} \times N_{parallel}$ configurations of the 2nd Layer, while keeping the number of Convolution Engines within the Layer constant. DSPs are not used in this example. Increasing the number of parallel inputs in the Layer is preferable over increasing the number of parallel outputs. 124
- 5.11 Results from exploring the design space for three different number of total Convolution Engines in the design, when $N_{parallel}^L \leq M_{parallel}^L$ for all the Layers, $M_{parallel}^{L=1} \in [1, 3]$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in [1, 32]$. Total Convolution Engines = $X + Y + Z$ and $X = x_1 \cdot x_2, Y = y_1 \cdot y_2, Z = z_1 \cdot z_2$. With black circles denoted the configurations in which $M_{parallel}$ and $N_{parallel}$ are divisors of the initial M_{total}, N_{total} , i.e $M_{parallel}^{L=1} \in \{1, 3\}$ and $N_{parallel}^{L=1}, M_{parallel}^{L=2}, N_{parallel}^{L=2}, M_{parallel}^{L=3}, N_{parallel}^{L=3} \in \{1, 2, 4, 8, 16, 32\}$. This proves that it is legitimate to not restrict the parallel inputs and outputs in such a way, as it may lead to sub-optimal solutions. 130
- 5.12 Results of extending the Design Space Exploration to other devices of the Zynq-7000 SoC family. The achieved throughput (Img/sec) and the amount of LUTs needed for each configuration is illustrated. The smallest device possible in which the "Modified Cifar-10 Full" CNN can be implemented is the Z-7015, due to the on-chip memory requirements. To compute all the convolutions of this particular CNN parallelly, a total of $3 \times 32 + 32 \times 32 + 32 \times 64 = 3168$ Convolution Engines would be needed and a huge amount of Block RAMs. 131
- 5.13 Characteristics of the Programmable Logic on the Xilinx Zynq Z-7020 SoC. 133
- 5.14 The configuration of the deliverable design's Layers. Results obtained through synthesis, implementation & simulation with Xilinx's Vivado Design Suite, for the Zynq Z-7020 SoC. 133
- 5.15 The properties of the Deliverable Design on the Zynq Z-7020. 134
- 5.16 Execution time of the "Modified Cifar-10 Full" CNN on the proposed configuration for Zynq Z-7020, in relation to the size of the input image. Zynq Z-7020 can handle images of up to $\sim 50 \times 50$ pixels. 135
- 5.17 The amount of FPGA resources needed in relation to the word-length of the data. Design configuration: $[3 \times 2, 16 \times 1, 8 \times 2]$. Both the amount of registers and Block RAM needed is within the resources of the Z-7020 for all the different word-length scenarios presented. However, the LUTs needed quickly exceed the available resources of Zynq Z-7020. The proposed word-length is to use 4-bits for the activations and 8-bits for the weights of the "Modified Cifar-10 Full" CNN. 136
- 5.18 Comparison of different implementations, executing the inference of the "Cifar-10" CNN. 139

5.19	Comparison of different implementations executing the inference of the "Cifar-10" CNN. FpgaConvNet and Haddoc2 implemented on Xilinx Zynq Z-7045 device. This thesis implemented on the smaller Zynq Z-7020 device.	140
5.20	Thesis design vs designs generated from the FpgaConvNet and the Haddoc2 frameworks, when targeting the same FPGA platform.	141

List of Tables

4.1 Comparison of Different CNN Topologies for Image Classification on ImageNet dataset. Source: [Gsc16]. The "Cifar-10 Full" model is included with its input image dimensions altered to 224x224, to get a better feeling of the network's size compared to the other networks.	88
4.2 "Modified Cifar-10 Full" CNN: Number of Operations & Network's Size	92
4.3 Accuracy rates for SAT-4 & SAT-6 datasets using different learning frameworks [Pap16].	94
4.4 "Modified Cifar-10 Full" CNN's accuracy analysis. Applying dynamic fixed point representation to one category at a time, while the rest remain in full 32-bit precision.	97
4.5 Different approximation scenarios applied on the "Modified Cifar-10 Full" network. Scenarios A-D use dynamic fixed point approximation. Scenario E forces weights to be power-of-two and applies dynamic fixed point to the activations.	97
4.6 Exact configuration of the chosen approximation. It corresponds to the scenario C in table 4.5, achieving 94.89% accuracy. Bw stands for "bit-width" and FL for "floating-length". Also note that the Bias follows the approximation of the Weights at each layer.	98
5.1 Memory Organization of 32 filters, with 3 channels each, for a 3x2 grid of Convolution Engines ($M_{par \times N_{par}}$).	114

