ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

FPGA-oriented deep learning for earth observation image classification

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ασπασία Ε. Σταυριανού

Επιβλέπων : Δημήτριος Ι. Σούντρης

Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2019

(this page is left intentionally blank)

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

FPGA-oriented deep learning for earth observation image classification

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ασπασία Ε. Σταυριανού

Επιβλέπων : Δημήτριος Ι. Σούντρης

Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Νοεμβρίου 2019.

.....	.....	.....
Δημήτριος Ι. Σούντρης	Παναγιώτης Δ.Τσανάκας	Γεώργιος Ι. Γκούμας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής

Αθήνα, Νοέμβριος 2019

……..

Ασπασία Ε. Σταυριανού
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Καθημερινά οι δορυφόροι συγκεντρώνουν μεγάλο όγκο πληροφοριών καθιστώντας αδύνατη την οργάνωση και την μελέτη τους από τον άνθρωπο. Για αυτό τον λόγο τεράστιο ερευνητικό ενδιαφέρον έχει στραφεί στον κλάδο της μηχανικής μάθησης και ειδικότερα στα συνελικτικά νευρωνικά δίκτυα. Ωστόσο, πολλές φορές τα παραδοσιακά υπολογιστικά συστήματα δεν συμβαδίζουν με τις υψηλές υπολογιστικές απαιτήσεις, με αποτέλεσμα να στρεφόμαστε σε επεξεργαστές ειδικού σκοπού όπως είναι ο επεξεργαστής FPGA.

Στην συγκεκριμένη διπλωματική εργασία σχεδιάστηκαν νευρωνικά δίκτυα με την βοήθεια του keras στο tensorflow με στόχο την υλοποίησή τους σε πλατφόρμα FPGA. Χρησιμοποιήθηκαν και εκπαιδεύτηκαν δεδομένα από δύο διαγωνισμούς της kaggle.Ο πρώτος διαγωνισμός λέγεται “Ships in Satellite Imagery” και αποσκοπεί στην αναγνώριση πλοίων από δορυφορικές εικόνες [4]. Ο δεύτερος διαγωνισμός λέγεται “Statoil iceberg classifier challenge” και αποσκοπεί στην κατηγοριοποίηση των εικόνων σε αυτές που περιέχουν παγόβουνα και σε αυτές που περιέχουν πλοία [3]. Για την εκπαίδευση των μοντέλων έγινε χρήση πλήρως συνδεδεμένων νευρωνικών δικτύων και συνελικτικών νευρωνικών δικτύων.

Ταυτόχρονα για τα δεδομένα αναγνώρισης πλοίων που ήταν περισσότερα, μεγάλο ενδιαφέρον δόθηκε στην βελτιστοποίηση του συνελικτικού μοντέλου ως προς την μνήμη. Αυτό επιτεύχθηκε εφαρμόζοντας δύο τρόπους. Ο πρώτος ήταν προσθέτοντας ένα επιπλέον συνελικτικό επίπεδο μειώνοντας έτσι τον αριθμό των παραμέτρων προς εκπαίδευση κατά 10 φορές. Ο δεύτερος ήταν η μετατροπή του τύπου των παραμέτρων από απλής ακρίβειας (float32) σε μισής ακρίβειας (float16) που έφερε επιπλέον μείωση της μνήμη. Στην συνέχεια αναλύθηκαν τα μοντέλα απλής και μισής ακρίβειας όσον αφορά τον χρόνο που απαιτεί κάθε λειτουργία κατά την πρόβλεψη εικόνων.

Στο δεύτερο κομμάτι της διπλωματικής, δοκιμάστηκαν τρία εργαλεία για μετατροπή του μοντέλου σε μορφή συμβατή για την εφαρμογή σε FPGA πλατφόρμα. Τα δύο πρώτα ανοικτού κώδικα εργαλεία ήταν το LeFlow που παράγει κώδικα verilog [22] και το hls4ml που παράγει κώδικα hls [23]. Το τρίτο εμπορικό εργαλείο ήταν το ML-Suite της Xilinx. Τελικά, μεγαλύτερη έμφαση δόθηκε στο εργαλείο hls4ml με το οποίο καταφέραμε να εφαρμόσουμε το μοντέλο στην πλακέτα ZYNQ-7 ZC702. Λόγω της φύσης του εργαλείου, σε πολύ μικρό χρόνο ανάπτυξης (περίπου μισό μήνα), πετύχαμε να υλοποιήσουμε το νευρωνικό δίκτυο χρησιμοποιώντας αποκλειστικά την μνήμη της προγραμματιζόμενης λογικής και με τελικό throughput 35 Images/sec.

Λέξεις κλειδιά

Νευρωνικό δίκτυο, συνελικτικό δίκτυο, πλήρως συνδεδεμένο δίκτυο , βαθιά μάθηση, βελτιστοποίηση μνήμης, πλατφόρμα FPGA.

Abstract

Satellites accumulate a large amount of information every day making it impossible for them to be organized and studied by humans. For this reason tremendous research interest has shifted to the field of machine learning and in particular to neural networks. However, many traditional computing systems do not comply with high computing requirements, so we can turn to special purpose processors such as the FPGA processor.

In this thesis, were designed neural networks with keras in tensorflow with the aim of implementing them on the FPGA platform. Were used and trained data from two kaggle competitions. The first competition is called "Ships in Satellite Imagery" and aims to identify ships from satellite images [4]. The second competition is called the “Statoil iceberg classifier challenge” and aims to determine if an image contains iceberg or ship [3]. The models that used to train were fully connected neural networks and convolutional neural networks.

At the same time for the ship classification data, great interest was given optimizing the memory of convolutional model. This was achieved in two ways. The first was to add an extra convolutional layer, thereby reducing the number of trainable parameters by 10 times. The second was to convert the parameter type from simple precision (float32) to half precision (float16) which further reduced the model's memory. Subsequently, simple and half-precision models were time profiled during image prediction.

In the second part of this thesis, three tools were tested to convert the model into code compatible for application to an FPGA processor. Two open source tools were first tested, LeFlow [22] and hls4ml [23] and then the commercial tool of Xilinx Machine Learning Suite [22, 23, 27]. The LeFlow tool generates verilog code [22] and the hls4ml tool generates hls code [23]. Finally, more emphasis was placed on the hls4ml tool with which we were able to apply the model to the ZYNQ-7 ZC702 board. Due to the nature of the tool, in a very short development time (about half a month), we were able to implement the neural network using only programmable logic memory and with a final throughput of 35 Images/sec.

Keywords

Neural network, convolutional neural network, fully connected neural network, deep learning, memory optimization, FPGA, ZYNG-7 ZC702, time profiling, earth observation, Keras, TensorFlow, hls4ml.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον επιβλέποντα καθηγητή κ. Δημήτριο Σούντρη για τις γνώσεις που μου μετέφερε όλα αυτά τα χρόνια των σπουδών μου, αλλά και για την ευκαιρία που μου έδωσε να διεκπεραιώσω την διπλωματική μου εργασία στο συγκεκριμένο εργαστήριο και να συνεργαστώ με αξιόλογους ερευνητές.

Επίσης, θα ήθελα να ευχαριστήσω τον μεταδιδακτορικό ερευνητή κ. Λεντάρη και τον υποψήφιο διδάκτορα κ. Δανόπουλο για την στενή συνεργασία και την καθοδήγηση καθ' όλη την διάρκεια της διπλωματικής μου εργασίας.

Τέλος, δεν θα μπορούσα να μην ευχαριστήσω την οικογένειά μου και τους φίλους μου για την υπομονή και την υποστήριξή τους όλα αυτά τα χρόνια.

Πίνακας περιεχομένων

Κατηγοριοποίηση δορυφορικών εικόνων με FPGA-προσανατολισμένη βαθιά μάθηση

1 Εισαγωγικές έννοιες

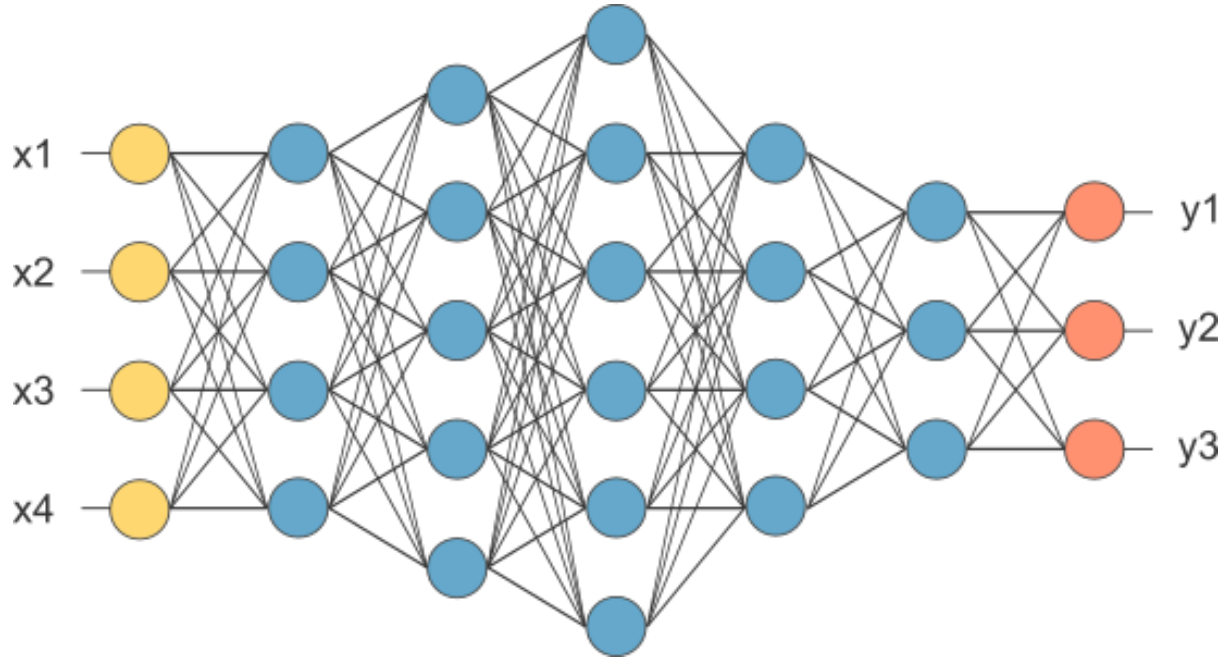
Καθημερινά οι δορυφόροι καταγράφουν τεράστιο όγκο πληροφοριών. Οι δορυφορικές εικόνες μπορούν να παρέχουν χρήσιμες πληροφορίες για διάφορους τομείς της αγοράς. Ειδικότερα, μπορούν να χρησιμοποιηθούν στην γεωργία, στην οικονομία αλλά και για την προστασία με την έγκαιρη ανακάλυψη φωτιάς, πετρελαιοκηλίδας, παγόβουνων κτλ. Προφανώς, είναι τέτοιος ο όγκος τους που καθιστούν αδύνατη την οργάνωση και την μελέτη τους από τον άνθρωπο. Έτσι, δημιουργήθηκε η ανάγκη της μηχανικής μάθησης και της ανάπτυξης νευρωνικών δικτύων για την γρήγορη εξαγωγή πληροφοριών.

Χαρακτηριστικό παράδειγμα αποτελεί μια πρόσφατη έρευνα στον τομέα της γεωργίας και της αγροβιοτεχνίας. Το νευρωνικό δίκτυο που δημιούργησαν τους βοήθησε να κάνουν έναν βιώσιμο σχεδιασμό χρήσης γης, μεγιστοποιώντας τον οικονομικό παράγοντα και ταυτόχρονα ελαχιστοποιώντας την εκπομπή CO2 και την υποβάθμιση της γης [7].

Στην συγκεκριμένη διπλωματική εργασία χρησιμοποιήθηκαν και εκπαιδεύτηκαν δεδομένα από δύο διαγωνισμούς της kaggle. Ο πρώτος διαγωνισμός λέγεται **“Ships in Satellite Imagery”** και αποσκοπεί στην αναγνώριση πλοίων από δορυφορικές εικόνες [4]. Ο δεύτερος διαγωνισμός λέγεται **“Statoil iceberg classifier challenge”** και αποσκοπεί στην κατηγοριοποίηση των εικόνων σε αυτές που περιέχουν παγόβουνα και σε αυτές που περιέχουν πλοία [3]. Αυτό που μας ενδιαφέρει είναι να σχεδιάσουμε νευρωνικά δίκτυα με σκοπό την μεγιστοποίηση της ακρίβειας και ταυτόχρονα την βελτιστοποίηση των μοντέλων ως προς την μνήμη. Για την εκπαίδευση των μοντέλων κάναμε χρήση πλήρως συνδεδεμένων νευρωνικών δικτύων και συνελικτικών νευρωνικών δικτύων. Τις βασικές τους έννοιες θα προσπαθήσουμε να τις καλύψουμε εν συντομία στο συγκεκριμένο κεφάλαιο.

1.1 Πλήρως συνδεδεμένα νευρωνικά δίκτυα

Ένα πλήρως συνδεδεμένο νευρωνικό δίκτυο αποτελείται από μια σειρά από πλήρως συνδεδεμένα επίπεδα. Ένα πλήρως συνδεδεμένο επίπεδο είναι μια συνάρτηση από το Rm στο Rn [8]. Κάθε έξοδος εξαρτάται από κάθε είσοδο όπως φαίνεται στην παρακάτω εικόνα.



Εικόνα 1 Πλήρως συνδεδεμένο νευρωνικό δίκτυο [39]

Έστω x ∈ Rm και αντιπροσωπεύει την είσοδο σε ένα πλήρως συνδεδεμένο επίπεδο. Έστω ότι yi ∈ Rn είναι η i-η έξοδος του επόμενου επιπέδου. Τότε yi ∈ Rn υπολογίζεται ως εξής:

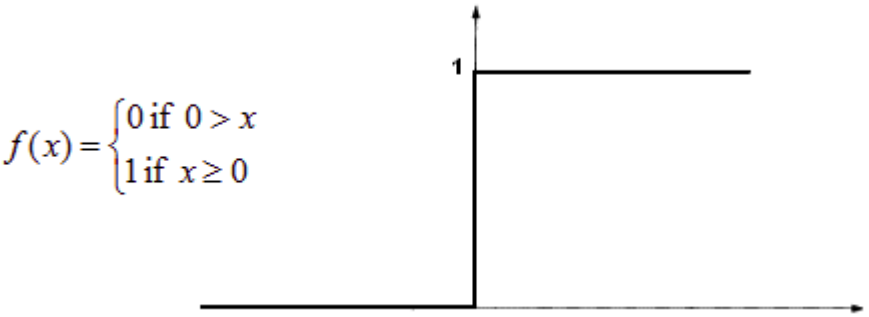
$y_i = \sigma (w_1 x_1 + \dots + w_m x_m) + c$, όπου η συνάρτηση σ ονομάζεται συνάρτηση ενεργοποίησης και την ορίζουμε στον σχεδιασμό του δικτύου, w_i τα βάρη που καταλήγουν στον νευρώνα y_i και c μία σταθερά.

1.2 Συνάρτηση ενεργοποίησης (Activation function)

Η τιμή κάθε νευρώνα μπορεί να κυμαίνεται από το -∞ έως το +∞. Επειδή ο νευρώνας δεν μπορεί να γνωρίζει αν θα πρέπει να ενεργοποιηθεί ή όχι, δημιουργήθηκαν οι συναρτήσεις ενεργοποίησης. Από το όνομα καταλαβαίνουμε ότι όταν η προκύπτουσα τιμή του νευρώνα ξεπεράσει κάποια όρια η συνάρτηση ενεργοποίησης απενεργοποιεί τον συγκεκριμένο νευρώνα [9]. Κάποιες από τις βασικές συναρτήσεις ενεργοποίησης που χρησιμοποιούνται στα νευρωνικά δίκτυα αναφέρονται στην συνέχεια.

Βηματική (Step function)

Η βηματική συνάρτηση ορίζεται ως εξής: η έξοδος είναι 1 (ενεργοποιημένος) όταν η τιμή είναι > 0 (κατώφλι) και η έξοδος είναι 0 (μη ενεργοποιημένος) διαφορετικά. Γραφικά φαίνεται στην παρακάτω εικόνα.



Εικόνα 2 Βηματική συνάρτηση ενεργοποίησης [41]

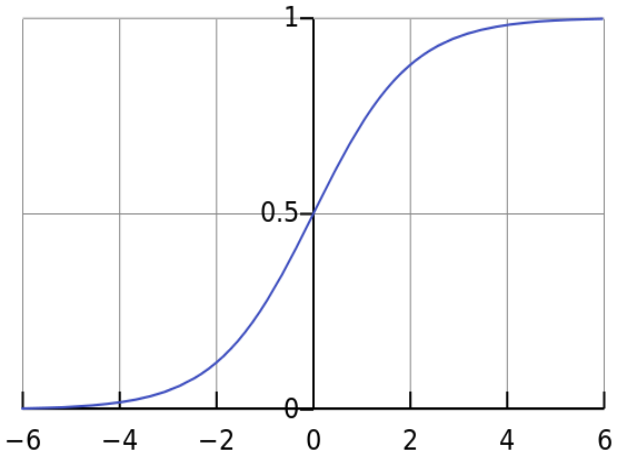
Η συγκεκριμένη συνάρτηση μπορεί να χρησιμοποιηθεί σε δυαδικά προβλήματα στα οποία η απάντηση είναι ένα «ναι» ή ένα «όχι». Βέβαια, θα ήταν καλύτερο εάν η ενεργοποίηση δεν ήταν δυαδική και αντίθετα για κάθε νευρώνα λέγαμε ότι "ενεργοποιήθηκε κατά 50%" ή "ενεργοποιήθηκε κατά 20%" και ούτω καθεξής. Και στη συνέχεια, εάν ενεργοποιούνταν πολλοί νευρώνες να μπορούσαμε να επιλέξουμε τον νευρώνα με την "υψηλότερη ενεργοποίηση".

Γραμμική (Linear function)

Η γραμμική συνάρτηση είναι της μορφής y =c**x** , δηλαδή η ενεργοποίηση θα είναι ανάλογη προς την είσοδο. Το πρόβλημα με αυτή την συνάρτηση είναι ότι ανεξάρτητα από το πόσα στρώματα έχουμε, αν όλα είναι γραμμικά στη φύση, η τελική συνάρτηση ενεργοποίησης του τελευταίου στρώματος δεν είναι τίποτε άλλο παρά μια γραμμική συνάρτηση της εισόδου του πρώτου στρώματος. Αυτό σημαίνει ότι αυτά τα δύο στρώματα (ή N στρώματα) μπορούν να αντικατασταθούν από ένα μόνο στρώμα.

Σιγμοειδής (Sigmoid function)

Η σιγμοειδής συνάρτηση είναι της μορφής $y = (1 + e^{-x})^{-1}$. Η γραφική της παράσταση απεικονίζεται παρακάτω.

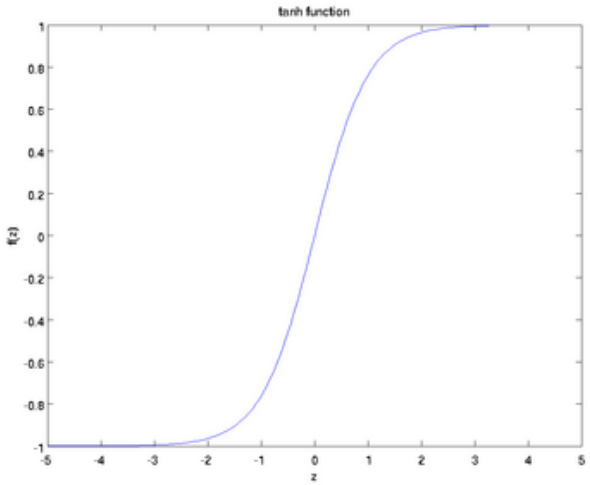


Εικόνα 3 Σιγμοειδής συνάρτηση ενεργοποίησης [40]

Η συγκεκριμένη συνάρτηση είναι μη γραμμική και από το 0 στο 1 έχει απότομη κλίση με αποτέλεσμα οι τελικές τιμές να είναι πιο κοντά στις ακραίες τιμές δηλαδή στο 0 και στο 1. Μπορούμε να παρατηρήσουμε λοιπόν ότι έχουμε συνδυάσει τα θετικά χαρακτηριστικά της βηματικής και της γραμμικής σε μία ενιαία συνάρτηση.

Υπερβολική εφαπτομένη (Tanh function)

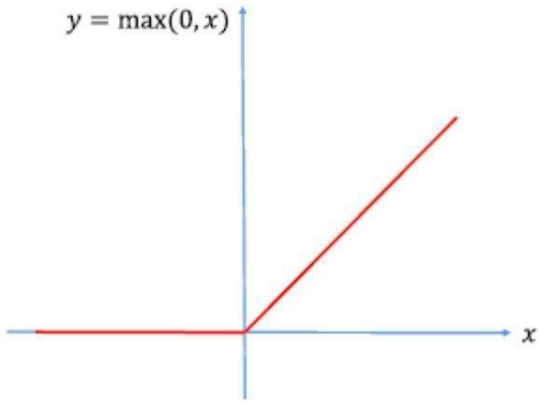
Η υπερβολική εφαπτομένη είναι της μορφής $y=2(1 + e^{-2x})^{-1} - 1$. Παρατηρούμε στην γραφική της παράσταση ότι είναι παρόμοια με την σιγμοειδής με δύο διαφορές. Η πρώτη διαφορά είναι ότι κυμαίνεται στο (-1,1) αντί για το διάστημα (0,1) και η δεύτερη είναι ότι έχει πιο απότομη κλίση.



Εικόνα 4 Συνάρτηση ενεργοποίησης υπερβολικής εφαπτομένης [42]

Συνάρτηση Relu

Η Relu είναι της μορφής $y=\max\{0,\mathbf{x}\}$. Αν και φαίνεται γραφικά ότι στον θετικό άξονα είναι γραμμική, εξ' ορισμού δεν αποτελεί γραμμική συνάρτηση και συνεπώς εφαρμόζοντάς την σε πολλά επίπεδα η έξοδος παραμένει μη γραμμική. Ταυτόχρονα με τον μηδενισμό πολλών νευρώνων το δίκτυό μας γίνεται πιο αραιό και αποδοτικό. Επίσης, η ReLu είναι λιγότερο υπολογιστικά ακριβή από την tanh και την sigmoid επειδή περιλαμβάνει απλούστερες μαθηματικές πράξεις και πρέπει να το λαμβάνουμε υπόψη όταν σχεδιάζουμε βαθιά νευρικά δίκτυα. Το αρνητικό είναι ότι δεν είναι φραγμένη στο $+\infty$, όπως παρατηρούμε και στην παρακάτω γραφική της παράσταση.



Εικόνα 5 Συνάρτηση ενεργοποίησης Relu [43]

1.3 Αλγόριθμος βελτιστοποίησης (Optimizer)

Κάθε φορά που ένα νευρωνικό δίκτυο ολοκληρώνει μία παρτίδα (batch) εκπαίδευσης και δημιουργεί τα αποτελέσματα πρόβλεψης, πρέπει να αποφασίσει πώς να χρησιμοποιήσει τη διαφορά μεταξύ των αποτελεσμάτων που πήρε και των τιμών που ξέρει ότι είναι αληθινά για να προσαρμόσει κατάλληλα τα βάρη στους κόμβους. Ο αλγόριθμος που καθορίζει το βήμα αυτό είναι γνωστός ως ο **αλγόριθμος βελτιστοποίησης** [10]. Στα μοντέλα που δημιουργήσαμε κάνουμε χρήση τριών αλγορίθμων βελτιστοποίησης τον **SGD** (Στοχαστική κλίση ή stochastic gradient descent), τον **RMSprop** (Μέση τετραγωνική διάδοση των ριζών ή Root Mean Square Propagation) και τον **Adam** (Προσαρμοστικός υπολογισμός ροπής ή Adaptive Moment Estimation). Αυτοί οι τρεις αλγόριθμοι αναλύονται παρακάτω.

Αλγόριθμος βελτιστοποίησης SGD

Ο αλγόριθμος βελτιστοποίησης SGD είναι ο "κλασικός" αλγόριθμος βελτιστοποίησης. Στον SGD υπολογίζουμε την κλίση της συνάρτησης απώλειας δικτύου σεβόμενοι κάθε μεμονωμένο βάρος του δικτύου. Κάθε κατεύθυνση προς τα εμπρός μέσω του δικτύου έχει ως αποτέλεσμα μια συγκεκριμένη παραμετροποιημένη συνάρτηση απώλειας και χρησιμοποιούμε κάθε κλίση που έχουμε υπολογίσει για κάθε ένα από τα βάρη, πολλαπλασιασμένο με ένα συγκεκριμένο ρυθμό εκμάθησης (learning rate), για να μετακινήσουμε τα βάρη μας σε οποιαδήποτε κατεύθυνση δείχνει η κλίση του.

Ο SGD είναι ο απλούστερος αλγόριθμος τόσο από εννοιολογική άποψη όσο και από άποψη συμπεριφοράς. Με δεδομένο ένα αρκετά μικρό ρυθμό εκμάθησης, το SGD ακολουθεί απλά την κλίση στην επιφάνεια κόστους. Τα νέα βάρη που παράγονται σε κάθε επανάληψη θα είναι πάντα αυστηρά καλύτερα από τα παλιά της προηγούμενης επανάληψης.

Η απλότητα του SGD τον καθιστά καλή επιλογή για τα ρηχά δίκτυα. Ωστόσο, σημαίνει επίσης ότι ο SGD συγκλίνει σημαντικά πιο αργά από άλλους πιο προηγμένους αλγόριθμους που είναι επίσης διαθέσιμοι στο keras.

Θα πρέπει να αναφέρουμε ότι υπάρχει κι ο SGD με ορμή (momentum) ο οποίος περιορίζει την ταλάντωση σε μία κατεύθυνση. Με αυτό τον τρόπο ο αλγόριθμος μπορεί να συγκλίνει γρηγορότερα στο βέλτιστο σημείο.

Αλγόριθμος βελτιστοποίησης RMSprop

Ο RMSprop είναι παρόμοιος με τον αλγόριθμο SGD με ορμή. Ο βελτιστοποιητής RMSprop περιορίζει τις ταλαντώσεις στην κάθετη κατεύθυνση. Ως εκ τούτου, μπορούμε να αυξήσουμε το ρυθμό μάθησης και ο αλγόριθμός μας θα μπορούσε να κάνει μεγαλύτερα βήματα στην οριζόντια κατεύθυνση συγκλίνοντας ταχύτερα. Η διαφορά μεταξύ του RMSprop και του SGD είναι στον τρόπο με τον οποίο υπολογίζονται οι κλίσεις [11].

Αλγόριθμος βελτιστοποίησης Adam

Ο Adam παρουσιάστηκε από τον Diederik Kingma από το OpenAI και τον Jimmy Ba από το Πανεπιστήμιο του Τορόντο το 2015. Ο αλγόριθμος υπολογίζει έναν εκθετικό κινούμενο μέσο της κλίσης και την τετραγωνική κλίση και οι παράμετροι beta1 και beta2 ελέγχουν τους ρυθμούς αποσύνθεσης αυτών των κινητών μέσων [12]. Οι δύο αυτοί παράμετροι είναι συνήθως κοντά στο 1.

1.4 Συνελικτικά νευρωνικά δίκτυα (Convolutional Neural Network – CNN)

Τα συνελικτικά νευρωνικά δίκτυα είναι μια κλάση νευρωνικών δικτύων που ειδικεύεται στην επεξεργασία δεδομένων που έχουν τοπολογία σαν ένα πλέγμα, όπως μια εικόνα. Μια ψηφιακή εικόνα είναι μια δυαδική αναπαράσταση οπτικών δεδομένων. Περιέχει μια σειρά από εικονοστοιχεία (pixel) διατεταγμένα σε τρόπο παρόμοιο με πλέγμα. Κάθε στοιχείο του πλέγματος περιέχει μία τιμή που δείχνει πόσο φωτεινό και ποιο χρώμα έχει [13].

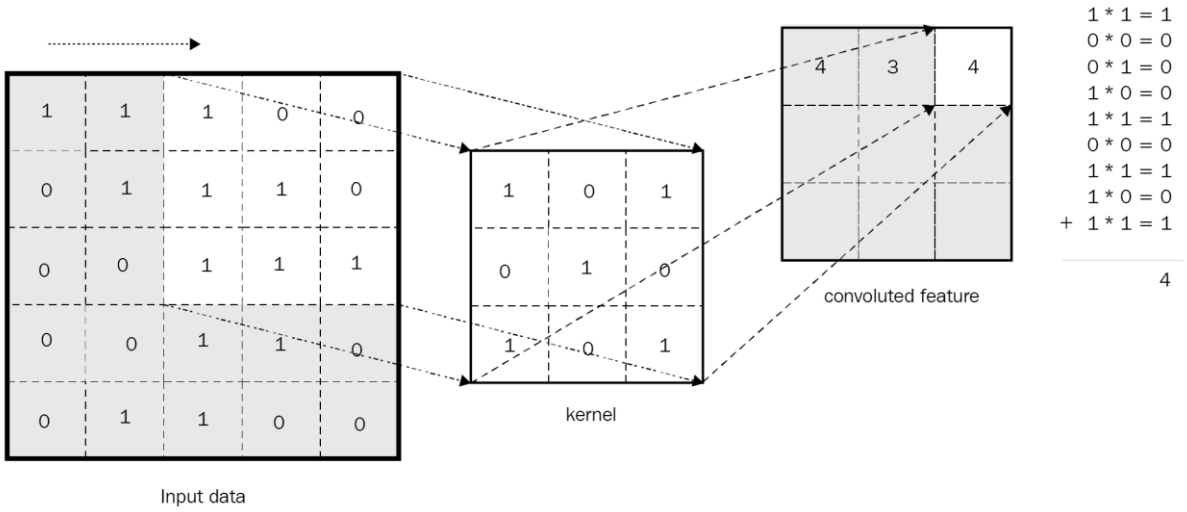
Συνελικτικό επίπεδο (Convolution Layer)

Ο ρόλος του συνελικτικού επιπέδου είναι να μειώσει το μέγεθος της εικόνας σε μια μορφή που είναι ευκολότερη να επεξεργαστεί, χωρίς να χαθούν χαρακτηριστικά που είναι κρίσιμα για την επίτευξη μιας καλής πρόβλεψης. Αυτό είναι σημαντικό όταν σχεδιάζουμε μια αρχιτεκτονική που δεν είναι μόνο καλή στο να μαθαίνει χαρακτηριστικά, αλλά είναι επίσης επεκτάσιμη για μεγάλα σύνολα δεδομένων [14].

Αυτό το επίπεδο εκτελεί πολλαπλασιασμό μεταξύ δύο πινάκων. Ο ένας πίνακας ονομάζεται πυρήνας (kernel) και αντιπροσωπεύει κάποιο συγκεκριμένο χαρακτηριστικό. Ο δεύτερος πίνακας είναι τμήμα της αρχικής εικόνας εισόδου. Ο πυρήνας είναι χωρικά μικρότερος από μια εικόνα, αλλά ίδιος σε βάθος. Αυτό σημαίνει ότι εάν η εικόνα αποτελείται από τρία (RGB) κανάλια, το ύψος και το πλάτος του πυρήνα θα είναι μικρά σε σχέση με την εικόνα, αλλά το βάθος θα εκτείνεται και στα τρία κανάλια.

Ο πυρήνας ολισθαίνει κατά μήκος του ύψους και του πλάτους της εικόνας και παράγει κάθε φορά ένα αποτέλεσμα. Αυτή η διαδικασία παράγει μια δισδιάστατη αναπαράσταση της εικόνας που είναι γνωστή

ως χάρτης ενεργοποίησης (activation map) που δείχνει την απόκριση του πυρήνα σε κάθε υποπεριοχή της εικόνας. Το πόσο θα μετακινηθεί ο πυρήνας ονομάζεται βήμα (stride). Για να γίνει περισσότερο κατανοητό ακολουθεί παράδειγμα στην παρακάτω εικόνα με δεδομένη εικόνα, δεδομένο πυρήνα και μοναδιαίο βήμα.

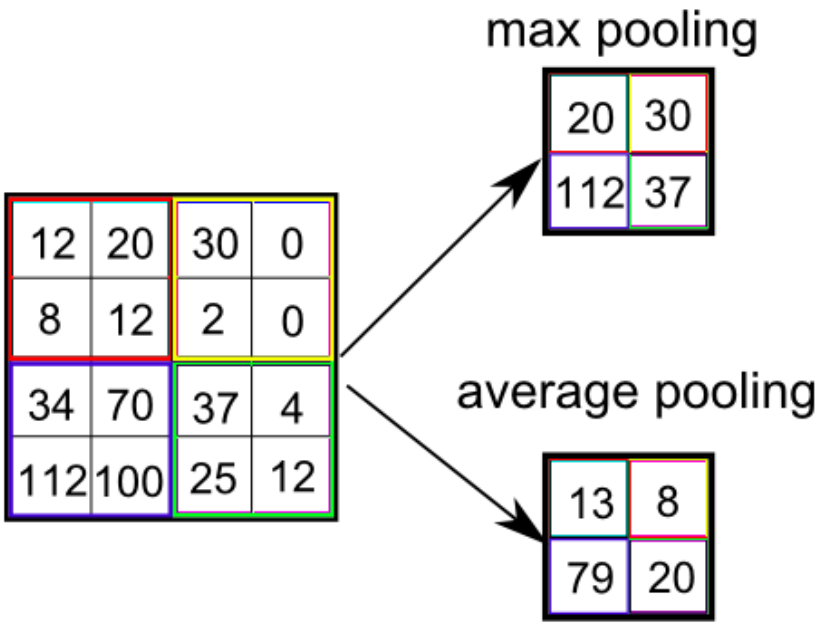


Εικόνα 6 Λειτουργία συνελικτικού επιπέδου [45]

Επίπεδο συγκέντρωσης (Pooling Layer)

Το στρώμα συγκέντρωσης αντικαθιστά την έξοδο του δικτύου σε ένα σύνολο θέσεων με μία τιμή, λαμβάνοντας υπόψη στατιστικά των κοντινών εξόδων. Αυτό βοηθά στη μείωση του χωροταξικού μεγέθους της αναπαράστασης, γεγονός που μειώνει τον απαιτούμενο αριθμό υπολογισμών και βαρών. Η διαδικασία συγκέντρωσης γίνεται μεμονωμένα σε κάθε κομμάτι της αναπαράστασης.

Υπάρχουν διάφορες λειτουργίες συγκέντρωσης όπως ο μέσος όρος της ορθογώνιας γειτονιάς ή ένας σταθμισμένος μέσος όρος με βάση την απόσταση από το κεντρικό εικονοστοιχείο (average pooling). Ωστόσο, η πιο δημοφιλής διαδικασία είναι η μέγιστη συγκέντρωση (max pooling), η οποία κρατάει το μέγιστο στοιχείο κάθε γειτονιάς. Παράδειγμα αυτής της διαδικασίας φαίνεται παρακάτω.



Εικόνα 7 Επίπεδο συγκέντρωσης μέγιστου στοιχείου και μέσου όρου [44]

Θα πρέπει να αναφερθεί ότι η διαδικασία της μέγιστης συγκέντρωσης λειτουργεί επίσης ως καταστολέας θορύβου. Ταυτόχρονα με την μείωση των διαστάσεων, απορρίπτει συνολικά τις θορυβώδεις ενεργοποιήσεις. Από την άλλη πλευρά, η μέση συγκέντρωση απλώς εκτελεί μείωση των διαστάσεων [14].

Επίπεδο Απορρίψεων (Dropout Layer)

Τα βαθιά νευρωνικά δίκτυα με μεγάλο αριθμό παραμέτρων είναι πολύ ισχυρά συστήματα μηχανικής μάθησης. Ωστόσο, η υπερφόρτωση (overfitting) είναι ένα σοβαρό πρόβλημα σε τέτοια δίκτυα. Το μοντέλο μας δηλαδή μαθαίνει τα δεδομένα εκπαίδευσης σε τέτοιο βαθμό που δεν μπορεί να γενικευθεί για άγνωστα δεδομένα [15]. Η απόρριψη είναι μια τεχνική αντιμετώπισης αυτού του προβλήματος. Η βασική ιδέα είναι να απορρίψει τυχαία νευρώνες (μαζί με τις συνδέσεις τους) από το συνολικό δίκτυο κατά τη διάρκεια της εκπαίδευσης. Αυτό εμποδίζει τους νευρώνες να προσαρμοστούν υπερβολικά στα συγκεκριμένα δεδομένα εκπαίδευσης και παράλληλα λαμβάνουμε ένα πιο αραιό δίκτυο [16].

1.5 Αύξηση δεδομένων (Data Augmentation)

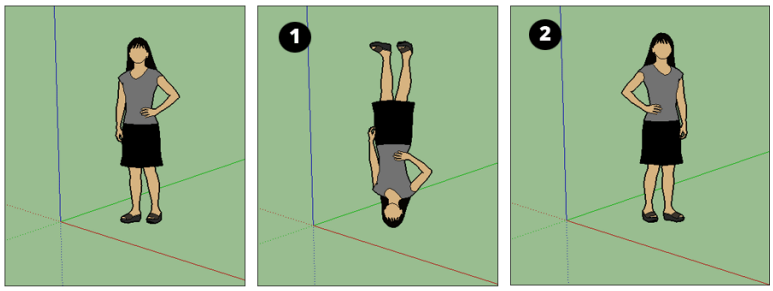
Η αύξηση των δεδομένων (συνήθως εικόνες) είναι μια τεχνική που μπορεί να χρησιμοποιηθεί για την τεχνητή επέκταση του μεγέθους του συνόλου δεδομένων εκπαίδευσης. Αυτό γίνεται με τη δημιουργία τροποποιημένων εκδόσεων των εικόνων στο σύνολο των δεδομένων. Ο εμπλουτισμός των νευρωνικών δικτύων με περισσότερα δεδομένα μπορεί να οδηγήσει σε πιο εξειδικευμένα μοντέλα. Ταυτόχρονα, οι

τεχνικές επαύξησης μπορούν να δημιουργήσουν παραλλαγές των εικόνων βελτιώνοντας την ικανότητα των μοντέλων να γενικεύουν τι έχουν μάθει με νέες εικόνες.

Μια επιλογή είναι να εκτελέσουμε τους μετασχηματισμούς στις εικόνες εκ των προτέρων, αυξάνοντας ουσιαστικά το μέγεθος των συνολικών μας δεδομένων. Η άλλη επιλογή είναι να εκτελέσουμε αυτούς τους μετασχηματισμούς σε μικρές παρτίδες, λίγο πριν τις τροφοδοτήσουμε για εκπαίδευση στο μοντέλο μας [17]. Η πρώτη επιλογή είναι γνωστή ως αύξηση χωρίς σύνδεση (offline augmentation). Αυτή η μέθοδος προτιμάται για σχετικά μικρά σύνολα δεδομένων, καθώς αυξάνει το μέγεθος των δεδομένων με έναν παράγοντα ίσο με τον αριθμό των μετασχηματισμών που πραγματοποιήσαμε (Για παράδειγμα, αναστρέφοντας όλες τις εικόνες μας, το μέγεθος των δεδομένων διπλασιάζεται). Η δεύτερη επιλογή είναι γνωστή ως αύξηση με σύνδεση (online augmentation). Αυτή η μέθοδος προτιμάται για μεγαλύτερα σύνολα δεδομένων, καθώς δεν μπορούμε να αντέξουμε από άποψη μνήμης την εκρηκτική αύξηση των δεδομένων. Για αυτό τον λόγο εκτελούνται οι μετασχηματισμοί ανά παρτίδα, οι οποίες τροφοδοτούνται στην συνέχεια στο μοντέλο. Παρακάτω παρουσιάζονται ορισμένες βασικές αλλά ισχυρές τεχνικές αύξησης που χρησιμοποιούνται ευρέως.

Αντιστροφή (Flip)

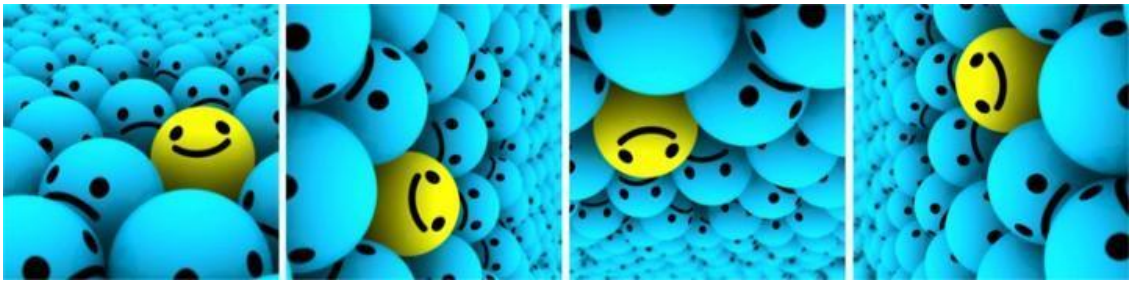
Μπορούμε να αντιστρέψουμε τις εικόνες οριζόντια και κάθετα. Παρακάτω παρατίθενται παραδείγματα για εικόνες που έχουν γυριστεί.



Εικόνα 8 Αντιστροφή κάθετη και οριζόντια [46]

Περιστροφή (Rotation)

Ένα σημαντικό πράγμα που πρέπει να σημειωθεί σχετικά με αυτή τη λειτουργία είναι ότι οι διαστάσεις της εικόνας μπορεί να μην διατηρούνται μετά την περιστροφή. Αν η εικόνα σας είναι τετράγωνη, περιστρέφοντάς την σε ορθές γωνίες θα διατηρηθεί το μέγεθος της εικόνας. Εάν είναι ορθογώνια, η περιστροφή της κατά 180 μοίρες θα διατηρούσε το μέγεθος. Η περιστροφή της εικόνας με μικρότερες γωνίες θα αλλάξει επίσης το τελικό μέγεθος της εικόνας. Σε αυτές τις περιπτώσεις συνήθως ο χώρος πέρα από το όριο της εικόνας θεωρείται ότι είναι το σταθερό 0 σε κάθε σημείο. Επομένως, όταν κάνουμε αυτούς τους μετασχηματισμούς, η μη ορισμένη περιοχή της μετασχηματισμένης εικόνας είναι μαύρη. Παρακάτω παρατίθενται παραδείγματα τετράγωνων εικόνων που περιστρέφονται υπό ορθή γωνία αλλά και μικρότερες γωνίες.



Εικόνα 9 Περιστροφές υπό ορθή γωνία [47]

Εικόνα 10 Περιστροφή με τυχαία γωνία [48]

Κλιμάκωση (Scale)

Η εικόνα μπορεί να μεγεθυνθεί και να σμικρυνθεί. Κατά την μεγέθυνση το τελικό μέγεθος της εικόνας θα είναι μεγαλύτερο από το αρχικό μέγεθος. Τις περισσότερες φορές κόβουμε ένα τμήμα από τη νέα εικόνα, ώστε το τελικό μέγεθος να είναι ίσο με την αρχική εικόνα. Η σμίκρυνση μειώνει το μέγεθος της εικόνας, αναγκάζοντάς μας να κάνουμε υποθέσεις για το τι βρίσκεται πέρα από το όριο. Παρακάτω παρατίθενται παραδείγματα εικόνων που έχουν κλιμακωθεί.

Εικόνα 11 Μεγέθυνση [49]

Εικόνα 12 Σμίκρυνση [48]

Περικοπή (Crop)

Σε αντίθεση με την κλιμάκωση, επιλέγουμε τυχαία ένα τμήμα της αρχικής εικόνα. Στη συνέχεια, αλλάζουμε τις διαστάσεις ώστε να έχει ίδιες με της αρχικής. Αυτή η μέθοδος είναι ευρέως γνωστή ως τυχαία περικοπή.

Ολίσθηση (Translation)

Η ολίσθηση περιλαμβάνει μόνο τη μετακίνηση της εικόνας κατά μήκος της κατεύθυνσης X ή Y (ή και των δύο). Στο παρακάτω παράδειγμα, υποθέτουμε ότι η εικόνα έχει ένα μαύρο φόντο πέρα από τα όριά της και μεταφράζεται κατάλληλα. Αυτή η μέθοδος αύξησης είναι πολύ χρήσιμη καθώς τα περισσότερα αντικείμενα μπορούν να τοποθετηθούν σχεδόν οπουδήποτε στην εικόνα. Αυτό αναγκάζει το νευρωνικό μας δίκτυο να κοιτάει παντού.

Εικόνα 13 Ολίσθηση κατά τον άξονα x και κατά τον άξονα y [50]

Γκαουσιανός θόρυβος (Gaussian Noise)

Σε πολλές περιπτώσεις το νευρωνικό μας δίκτυο προσπαθεί να μάθει χαρακτηριστικά υψηλής συχνότητας (σχέδια που εμφανίζονται πολύ) που μπορεί να μην είναι χρήσιμα. Ο θόρυβος Gauss, ο οποίος έχει μηδενική μέση τιμή, έχει ουσιαστικά δεδομένα σε όλες τις συχνότητες, πράγμα που παραμορφώνει αποτελεσματικά τα χαρακτηριστικά υψηλής συχνότητας. Αυτό σημαίνει επίσης ότι τα στοιχεία χαμηλότερης συχνότητας είναι επίσης παραμορφωμένα, αλλά το νευρωνικό δίκτυο μπορεί να μάθει να βλέπει το παρελθόν. Η προσθήκη ακριβώς της σωστής ποσότητας θορύβου μπορεί να ενισχύσει τη δυνατότητα εκμάθησης χαρακτηριστικών χαμηλότερης συχνότητας. Μία ήπια έκδοση αυτού του τύπου είναι ο λεγόμενος θόρυβος του αλατιού και του πιπεριού, ο οποίος εμφανίζεται ως τυχαία ασπρόμαυρα εικονοστοιχεία που διασκορπίζονται σε όλη την εικόνα. Αυτό είναι παρόμοιο με το αποτέλεσμα που παράγεται από την προσθήκη θορύβου Gauss σε μια εικόνα, αλλά με χαμηλότερο επίπεδο παραμόρφωσης πληροφοριών.

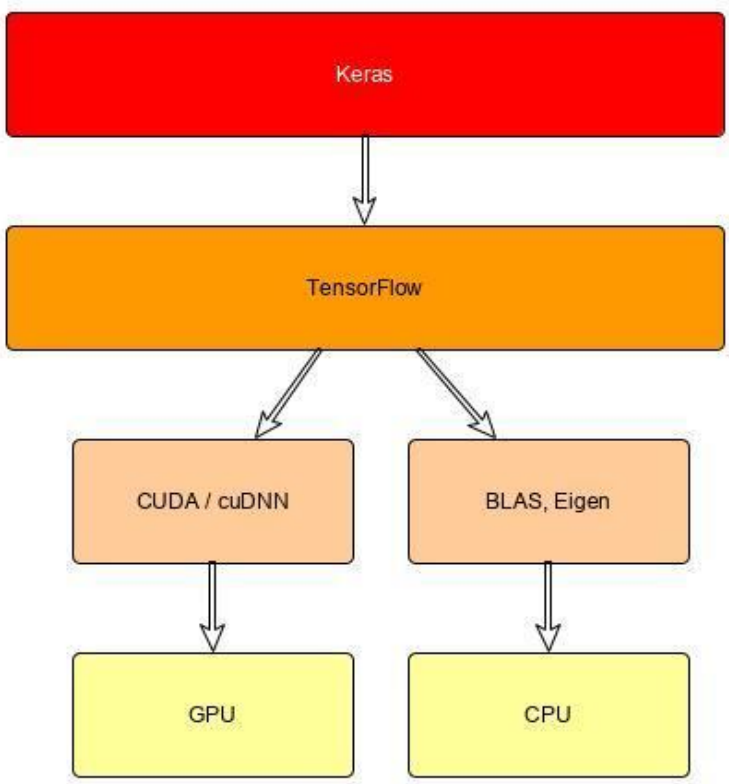
Εικόνα 14 Γκαουσιανός θόρυβος σε εικόνες [51]

1.6 Ανάπτυξη νευρωνικών δικτύων με το Keras framework του Tensorflow

Το Keras είναι εφαρμογή υψηλού επιπέδου νευρωνικών δικτύων, γραμμένο σε Python και ικανό να τρέχει πάνω από το TensorFlow, το CNTK ή το Theano. Στην συγκεκριμένη διπλωματική εργασία τα μοντέλα που αναπτύξαμε σε keras τρέξαν με την υποστήριξη του TensorFlow.

Με απλά λόγια, το TensorFlow είναι μια βιβλιοθήκη ανοιχτού κώδικα για την κατασκευή μοντέλων μηχανικής μάθησης μεγάλης κλίμακας [29]. Είναι κατά πολύ η πιο δημοφιλής βιβλιοθήκη για την κατασκευή μοντέλων βαθιάς μάθησης. Έχει επίσης την ισχυρότερη και μια τεράστια κοινότητα προγραμματιστών, ερευνητών και συντελεστών. Το Tensorflow κατασκευάζει ένα υπολογιστικό γράφημα για κάθε είδους υπολογισμό που γίνεται, από την πρόσθεση δύο αριθμών, μέχρι την κατασκευή ενός περίπλοκου συνελικτικού δικτύου. Μόλις δημιουργηθεί ένα γράφημα, εκτελείται σε μια επονομαζόμενη περίοδο λειτουργίας (session).

Το Keras υποστηρίζει συνελικτικά (CNNs) αλλά και επαναλαμβανόμενα (RNNs) δίκτυα, καθώς και συνδυασμούς αυτών των δύο. Επίσης, πολύ σημαντικό είναι ότι μέσω του TensorFlow μπορεί να τρέξει τόσο σε κοινό επεξεργαστή CPU όσο και σε GPU. Όταν εκτελείται με CPU, το TensorFlow τυλίγεται με μια βιβλιοθήκη χαμηλού επιπέδου για λειτουργίες tensor, που ονομάζεται Eigen. Όταν εκτελείται με GPU, το TensorFlow συνδέεται με μια βιβλιοθήκη με καλά βελτιστοποιημένες λειτουργίες βαθιάς εκμάθησης που ονομάζεται βιβλιοθήκη NVIDIA CUDA Deep Neural Network (cuDNN).



Εικόνα 15 Συνεργασία εφαρμογών και βιβλιοθηκών για να τρέξουμε ένα keras μοντέλο σε CPU ή GPU

Μερικά από τα πλεονεκτήματα του keras είναι η φιλικότητα προς τον χρήστη, η ανεξαρτησία μεταξύ των στοιχείων, η εύκολη επεκτασιμότητα καθώς και η υποστήριξή του από την python τα οποία αναλύονται παρακάτω [28].

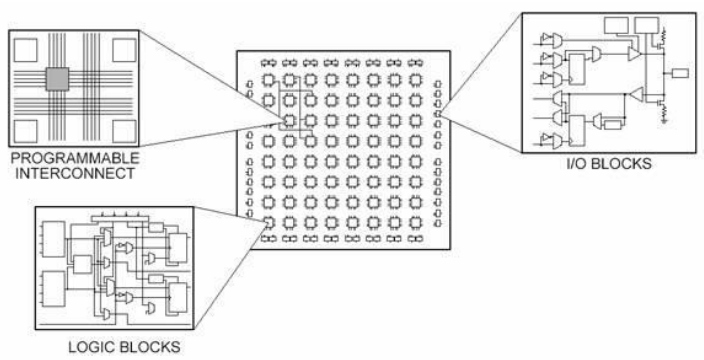
- Φιλικότητα προς τον χρήστη. Το Keras είναι εφαρμογή σχεδιασμένη για ανθρώπους, όχι για μηχανές. Βάζει την εμπειρία του χρήστη μπροστά και στο κέντρο. Το Keras ακολουθεί τις βέλτιστες πρακτικές για τη μείωση του γνωστικού φορτίου: προσφέρει συνεπή & απλή εφαρμογή, ελαχιστοποιεί τον αριθμό των ενεργειών χρήστη που απαιτούνται για κοινές περιπτώσεις χρήσης και παρέχει σαφή και ενεργή ανατροφοδότηση κατά το σφάλμα του χρήστη.
- Ένα μοντέλο νοείται ως ακολουθία ή γράφημα ανεξάρτητων, πλήρως διαμορφώσιμων ενοτήτων που μπορούν να συνδεθούν μαζί με όσο το δυνατόν λιγότερους περιορισμούς. Συγκεκριμένα τα νευρωνικά επίπεδα, οι συναρτήσεις κόστους, οι βελτιστοποιητές, τα προγράμματα αρχικοποίησης, οι συναρτήσεις ενεργοποίησης και τα προγράμματα ρύθμισης είναι όλα ανεξάρτητα στοιχεία που μπορούν να συνδυαστούν για την δημιουργία νέων μοντέλων.
- Εύκολη επεκτασιμότητα. Η προσθήκη νέων στοιχείων είναι απλή. Για να είναι εύκολο να δημιουργηθούν νέες μονάδες, επιτρέπει τη συνολική εκφραστικότητα, καθιστώντας το Keras κατάλληλο για προηγμένη έρευνα.
- Υποστηρίζεται από την Python. Δεν υπάρχουν ξεχωριστά αρχεία ρυθμίσεων μοντέλων σε δηλωτική μορφή. Τα μοντέλα περιγράφονται στον κώδικα Python, ο οποίος είναι συμπαγής, ευκολότερος στην εκσφαλμάτωση και επιτρέπει την ευκολότερη επεκτασιμότητα.

Με την ενσωμάτωση του Keras στο TensorFlow δημιουργήθηκε μία εφαρμογή υψηλού επιπέδου tf.keras για την κατασκευή νευρωνικών δικτύων και άλλων μοντέλων μηχανικής μάθησης [30]. Έτσι παρέχονται όλα τα πλεονεκτήματα του keras και επίσης υπάρχει δυνατότητα πρόσβασης σε όλες τις τάξεις χαμηλού επιπέδου του TensorFlow. Όλα αυτά καθιστούν το keras πολύ δημοφιλή εφαρμογή στην ερευνητική κοινότητα και για αυτό το επιλέξαμε για την ανάπτυξη των νευρωνικών μας δικτύων.

1.7 Μια ματιά στον κόσμο των FPGAs

Πριν προχωρήσουμε στην εφαρμογή των εργαλείων είναι χρήσιμο να αναφέρουμε κάποιες γενικές πληροφορίες γύρω από τις πλακέτες FPGA (Field Programmable Gate Arrays). Τα FPGAs είναι συσκευές ημιαγωγών που βασίζονται σε μια μήτρα διαμορφωμένων λογικών μπλοκ (CLB) που συνδέονται μέσω προγραμματιζόμενων διασυνδέσεων [36]. Τα FPGA μπορούν να επαναπρογραμματιστούν στις επιθυμητές απαιτήσεις της εφαρμογής ή λειτουργικότητας ακόμη και μετά την κατασκευή τους. Αυτό το χαρακτηριστικό διακρίνει τα FPGAs από τα ειδικά εφαρμοσμένα ολοκληρωμένα κυκλώματα (ASICs), τα οποία κατασκευάζονται προσαρμοσμένα για συγκεκριμένες εφαρμογές σχεδιασμού. Βέβαια, αν και είναι διαθέσιμες (OTP) FPGAs που προγραμματίζονται μόνο μία φορά, οι κυρίαρχοι τύποι βασίζονται σε SRAM και μπορούν να επαναπρογραμματιστούν καθώς εξελίσσεται ο σχεδιασμός της εφαρμογής.

Κάθε τσιπ FPGA αποτελείται από έναν πεπερασμένο αριθμό προκαθορισμένων πόρων με προγραμματιζόμενες διασυνδέσεις για την υλοποίηση ενός αναδιαμορφώσιμου ψηφιακού κυκλώματος και μπλοκ εισόδου / εξόδου [37].



Εικόνα 16 Διαφορετικοί πόροι του FPGA [52]

Οι προδιαγραφές πόρων του FPGA περιλαμβάνουν συχνά τον αριθμό των διαμορφωμένων λογικών μπλοκ, τον αριθμό των σταθερών λογικών μπλοκ λειτουργιών όπως οι πολλαπλασιαστές, και το μέγεθος των πόρων μνήμης όπως η ενσωματωμένη μνήμη RAM. Από τις πολλές προδιαγραφές FPGA, αυτές είναι συνήθως οι πιο σημαντικές κατά την επιλογή και την σύγκριση των FPGAs για μια συγκεκριμένη εφαρμογή.

Τα διαμορφωμένα λογικά μπλοκ (CLB) είναι η βασική λογική μονάδα ενός FPGA. Μερικές φορές αναφερόμενες ως slices ή logic cells, τα CLB αποτελούνται από δύο βασικά στοιχεία: flip-flops και πίνακες αναζήτησης (LUTs). Τα κύρια χαρακτηριστικά μίας πλατφόρμας FPGA τα αναλύουμε παρακάτω.

Τα **flip-flops** είναι καταχωρητές δυαδικής μετατόπισης που χρησιμοποιούνται για τον συγχρονισμό της λογικής και την εξοικονόμηση λογικών καταστάσεων μεταξύ κύκλων ρολογιού μέσα σε ένα κύκλωμα FPGA. Σε κάθε κύκλο του ρολογιού, ένα flip-flop κλειδώνει την τιμή 1 ή 0 (TRUE ή FALSE) στην είσοδό του και κρατά αυτή την τιμή σταθερή μέχρι τον επόμενο κύκλο του ρολογιού.

Μεγάλο μέρος της λογικής των **CLBs** (configurable logic blocks) υλοποιείται χρησιμοποιώντας πολύ μικρές ποσότητες μνήμης RAM με τη μορφή **LUT** (look up table). Είναι εύκολο να υποθέσουμε ότι ο αριθμός πυλών του συστήματος σε ένα FPGA αναφέρεται στον αριθμό πυλών NAND και πυλών NOR σε ένα συγκεκριμένο τσιπ. Αλλά, στην πραγματικότητα, όλες οι συνδυαστικές λογικές (ANDs, ORs, NANDs, XORs κ.ο.κ.) υλοποιούνται ως πίνακες αλήθειας μέσα στη μνήμη LUT. Ένας πίνακας αλήθειας είναι μια προκαθορισμένη λίστα εξόδων για κάθε συνδυασμό εισόδων.

Τα FPGAs έχουν εγκατεστημένα κυκλώματα πολλαπλασιαστών (**Multipliers**) για εξοικονόμηση σε LUTs και flip-flops χρησιμοποιούνται σε μαθηματικές εφαρμογές και εφαρμογές επεξεργασίας σημάτων. Πολλοί αλγόριθμοι επεξεργασίας σήματος περιλαμβάνουν τη διατήρηση του συνολικού τρέχοντος αριθμού που πολλαπλασιάζεται και ως αποτέλεσμα τα FPGA υψηλότερης απόδοσης, όπως οι FPGA Xilinx Virtex-5, έχουν προκατασκευασμένα κυκλώματα πολλαπλών συσσωρευτών. Αυτά τα προκατασκευασμένα μπλοκ επεξεργασίας, επίσης γνωστά ως **slices DSP48**, ενσωματώνουν έναν πολλαπλασιαστή 25 bit με 18 bit με κύκλωμα με προσθέσεις.

Οι πόροι μνήμης (**Block RAMs**) είναι μια άλλη βασική προδιαγραφή των FPGAs. Η ορισμένη από το χρήστη RAM, ενσωματωμένη σε ολόκληρο το τσιπ FPGA, είναι χρήσιμη για την αποθήκευση συνόλων δεδομένων ή τη μεταφορά τιμών μεταξύ παράλληλων εργασιών. Ανάλογα με την οικογένεια FPGA, διαμορφώνεται και η ενσωματωμένη μνήμη RAM σε μπλοκ των 16, 18 ή 36 kb. Εξακολουθεί να υπάρχει βέβαια και η επιλογή εφαρμογής συνόλων δεδομένων ως συστοιχίες χρησιμοποιώντας flip-flops. Ωστόσο, οι μεγάλες συστοιχίες γίνονται γρήγορα δαπανηρές για τους πόρους του FPGA.

Ταυτόχρονα η εγγενής παράλληλη εκτέλεση των FPGA επιτρέπει την αυτόματη οδήγηση ανεξάρτητων λογικών υλικών από διαφορετικά ρολόγια. Η μετάδοση δεδομένων μεταξύ λογικής που εκτελείται με διαφορετικούς ρυθμούς μπορεί να είναι δύσκολη και η μνήμη χρησιμοποιεί συχνά για την εξομάλυνση της μεταφοράς στοίβες first in first out (**FIFO**).

Παραδοσιακά εργαλεία σχεδίασης FPGA

Οι γλώσσες περιγραφής υλικού (HDL) όπως η VHDL και η Verilog αποτελούν τις πρώτες γλώσσες για το σχεδιασμό των αλγορίθμων που τρέχουν σε τσιπ FPGA. Οι γλώσσες HDL αντικατοπτρίζουν ορισμένες από τις ιδιότητες άλλων γλωσσών κειμένου, αλλά διαφέρουν σημαντικά επειδή βασίζονται σε ένα μοντέλο ροής δεδομένων όπου οι είσοδοι κι οι έξοδοι είναι συνδεδεμένοι σε μια σειρά λειτουργικών μονάδων μέσω σημάτων.

Μετά την δημιουργία και την επαλήθευση ενός σχεδίου FPGA χρησιμοποιώντας HDL, τροφοδοτείται σε ένα εργαλείο compilation που παίρνει τη λογική που βασίζεται στο κείμενο και μέσω πολλών πολύπλοκων βημάτων συνθέτει το HDL κώδικα σε ένα αρχείο ρυθμίσεων ή bitstream που περιέχει πληροφορίες σχετικά με το πώς πρέπει να συνδέονται τα διάφορα στοιχεία μεταξύ τους.

Εργαλεία σχεδίασης υψηλού επιπέδου σύνθεσης

Ένας άλλος τρόπος να σχεδιάσουμε μία εφαρμογή σε FPGA είναι μέσω μίας γλώσσας υψηλού επιπέδου όπως C/C++. Για παράδειγμα έχουν αναπτυχθεί εργαλεία όπως τα Vivado HLx, τα οποία επιτυγχάνουν τη δημιουργία IP, καθιστώντας δυνατή την άμεση εφαρμογή των προδιαγραφών, που έχουν αναπτυχθεί με C/C ++ και System C, σε προγραμματιζόμενες συσκευές Xilinx χωρίς την ανάγκη δημιουργίας HDL κώδικα.

1.8 Προκλήσεις κατά την υλοποίηση νευρωνικών δικτύων με FPGA

Πολλές έρευνες έχουν δείξει ότι η χρήση επεξεργαστών FPGA στον τομέα της μηχανικής μάθησης έχει επιφέρει τεράστια επιτάχυνση των εφαρμογών [24, 25, 26]. Αυτό γίνεται διότι τα FPGAs λόγω της επαναπρογραμματιζόμενης φύσης τους μπορούν να προσαρμοστούν κάθε φορά στην επιθυμητή εφαρμογή. Έχουν δυνατότητα παραλληλοποίησης εργασιών που μειώνει την καθυστέρηση. Ταυτόχρονα, όλα αυτά μπορούν να υλοποιηθούν με πολύ χαμηλότερη κατανάλωση ενέργειας.

Βέβαια, κατά την υλοποίηση νευρωνικών δικτύων με FPGA υπάρχουν προκλήσεις και περιορισμοί που πρέπει να λάβουμε υπόψη μας. Η πρώτη πρόκληση έχει να κάνει με την διαθέσιμη μνήμη του FPGA. Τα FPGA συχνά έχουν λιγότερη από 10 MB μνήμη πάνω στο ολοκληρωμένο κύκλωμα (onchip) και δεν έχουν επιπλέον μνήμη ή αποθηκευτικό χώρο. Αυτό καθιστά πολύ δύσκολο τον σχεδιασμό ενός δικτύου που να ικανοποιεί από την μία πλευρά τις απαιτήσεις σε μνήμη και από την άλλη να πετυχαίνει υψηλή ακρίβεια πρόβλεψης. Συνεπώς, επικεντρωνόμαστε στο σχεδιασμό μικρών συνελικτικών μοντέλων, με μικρό αριθμό παραμέτρων προς εκπαίδευση και παράλληλα η αρχιτεκτονική και οι υπερ παράμετροι του δικτύου να είναι κατάλληλοι για τα δεδομένα μας, ώστε να πετυχαίνει υψηλή ακρίβεια. Ο τελικός στόχος μας είναι ο σχεδιασμός ενός μοντέλου που ικανοποιεί τις απαιτήσεις μνήμης του FPGA και δεν χρειάζεται να αφιερώνεται μεγάλο ποσοστό του χρόνου πρόβλεψης στην επικοινωνία για την ενημέρωση κάθε φορά των παραμέτρων.

Μία δεύτερη πρόκληση κατά την υλοποίηση νευρωνικών δικτύων σε FPGA είναι οι περιορισμένοι πόροι της πλατφόρμας για την ανάπτυξη της εφαρμογής. Ανάλογα το δίκτυο που επιθυμούμε να υλοποιήσουμε, πρέπει να οριστούν κατάλληλα τα στοιχεία του νευρωνικού προς υλοποίηση με στόχο χαμηλή καθυστέρηση και κατανάλωση ενέργειας.

Τέλος, μία πολύ μεγάλη πρόκληση είναι η τελική υλοποίηση του νευρωνικού σε γλώσσα περιγραφής υλικού. Ο πιο συνηθισμένος τρόπος για την δημιουργία ενός νευρωνικού δικτύου είναι με την χρήση ενός εργαλείου υψηλού επιπέδου όπως το Tensorflow ή το Keras σε γλώσσα python. Συνεπώς, για να γίνει σύνθεση του μοντέλου και να υλοποιηθεί με FPGA απαιτείται ανάπτυξη των στοιχείων του δικτύου είτε σε C/C++ είτε σε HDL, όπως αναφέραμε παραπάνω. Αυτή η διαδικασία είναι πολύ χρονοβόρα και απαιτεί εμπειρία που περιορίζει την εφαρμογή των FPGAs στον τομέα αυτό. Έτσι μεγάλο ερευνητικό ενδιαφέρον έχει στραφεί στην δημιουργία εργαλείων τα οποία μετασχηματίζουν τα δημιουργημένα μοντέλα, με tensorflow, keras, caffe ή κάποιο άλλο framework, σε μία από τις προαναφερόμενες γλώσσες σχεδιασμού. Το τελευταίο διάστημα έχουν αναπτυχθεί διάφορα τέτοιου είδους εργαλεία άλλα εμπορικά και άλλα ανοικτού κώδικα. Στην συγκεκριμένη διπλωματική εργασία δοκιμάσαμε τρία τέτοια εργαλεία. Αρχικά δοκιμάστηκαν δύο εργαλεία ανοικτού κώδικα το LeFlow [22] και το hls4ml [23] και στη συνέχεια το εμπορικό εργαλείο της Xilinx Machine Learning Suite [22, 23, 27].

2 Περίπτωση Νο1: Αναγνώριση παγόβουνων

Ένα παγόβουνο είναι ένα μεγάλο κομμάτι πάγου που έχει αποκοπεί από έναν παγετώνα. Κάθε παγόβουνο μπορεί να έχει διαφορετικό σχήμα και μέγεθος. Επειδή το μεγαλύτερο μέρος του παγόβουνου βρίσκεται κάτω από την επιφάνεια του νερού, παρασύρεται με τα ρεύματα των ωκεανών. Αυτό δημιουργεί κινδύνους στην πλοήγηση και στη υποδομή των πλοίων. Επί του παρόντος, πολλές εταιρείες και ιδρύματα χρησιμοποιούν αεροσκάφη για τον εντοπισμό παγόβουνων στις διαδρομές των πλοίων. Ωστόσο, σε απομακρυσμένες περιοχές με ιδιαίτερα δυσμενείς καιρικές συνθήκες, αυτοί οι μέθοδοι δεν είναι εφικτοί και η μόνη βιώσιμη επιλογή παρακολούθησης είναι μέσω δορυφόρων [3],[5],[6].

2.1 Ανάλυση και επεξεργασία δεδομένων

Τα δεδομένα είναι συνολικά 1604, διαστάσεων 75*75 και έχουμε τα αποτελέσματα δύο μετρήσεων (ζώνη_1 και ζώνη_2). Πρέπει να σημειωθεί ότι οι τιμές δεν είναι μη αρνητικοί ακέραιοι αριθμοί, όπως συνήθως συμβαίνει στις εικόνες. Αυτό συμβαίνει επειδή έχουν φυσική σημασία και είναι δεκαδικοί αριθμοί και αντιστοιχούν σε μονάδες dB. Οι ζώνες 1 και 2 είναι σήματα τα οποία έχουν καταγραφεί από ραντάρ. Τα σήματα παράγονται από διαφορετικές πολώσεις και με συγκεκριμένη γωνία πρόσπτωσης. Οι πολώσεις αντιστοιχούν σε HH (μετάδοση / λήψη οριζόντια) και HV (μεταδίδουν οριζόντια και λαμβάνουν κατακόρυφα). Για κάθε εικόνα δίνονται οι τιμές των σημάτων 75*75 της ζώνης_1 και της ζώνης_2, η γωνία πρόσπτωσης, ο κωδικός της εικόνας καθώς και η κατηγορία που ανήκει. Οι εικόνες χωρίζονται σε δύο κατηγορίες, η μία είναι τα «παγόβουνα» και η άλλη είναι τα «πλοία».

Για την εκπαίδευση των νευρωνικών μοντέλων κάναμε χρήση των δεδομένων ζώνη_1 και ζώνη_2. Την πληροφορία για την γωνία πρόσπτωσης δεν την χρησιμοποιούμε ως είσοδο στα μοντέλα, αφού υπάρχουν παραπάνω από 130 τιμές που λείπουν. Επίσης, χωρίσαμε τα δεδομένα σε δεδομένα εκπαίδευσης (train data) και δεδομένα ελέγχου (test data). Για την εκπαίδευση χρησιμοποιήσαμε το 80% των δεδομένων (1283 εικόνες) και τις υπόλοιπες για τον έλεγχο (321 εικόνες). Επίσης, εφαρμόσαμε δύο από τους μηχανισμούς της αύξησης δεδομένων περιστροφή και αντιστροφή. Αυτό είχε ως αποτέλεσμα τον τριπλασιασμό των δεδομένων προς εκπαίδευση και προς έλεγχο, χωρίς όμως να οδηγήσει σε περαιτέρω αύξησης της ακρίβειας των μοντέλων.

Κατά την εκπαίδευση ενός νευρικού δικτύου, υπάρχει ένας αριθμός υπερπαραμέτρων για επιλογή, συμπεριλαμβανομένου του αριθμού των κρυφών επιπέδων (hidden layers), του αριθμού των νευρώνων για κάθε κρυφό επίπεδο (filters), του ρυθμού εκμάθησης (learning rate), της συνάρτησης ενεργοποίησης (activation function) αλλά και του αλγορίθμου βελτιστοποίησης (optimizer). Η δημιουργία του βέλτιστου συνδυασμού από τέτοιους υπερπαραμέτρους είναι ένα δύσκολο έργο.

2.2 Διερεύνηση με πλήρως συνδεδεμένα νευρωνικά δίκτυα

Για τα συγκεκριμένα δεδομένα δημιουργήσαμε τρία νευρωνικά δίκτυα τριών, τεσσάρων και πέντε πλήρως συνδεδεμένων επιπέδων. Στην συνέχεια εκπαιδεύσαμε τα μοντέλα για διάφορους συνδυασμούς υπερπαραμέτρων σχετικά με τον αριθμό των φίλτρων σε κάθε κρυφό επίπεδο, την συνάρτηση ενεργοποίησης αλλά και του αλγόριθμου βελτιστοποίησης. Οι συναρτήσεις ενεργοποίησης που δοκιμάστηκαν ήταν η σιγμοειδής (sigmoid) και η υπερβολική εφαπτομένη (tanh). Αντίστοιχα, οι αλγόριθμοι βελτιστοποίησης που δοκιμάστηκαν ήταν της στοχαστικής κλίσης (SGD), του προσαρμοστικού υπολογισμού ροπής (Adam) και της μέσης τετραγωνικής διάδοσης των ριζών (RMSprop). Τα αποτελέσματα φαίνονται στον παρακάτω πίνακα.

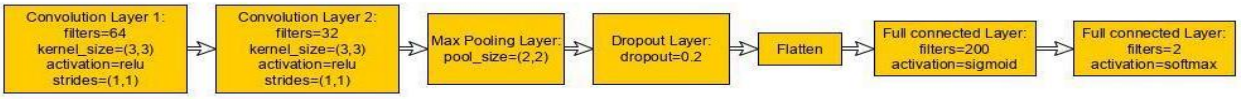
Αλγόριθμος βελτιστοποίησης	Αριθμός Επιπέδων	Φίλτρα-Νευρώνες σε κάθε επίπεδο	Συνάρτηση ενεργοποίησης	Συνάρτηση κόστους	Ακρίβεια (%)
Adam	3	100_50_2	sigmoid	0.7	49.22
Adam	3	300_100_2	sigmoid	0.7	49.22
SGD	3	100_50_2	sigmoid	0.71	48.29
SGD	3	300_80_2	sigmoid	0.66	63.24
RMSprop	3	100_50_2	sigmoid	0.7	49.22
RMSprop	3	250_90_2	sigmoid	0.7	49.22
SGD	3	300_80_2	tanh	0.53	74.14
SGD	3	300_50_2	tanh	0.61	65.42
SGD	3	600_200_2	tanh	0.78	60.44
Adam	4	500_200_80_2	tanh	0.7	54.21
SGD	4	500_200_80_2	tanh	0.64	67.6
RMSprop	4	500_200_80_2	tanh	0.76	45.79
SGD	4	400_150_50_2	tanh	0.58	69.16
SGD	4	600_200_50_2	tanh	0.55	70.4
Adam	5	200_100_60_30_2	sigmoid	0.69	54.21
SGD	5	200_100_60_30_2	sigmoid	0.69	54.21
RMSprop	5	200_100_60_30_2	sigmoid	0.69	54.21
Adam	5	500_200_100_50_2	sigmoid	0.69	54.21
SGD	5	500_200_100_50_2	sigmoid	0.69	54.21
RMSprop	5	500_200_150_50_2	tanh	0.69	64.17
SGD	5	400_300_200_70_2	tanh	1.51	48.29
SGD	5	600_400_200_70_2	tanh	0.56	69.47

Πίνακας 1 Αποτελέσματα εκπαίδευσης πλήρως συνδεδεμένων νευρωνικών δικτύων

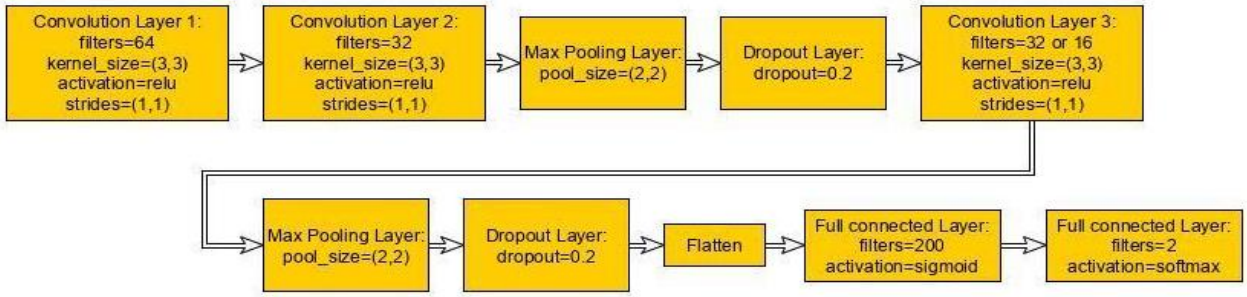
Παρατηρώντας τα αποτελέσματα για τις διάφορες τιμές των υπερπαραμέτρων καταλήγουμε στο συνδυασμό του πίνακα που είναι με έντονα γράμματα. Ο αλγόριθμος βελτιστοποίησης είναι της στοχαστικής κλίσης (SGD), η συνάρτηση ενεργοποίησης είναι η υπερβολική εφαπτομένη (tanh) και τα κρυφά επίπεδα είναι 2 με νευρώνες 300 και 80, αντίστοιχα.

2.3 Διερεύνηση με συνελικτικά νευρωνικά δίκτυα

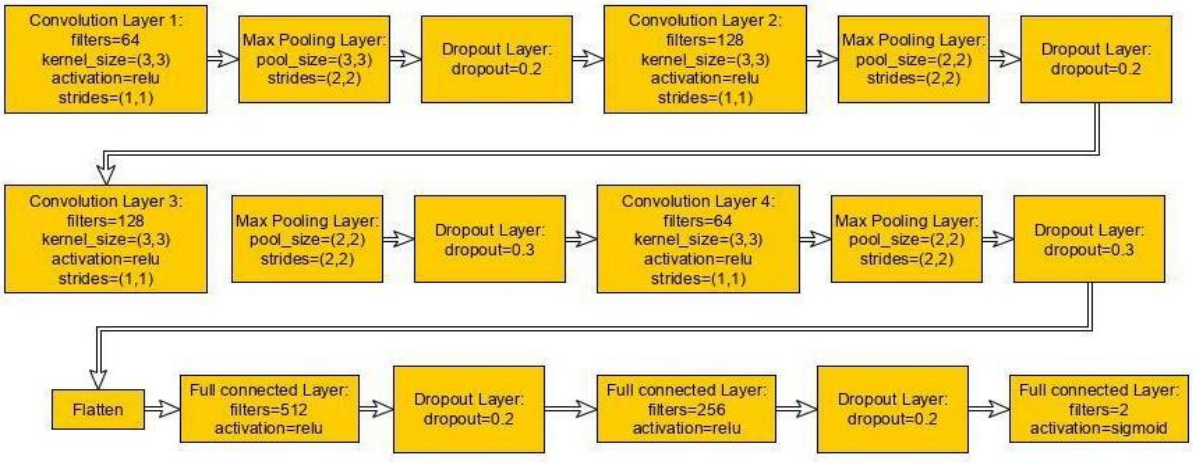
Για την εκπαίδευση με συνελικτικό νευρωνικό δίκτυο επιλέχθηκαν τρεις αρχιτεκτονικές δικτύων. Οι αρχιτεκτονικές εικονίζονται παρακάτω και είναι των δύο, των τριών και των τεσσάρων συνελικτικών επιπέδων.



Εικόνα 17 Αρχιτεκτονική μοντέλου με δύο συνελικτικά επίπεδα

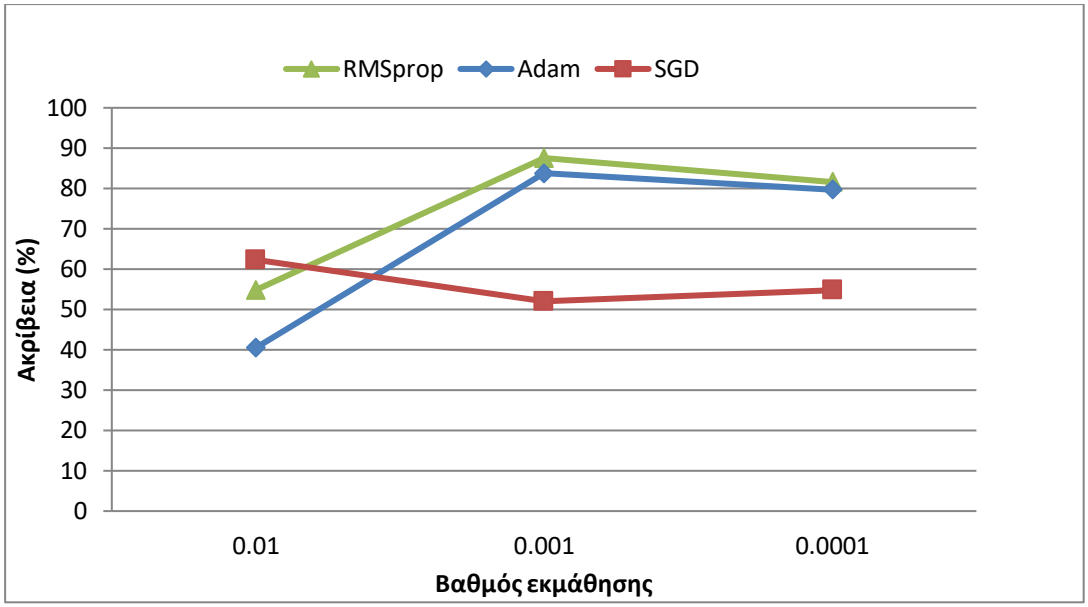


Εικόνα 18 Αρχιτεκτονική μοντέλου με τρία συνελικτικά επίπεδα

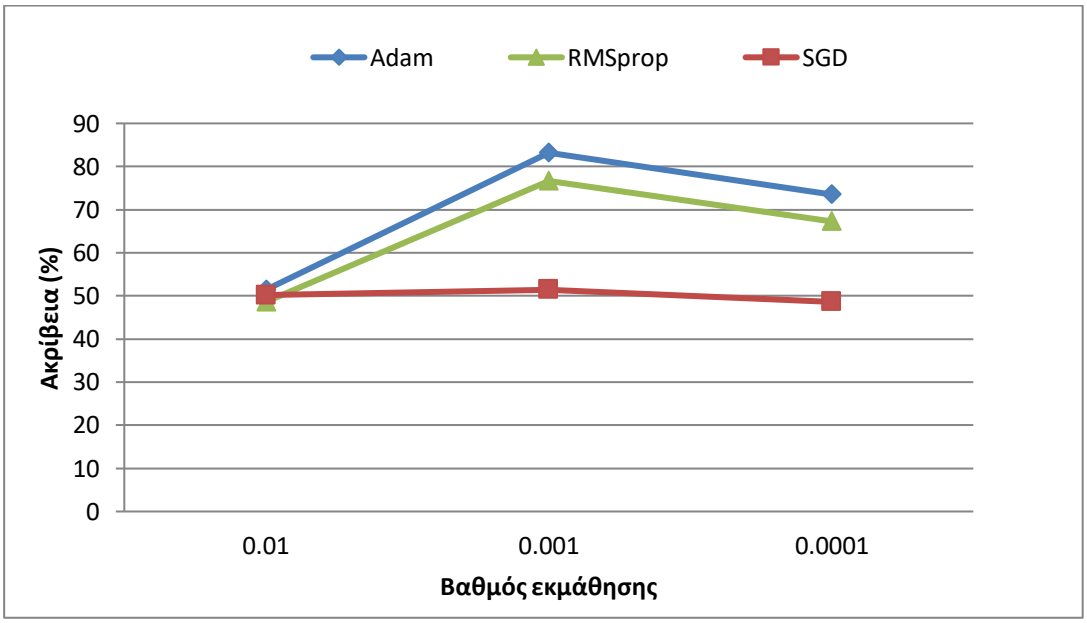


Εικόνα 19 Αρχιτεκτονική μοντέλου με τέσσερα συνελικτικά επίπεδα

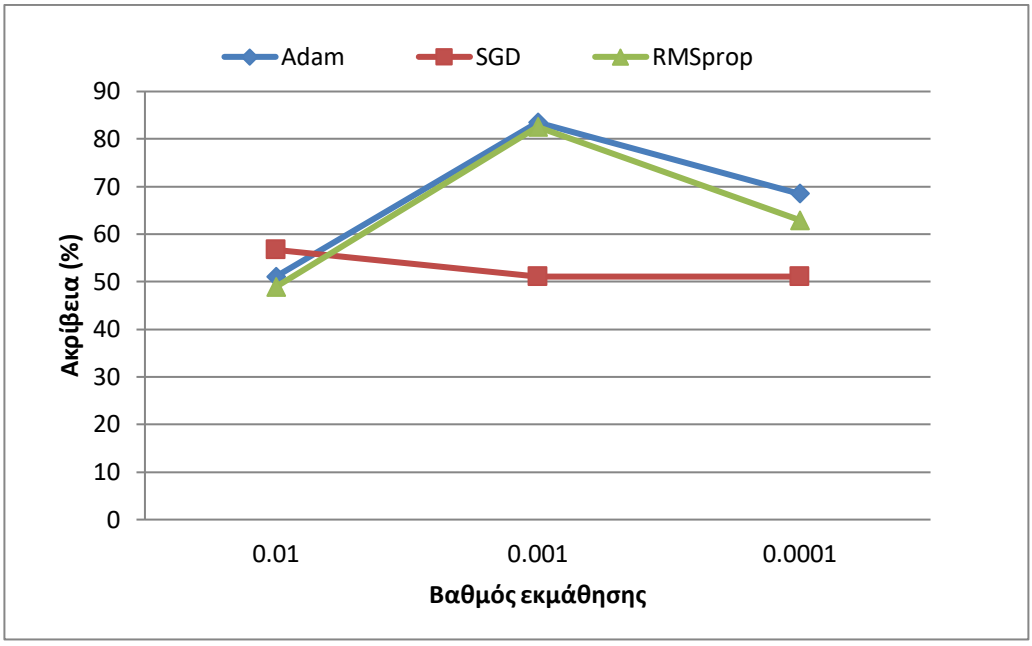
Αυτή την φορά οι υπερπαράμετροι προς βελτιστοποίηση ήταν ο βαθμός εκμάθησης (learning rate), ο αλγόριθμος βελτιστοποίησης (optimizer) και ο αριθμός των φίλτρων σε ορισμένα συνελικτικά επίπεδα. Οι υπόλοιπες τιμές των παραμέτρων παρέμειναν σταθεροί σε κάθε εκπαίδευση, όπως αναγράφονται στις παραπάνω εικόνες. Τα αποτελέσματα αναπαριστώνται στα διαγράμματα παρακάτω.



Εικόνα 20 Η ακρίβεια των μοντέλων σε σχέση με τον βαθμό εκμάθησης και τον αλγόριθμο βελτιστοποίησης για την αρχιτεκτονική με δύο συνελικτικά επίπεδα και φίλτρα 64/32



Εικόνα 21 Η ακρίβεια των μοντέλων σε σχέση με τον βαθμό εκμάθησης και τον αλγόριθμο βελτιστοποίησης για την αρχιτεκτονική με τρία συνελικτικά επίπεδα και φίλτρα 64/32/16



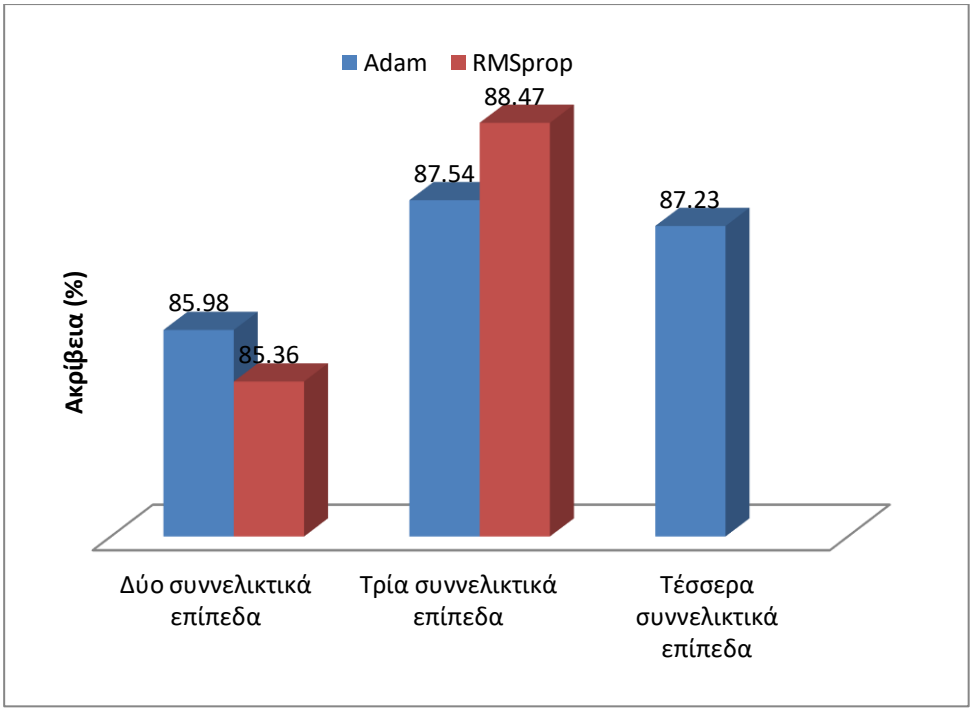
Εικόνα 22 Η ακρίβεια των μοντέλων σε σχέση με τον βαθμό εκμάθησης και τον αλγόριθμο βελτιστοποίησης για την αρχιτεκτονική με τρία συνελικτικά επίπεδα και φίλτρα 64/32/32

Στην πρώτη αρχιτεκτονική εκπαιδεύσαμε τα δίκτυα δοκιμάζοντας τρεις αλγορίθμους βελτιστοποίησης (optimizer= Adam, SGD, RMSprop) και τρεις τιμές για τον βαθμό εκμάθησης (learning rate = 0.01, 0.001, 0,0001). Την μεγαλύτερη ακρίβεια 87.54% την έδωσε το μοντέλο με αλγόριθμο βελτιστοποίησης RMSprop και βαθμό εκμάθησης 0.001.

Στην δεύτερη αρχιτεκτονική ακολουθήσαμε την ίδια λογική, με την διαφορά ότι επιπλέον δοκιμάσαμε διαφορετικές τιμές (32 και 16) στα φίλτρα του τρίτου συνελικτικού επιπέδου. Την μεγαλύτερη ακρίβεια 83.49% την έδωσε το μοντέλο με αλγόριθμο βελτιστοποίησης Adam, βαθμό εκμάθησης 0.001 και 32 φίλτρα στο τρίτο συνελικτικό επίπεδο.

Σε αντίθεση με τις δύο πρώτες αρχιτεκτονικές που υπήρχαν παράμετροι προς βελτιστοποίηση, η τρίτη αρχιτεκτονική έχει όλες τις παραμέτρους σταθερές. Αυτό συμβαίνει διότι αποτελεί μία από τις λύσεις του διαγωνισμού και υπάρχει στο αποθετήριο του github [18]. Συνεπώς, συμπεριλήφθηκε για σύγκριση των αποτελεσμάτων σε σχέση με τα μοντέλα που δημιουργήσαμε και όχι για βελτιστοποίηση. Η ακρίβεια του μοντέλου ήταν 81.93% με αλγόριθμο βελτιστοποίησης Adam και βαθμό εκμάθησης 0.001.

Στην συνέχεια αλλάξαμε τα δεδομένα εισόδου, δημιουργώντας έναν επιπλέον πίνακα με το άθροισμα της ζώνης_1 και της ζώνης_2. Αυτή την φορά εκπαιδεύσαμε μόνο τα μοντέλα με βαθμό εκμάθησης 0.001 και αλγόριθμους βελτιστοποίησης Adam και RMSprop, λόγω της χαμηλής ακρίβειας των υπόλοιπων μοντέλων. Στην δεύτερη αρχιτεκτονική τα φίλτρα του τρίτου συνελικτικού επιπέδου είναι 32 για τον αλγόριθμο βελτιστοποίησης Adam και 16 για τον RMSprop. Τα συνολικά αποτελέσματα για τις τρεις αρχιτεκτονικές απεικονίζονται στο διάγραμμα που ακολουθεί.

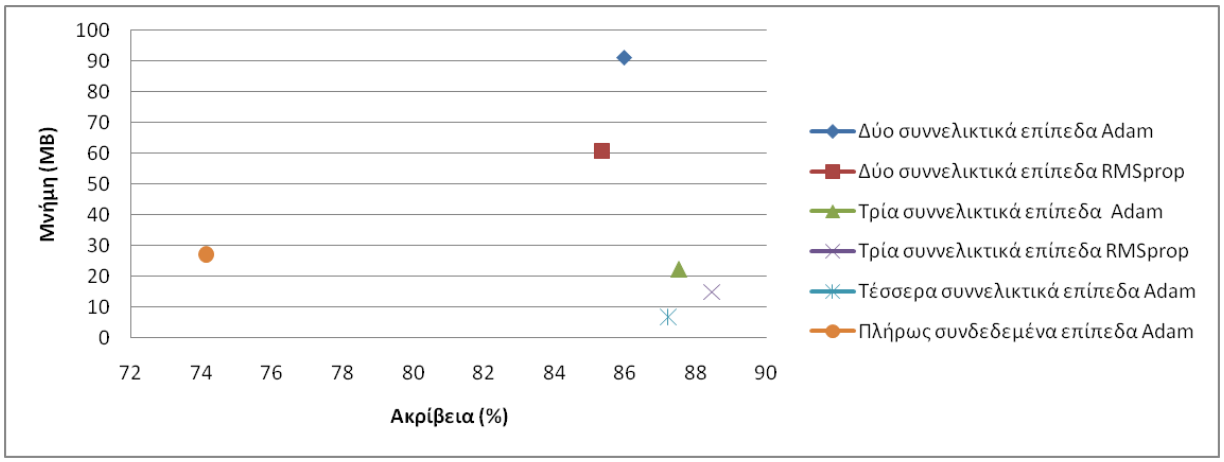


Εικόνα 23 Ακρίβεια μοντέλων για τις τρεις αρχιτεκτονικές με αλλαγή στα δεδομένα εισόδου

2.4 Επιλογή μοντέλου με βάση την ακρίβεια και την μνήμη

Με την συνεχή εξέλιξη των συνελικτικών νευρωνικών δικτύων (CNN), μια γενική τάση είναι να μεγεθύνεται τόσο το μέγεθος του δικτύου όσο και η πολυπλοκότητα του υπολογισμού. Για να ικανοποιήσουν την αυξανόμενη ζήτηση για υπολογισμό ερευνητές έχουν χρησιμοποιήσει ή δημιουργήσει διάφορες πλατφόρμες υλικού υψηλής απόδοσης, συμπεριλαμβανομένων των GPU ή προσαρμοσμένων επιταχυντών όπως το FPGA και το ASIC για τη βελτίωση της απόδοσης και της αποτελεσματικότητας. Οι επιταχυντές που βασίζονται σε FPGA έχουν αποκτήσει δημοτικότητα στην επιτάχυνση των μοντέλων CNN μεγάλης κλίμακας επειδή μπορούν να πετύχουν χαμηλότερη λανθάνουσα κατάσταση και καταναλώνουν πολύ λιγότερη ισχύ σε σύγκριση με τις GPU. Είναι επίσης πιο ευέλικτο σε σύγκριση με το ASICs [19].

Ωστόσο, η ανάπτυξη εφαρμογών στην πλακέτα συναντά και πολλούς περιορισμούς. Τα FPGAs συνήθως έχουν λιγότερες μονάδες κινητής υποδιαστολής και μνήμης, η οποία περιορίζει τη συνολική υπολογιστική ισχύ τους. Ταυτόχρονα, οι υπολογιστικοί πόροι που απαιτεί ένα νευρωνικό δίκτυο μπορεί να είναι περισσότεροι, με αποτέλεσμα να είναι εξαιρετικά δύσκολο να εξισορροπηθεί η κατανομή πόρων σε ένα μόνο FPGA. Συνεπώς κατά την ανάπτυξη ενός νευρωνικού δικτύου μας ενδιαφέρει να καταφέρουμε όσο το δυνατόν μεγαλύτερη ακρίβεια και παράλληλα όσο το δυνατόν μικρότερες απαιτήσεις σε μνήμη.



Εικόνα 24 Διάγραμμα που απεικονίζει την μνήμη των μοντέλων σε σχέση με την ακρίβεια

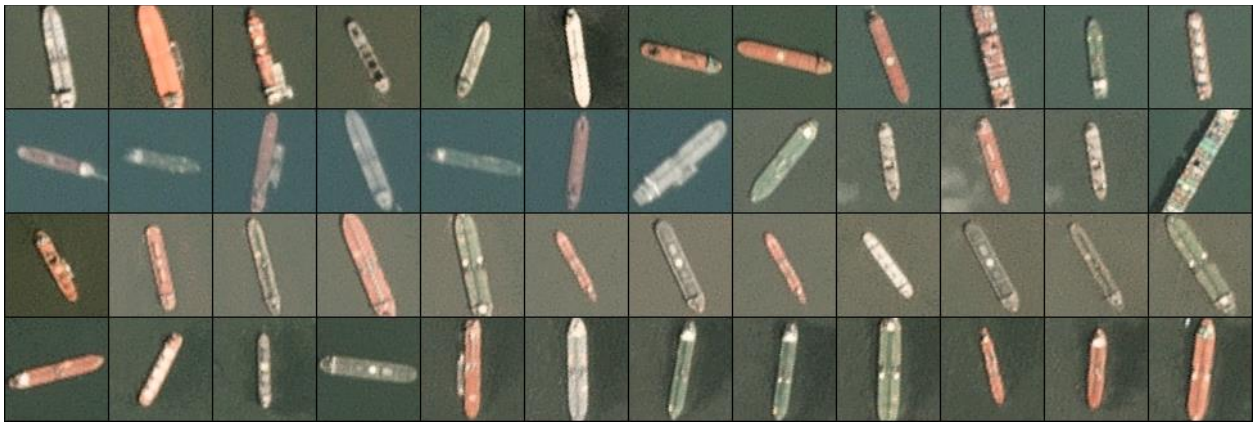
Στην εικόνα 23 αναπαριστώνται τα βασικά μοντέλα που εξετάσαμε και διαγράφεται η ακρίβειά τους σε σχέση με την μνήμη που απαιτούν. Οι βέλτιστες λύσεις κατά παρέτο είναι δύο. Το ένα μοντέλο είναι αυτό με τρία συνελικτικά επίπεδα, αλγόριθμο βελτιστοποίησης RMSprop, βαθμό εκμάθησης 0.001, 32 φίλτρα στο τρίτο συνελικτικό επίπεδο, ακρίβεια 88.47%, συνάρτηση κόστους 0.29 και συνολική απαίτηση σε μνήμη 15MB. Το άλλο μοντέλο είναι αυτό με τέσσερα συνελικτικά επίπεδα, αλγόριθμο βελτιστοποίησης Adam, βαθμό εκμάθησης 0.001, ακρίβεια 87.23%, συνάρτηση κόστους 0.31 και συνολική απαίτηση σε μνήμη 6.9MB.

3 Περίπτωση No2: Αναγνώριση πλοίων από δορυφορικές εικόνες

3.1 Ανάλυση και επεξεργασία δεδομένων

Τα δεδομένα συγκεντρώθηκαν από την αμερικανική εταιρεία Planet στις περιοχές του κόλπου του Σαν Φρανσίσκο και του κόλπου του Σαν Πέδρο της Καλιφόρνιας. Συνολικά είναι 4000, διαστάσεων 80*80 RGB φωτογραφίες, οι οποίες είναι ταξινομημένες σε «πλοία» και «μη-πλοία». Η ανάλυση των εικόνων είναι τέτοια ώστε κάθε εικονοστοιχεία αντιστοιχεί σε τρία μέτρα.

Η κλάση των «πλοίων» περιλαμβάνει 1000 φωτογραφίες. Τα πλοία που περιλαμβάνονται έχουν διάφορους προσανατολισμούς και μεγέθη, καθώς επίσης υπάρχουν διαφορές και στις ατμοσφαιρικές συνθήκες συλλογής τους. Δείγμα εικόνων αυτής της κλάσης φαίνεται παρακάτω.



Εικόνα 25 Δείγμα εικόνων από την κατηγορία "πλοία" [53]

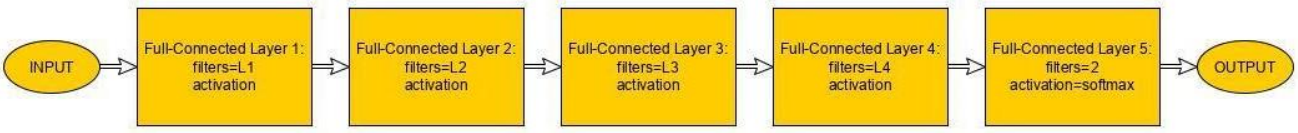
Η κλάση των «μη-πλοία» περιλαμβάνει 3000 φωτογραφίες, οι οποίες κατηγοριοποιούνται σε τρεις υποομάδες. Η μία ομάδα περιλαμβάνει διάφορα τοπία της γης όπως θάλασσα, στεριά, πόλεις κτλ. Η δεύτερη ομάδα φωτογραφιών αποτελείται από τμήματα πλοίων τα οποία δεν είναι επαρκή για να ενταχθούν στην κατηγορία «πλοία». Η τελευταία ομάδα περιλαμβάνει εικόνες που έχουν προηγουμένως αποτύχει από άλλα μοντέλα μηχανικής μάθησης, που συνήθως προκαλείται από φωτεινά εικονοστοιχεία ή ισχυρά γραμμικά χαρακτηριστικά. Δείγμα εικόνων αυτής της κλάσης φαίνεται παρακάτω.

Εικόνα 26 Δείγμα εικόνων από την κατηγορία "μη-πλοία" [54]

Για την εκπαίδευση των νευρωνικών δικτύων υπάρχει ένας αριθμός υπερπαραμέτρων για επιλογή. Στα συγκεκριμένα δεδομένα οι υπερπαράμετροι που μας ενδιέφεραν προς βελτιστοποίηση ήταν ο αριθμός των κρυφών επιπέδων (hidden layers), ο αριθμός των νευρώνων για κάθε κρυφό επίπεδο (filters), η συνάρτηση ενεργοποίησης (activation function), ο ρυθμός εκμάθησης (learning rate), ο αλγόριθμος βελτιστοποίησης (optimizer) καθώς και το ποσοστό του επιπέδου απορρίψεων. Όπως και στα προηγούμενα δεδομένα χρησιμοποιήθηκαν το 80% των δεδομένων (3200 εικόνες) για εκπαίδευση και το υπόλοιπο 20% για έλεγχο (800 εικόνες).

3.2 Διερεύνηση με πλήρως συνδεδεμένα νευρωνικά δίκτυα

Στην παρακάτω εικόνα φαίνεται η αρχιτεκτονική του νευρωνικού δικτύου με πέντε πλήρως συνδεδεμένα νευρωνικά επίπεδα, το οποίο εκπαιδεύτηκε για διάφορες τιμές των υπερπαραμέτρων του.



Εικόνα 27 Αρχιτεκτονική νευρωνικού δικτύου με πέντε πλήρως συνδεδεμένα νευρωνικά επίπεδα

Για την βελτιστοποίηση των υπερπαραμέτρων ακολουθήσαμε την εξής μεθοδολογία. Αρχικά εκπαιδεύσαμε το δίκτυο για διάφορους συνδυασμούς τιμών στα φίλτρα κάθε επιπέδου L1, L2, L3 και L4. Ο αλγόριθμος βελτιστοποίησης ήταν ο Adam με βαθμό εκμάθησης 0.001 και ως συνάρτηση ενεργοποίησης χρησιμοποιήθηκε η σιγμοειδής.

Φίλτρα/Νευρώνες σε κάθε επίπεδο (L1_L2_L3_L4)	Συνάρτηση Κόστους	Ακρίβεια (%)
200_100_60_30	0.34	85.75
500_200_100_50	0.28	89.25
150_100_50_20	0.36	86.5
170_110_70_30	0.32	81.25
120_90_60_25	0.33	89
100_80_40_20	0.37	85.38
600_300_150_80	**0.27**	**90.88**
400_250_100_40	0.26	88.38
700_350_200_90	0.26	86.62
800_400_150_70	0.31	84.38

Πίνακας 2 Αποτελέσματα εκπαίδευσης συνελικτικού δικτύου με διαφορετικό αριθμό νευρώνων σε κάθε επίπεδο

Στη συνέχεια για τον καλύτερο συνδυασμό φίλτρων-νευρώνων (L1=600, L2=300, L3=150, L4=80) εκπαιδεύσαμε το δίκτυο με συνάρτηση ενεργοποίησης την αυστηρά σιγμοειδής (hard sigmoid) καθώς και με την υπερβολική εφαπτομένη (tanh).

Συνάρτηση ενεργοποίησης	Φίλτρα/Νευρώνες επιπέδων	Συνάρτηση Κόστους	Ακρίβεια (%)
sigmoid	**600_300_150_80**	**0.27**	**90.88**
hard_sigmoid	600_300_150_80	0.32	89.38
tanh	600_300_150_80	0.55	76.5

Πίνακας 3 Αποτελέσματα εκπαίδευσης συνελικτικού δικτύου με διαφορετικές συναρτήσεις ενεργοποίησης

Τέλος, εκπαιδεύσαμε το δίκτυο με την σιγμοειδής συνάρτηση ενεργοποίησης που έφερε την μεγαλύτερη ακρίβεια, αλλάζοντας τον αλγόριθμο βελτιστοποίησης από τον προσαρμοστικό υπολογισμό ροπής (Adam) στον αλγόριθμο μέσης τετραγωνικής διάδοσης των ριζών (RMSprop) και στον αλγόριθμο της στοχαστικής κλίσης (SGD).

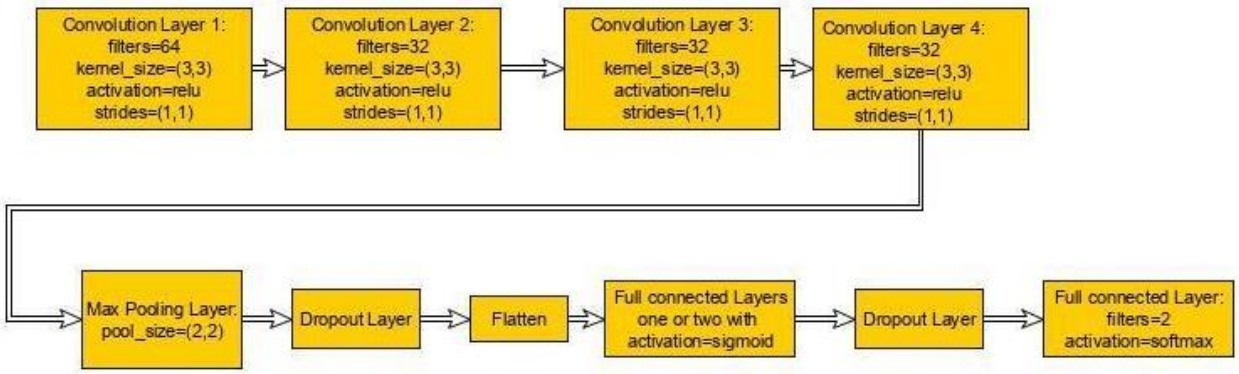
Αλγόριθμος βελτιστοποίησης	Συνάρτηση ενεργοποίησης	Φίλτρα/Νευρώνες επιπέδων	Συνάρτηση Κόστους	Ακρίβεια (%)
Adam	**sigmoid**	**600_300_150_80**	**0.27**	**90.88**
RMSprop	sigmoid	600_300_150_80	0.28	86.25
SGD	sigmoid	600_300_150_80	0.54	76.38

Πίνακας 4 Αποτελέσματα εκπαίδευσης συνελικτικού δικτύου με διαφορετικούς αλγορίθμους βελτιστοποίησης

Παρατηρώντας τα αποτελέσματα των εκπαιδεύσεων για τους διάφορους συνδυασμούς καταλήγουμε στο δίκτυο με αλγόριθμο βελτιστοποίησης Adam, σιγμοειδής συνάρτηση ενεργοποίησης και με αριθμό νευρώνων L1=600, L2=300, L3=150 και L4=80 σε κάθε κρυφό επίπεδο. Το συγκεκριμένο μοντέλο είχε ακρίβεια 90.88% και συνάρτηση κόστους 0.27.

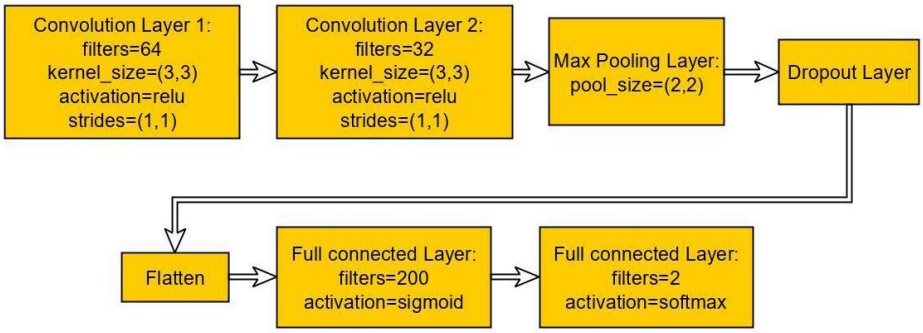
3.3 Διερεύνηση με συνελικτικά νευρωνικά δίκτυα

Για την εκπαίδευση συνελικτικού νευρωνικού δικτύου αρχικά κατασκευάστηκε ένα δίκτυο με τέσσερα συνελικτικά επίπεδα όπως αναπαριστάται παρακάτω. Ο αλγόριθμος βελτιστοποίησης ήταν ο Adam με βαθμό εκπαίδευσης 0.001.



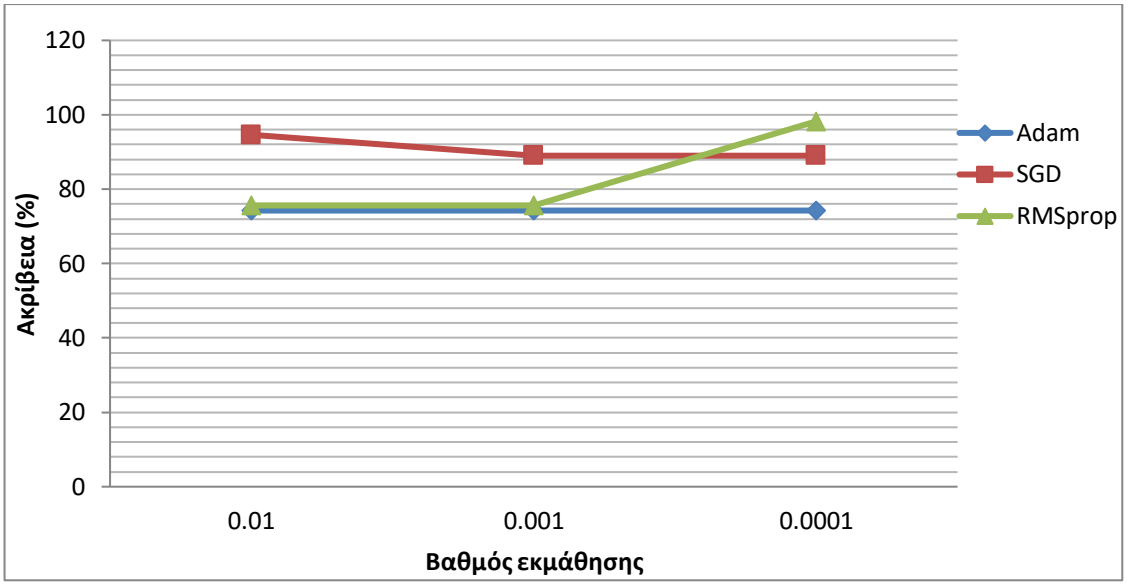
Εικόνα 28 Αρχιτεκτονική μοντέλου με τέσσερα συνελικτικά επίπεδα

Οι υπερπαράμετροι που εξετάστηκαν ήταν το πλήθος των επιπέδων και ο αριθμός των νευρώνων των πλήρως συνδεδεμένων επιπέδων, καθώς και το ποσοστό του επιπέδου απορρίψεων (dropout). Οι τιμές του ποσοστού απορρίψεων που δοκιμάστηκαν ήταν 0, 0.1 και 0.15. Το μοντέλο με ένα κρυφό επίπεδο πλήρως συνδεδεμένων νευρώνων εκπαιδεύτηκε για 100 και για 200 νευρώνες, ενώ το μοντέλο με δύο κρυφά επίπεδα εκπαιδεύτηκε με συνδυασμούς νευρώνων 200_80 και 300_100. Τα μοντέλα που προέκυψαν είχαν ακρίβεια από 73.25% έως και 76.75%. Το ποσοστό ακρίβειας δεν ήταν ικανοποιητικό και συνεπώς κατασκευάσαμε την παρακάτω αρχιτεκτονική με δύο συνελικτικά επίπεδα.

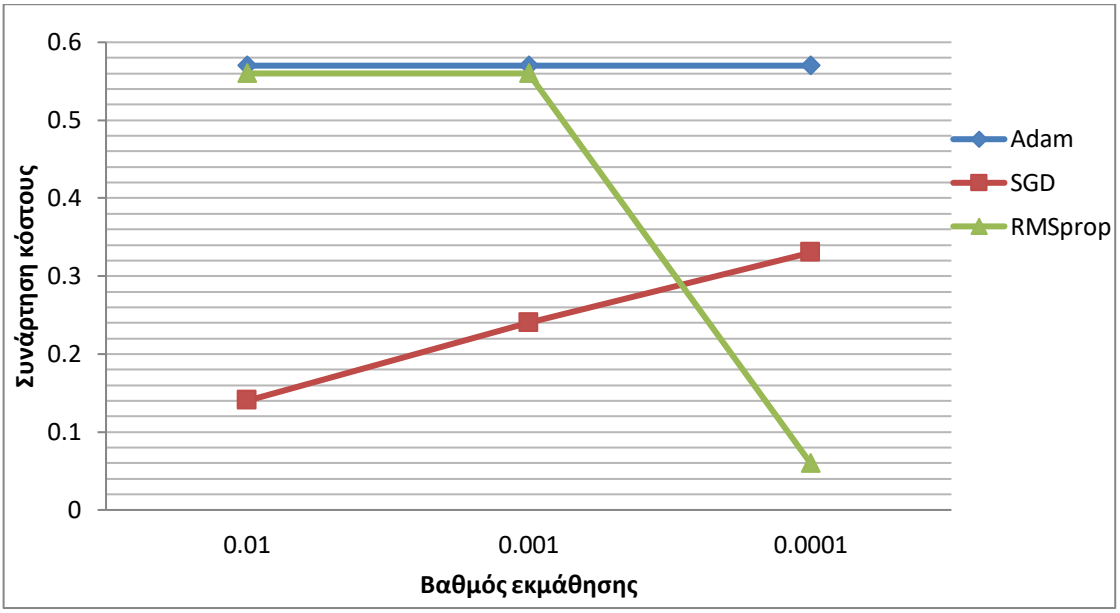


Εικόνα 29 Αρχιτεκτονική μοντέλου με δύο συνελικτικά επίπεδα και ένα κρυφό επίπεδο πλήρως συνδεδεμένων νευρώνων

Οι υπερπαράμετροι που εξετάστηκαν ήταν ο αλγόριθμος βελτιστοποίησης και ο βαθμός εκμάθησης. Το ποσοστό των απορρίψεων επιλέχθηκε στο 10%. Αρχικά εκπαιδεύσαμε το δίκτυο για τους τρεις αλγόριθμους βελτιστοποίησης και τρεις τιμές στο βαθμό εκμάθησης. Τα αποτελέσματα φαίνονται στα παρακάτω διαγράμματα.



Εικόνα 30 Ακρίβεια συναρτήσει του βαθμού εκμάθησης για τους τρεις αλγορίθμους βελτιστοποίησης



Εικόνα 31 Συνάρτηση κόστους συναρτήσει του βαθμού εκμάθησης για τους τρεις αλγορίθμους βελτιστοποίησης

Η εκπαίδευση των δικτύων έγινε για 15 εποχές. Στη συνέχεια εκπαιδεύσαμε για κάθε αλγόριθμο βελτιστοποίησης το μοντέλο με την καλύτερη ακρίβεια για το διάστημα των 40 εποχών.

Αλγόριθμος βελτιστοποίησης	Φίλτρα σε κάθε συννελικτικό επίπεδο	Βαθμός εκμάθησης	Συνάρτηση κόστους	Ακρίβεια (%)	Μνήμη (MB)
Adam	64_32	0.01	0.56	75.25	98.6
SGD	64_32	0.01	0.07	97.5	82.1
RMSprop	64_32	0.0001	0.56	75.25	82.1

Πίνακας 5 Εκπαίδευση μοντέλων για 40 εποχές

Με τα παραπάνω αποτελέσματα καταλήξαμε ότι για τα συγκεκριμένα δεδομένα καταλληλότερος είναι ο αλγόριθμος βελτιστοποίησης SGD με βαθμό εκμάθησης (learning rate) 0.01. Το συγκεκριμένο νευρωνικό δίκτυο καταφέρνει υψηλή ακρίβεια πρόβλεψης και ταυτόχρονα χαμηλή απαίτηση μνήμης σε σχέση με τα υπόλοιπα μοντέλα. Βέβαια, παρόλο που η ακρίβεια πρόβλεψης μπορεί να θεωρηθεί ικανοποιητική, δεν ισχύει το ίδιο και για τις απαιτήσεις σε μνήμη. Τρόπους για περαιτέρω μείωση της μνήμης διατηρώντας υψηλή ακρίβεια πρόβλεψης θα αναλύσουμε στο επόμενο κεφάλαιο.

4 Βελτιστοποίηση ως προς μνήμη

Πολλές από τις πρόσφατες έρευνες σχετικά με τα βαθιά συνελικτικά νευρωνικά δίκτυα (CNNs) επικεντρώθηκαν στην αύξηση της ακρίβειας. Για δεδομένο επίπεδο ακρίβειας, συνήθως υπάρχουν πολλαπλές αρχιτεκτονικές που το επιτυγχάνουν. Δεδομένης της ισοδύναμης ακρίβειας, μια αρχιτεκτονική συνελικτικού δικτύου (CNN) με λιγότερες παραμέτρους έχει αρκετά πλεονεκτήματα.

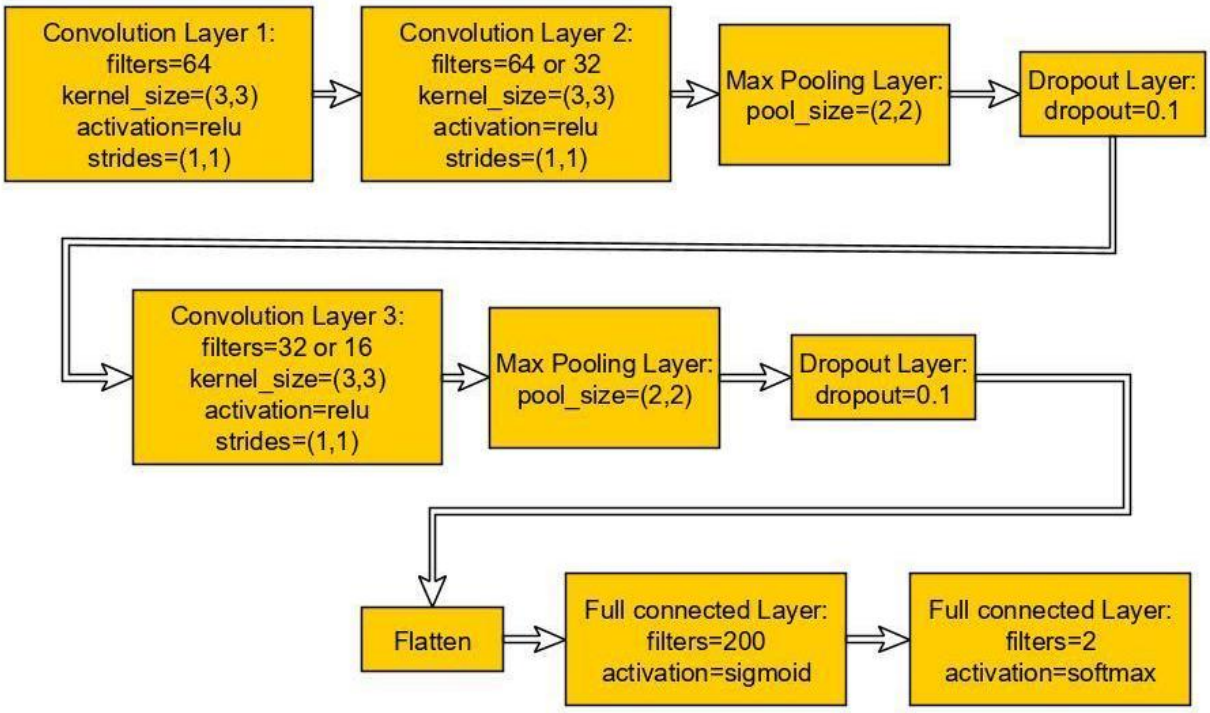
Ένα από τα βασικά πλεονεκτήματα είναι η αποτελεσματικότερη κατανεμημένη κατάρτιση. Η επικοινωνία μεταξύ των εξυπηρετητών (servers) είναι ο περιοριστικός παράγοντας στην κλιμάκωση της κατανεμημένης εκπαίδευσης συνελικτικών δικτύων (CNNs). Για την κατανεμημένη παράλληλη εκπαίδευση δεδομένων, ο απαιτούμενος χρόνος επικοινωνίας είναι άμεσα ανάλογος με τον αριθμό των παραμέτρων στο μοντέλο. Εν ολίγοις, μικρά μοντέλα εκπαιδεύονται γρηγορότερα λόγω της μικρότερης απαίτησης σε επικοινωνία.

Ταυτόχρονα, επιφέρουν μικρότερη επιβάρυνση κατά την εξαγωγή νέων μοντέλων σε πελάτες. Για παράδειγμα στην αυτόνομη οδήγηση, εταιρείες όπως η Tesla αντιγράφουν περιοδικά νέα μοντέλα από τους διακομιστές τους σε αυτοκίνητα πελατών. Αυτό αναφέρεται συχνά ως επικαιροποιημένη ενημέρωση. Ωστόσο, οι επικαιροποιημένες ενημερώσεις των τυπικών σημερινών μοντέλων CNN / DNN ενδέχεται να απαιτούν μεγάλες μεταφορές δεδομένων. Μικρότερα μοντέλα απαιτούν λιγότερη επικοινωνία, καθιστώντας τις συχνές ενημερώσεις πιο εφικτές [20].

Στην συγκεκριμένη διπλωματική εργασία δόθηκε ιδιαίτερη έμφαση στην μείωση των παραμέτρων και γενικότερα της μνήμης του μοντέλου ώστε να μπορεί να υλοποιηθεί σε FPGA επεξεργαστή. Οι FPGA πλατφόρμες συχνά έχουν λιγότερη από 10MB μνήμη πάνω στο ολοκληρωμένο κύκλωμα (onchip) και δεν έχουν επιπλέον μνήμη ή αποθηκευτικό χώρο. Για συμπεράσματα και προβλέψεις, ένα αρκετά μικρό μοντέλο θα μπορούσε να αποθηκευτεί απευθείας στο FPGA αντί να αφιερώνεται μεγάλο ποσοστό του χρόνου πρόβλεψης στην επικοινωνία για την ενημέρωση κάθε φορά των παραμέτρων.

4.1 Μείωση παραμέτρων εκπαίδευσης

Προκειμένου να μειώσουμε την μνήμη του μοντέλου κατασκευάσαμε την παρακάτω αρχιτεκτονική δικτύου προσθέτοντας ένα επιπλέον συνελικτικό επίπεδο, ακολουθούμενο από ένα επίπεδο συγκέντρωσης.



Εικόνα 32 Αρχιτεκτονική δικτύου με τρία συνελικτικά επίπεδα και ένα κρυφό πλήρως συνδεδεμένο επίπεδο

Από το βελτιστοποιημένο μοντέλο της προηγούμενης αρχιτεκτονικής με τα δύο συνελικτικά επίπεδα, κρατήσαμε τον αλγόριθμο βελτιστοποίησης SGD καθώς και τον βαθμό εκμάθησης στο 0.01. Οι υπερπαράμετροι προς βελτιστοποίηση για την εκπαίδευση του μοντέλου με την παραπάνω αρχιτεκτονική ήταν ο αριθμός των φίλτρων σε κάθε συνελικτικό επίπεδο. Δοκιμάστηκαν τρεις συνδυασμοί από τους οποίους τα αποτελέσματα ήταν ικανοποιητικά, όπως φαίνονται παρακάτω, με αποτέλεσμα να μην δοκιμαστούν περισσότεροι.

Αλγόριθμος βελτιστοποίησης	**Φίλτρα σε κάθε συννελικτικό επίπεδο**	**Βαθμός εκμάθησης**	**Συνάρτηση κόστους**	**Ακρίβεια (%)**	**Μνήμη (MB)**
SGD	64_32_32	0.01	0.06	98.5	16.7
SGD	64_32_16	0.01	0.06	98.25	10.5
SGD	64_64_32	0.01	0.06	98.38	16.9

Εικόνα 33 Αποτελέσματα εκπαίδευσης μοντέλου με τρία συνελικτικά επίπεδα

Παρατηρούμε στα παραπάνω αποτελέσματα ότι παραμένοντας η ακρίβεια πρόβλεψης σε υψηλά επίπεδα καταφέραμε να μειώσουμε την μνήμη του μοντέλου από 82.1 MB στα 10.5 MB, δηλαδή μείωση κατά 87% της μνήμης. Για το δίκτυο με 64, 32 και 16 φίλτρα σε κάθε συνελικτικό επίπεδο, εφαρμόσαμε αύξηση δεδομένων με τον μηχανισμό της περιστροφής διπλασιάζοντας τόσο τα δεδομένα εκπαίδευσης όσο και τα δεδομένα επαλήθευσης. Αυτό έφερε ως αποτέλεσμα μία επιπλέον αύξηση της ακρίβειας φτάνοντας στο 98.63% και συνάρτηση κόστους στο 0.04. Μελετώντας τον αριθμό των παραμέτρων σε κάθε επίπεδο του δικτύου για το μοντέλο των 10.5MB και των 82.1MB λαμβάνουμε τα παρακάτω αποτελέσματα.

Layer (type)	Output Shape	Parameters
Conv2D_1	(,64,80,80)	1792
Conv2D_2	(,32,80,80)	18464
Max_Pooling2D	(,32,40,40)	0
Dropout	(,32,40,40)	0
Flatten	(,51200)	0
Dense_1	(,200)	10240200
Dense_2	(,2)	402
	Total:	10260858

Πίνακας 6 Παράμετροι κάθε επιπέδου δικτύου με δύο συνελικτικά επίπεδα και φίλτρα 64 και 32, αντίστοιχα

Layer (type)	Output Shape	Parameters
Conv2D_1	(,64,80,80)	1792
Conv2D_2	(,32,80,80)	18464
Max_Pooling2D	(,32,40,40)	0
Dropout	(,32,40,40)	0
Conv2D_3	(,16,40,40)	4624
Max_Pooling2D	(,16,20,20)	0
Dropout	(,16,20,20)	0
Flatten	(,6400)	0
Dense_1	(,200)	1280200
Dense_2	(,2)	402
	Total:	1305482

Πίνακας 7 Παράμετροι κάθε επιπέδου δικτύου με τρία συνελικτικά επίπεδα και φίλτρα 64, 32 και 16, αντίστοιχα

Αν λάβουμε υπόψη μας ένα στρώμα συνέλιξης που αποτελείται εξ' ολοκλήρου από φίλτρα 3x3, η συνολική ποσότητα παραμέτρων σε αυτό το επίπεδο είναι (αριθμός των καναλιών εισόδου) * (αριθμός φίλτρων + 1) * (3 * 3). Για να διατηρήσουμε ένα μικρό συνολικό αριθμό παραμέτρων σε ένα συνελικτικό νευρωνικό δίκτυο, είναι σημαντικό να μειώσουμε είτε τον αριθμό των καναλιών εισόδου στα φίλτρα 3x3, είτε τον αριθμό των φίλτρων. Παρατηρούμε στα παραπάνω αποτελέσματα ότι μεγαλύτερο όγκο παραμέτρων καταλαμβάνει το πρώτο πλήρως συνδεδεμένο επίπεδο. Οι παράμετροι κάθε πλήρως συνδεδεμένου επιπέδου είναι (αριθμός φίλτρων + 1)*(διαστάσεις εισόδου). Για να διατηρήσουμε ένα μικρό αριθμό παραμέτρων, στα επίπεδα αυτά, είναι πολύ σημαντικό να μειώσουμε τις διαστάσεις εισόδου. Μια μείωση των διαστάσεων εισόδου μπορεί πολύ εύκολα να πραγματοποιηθεί με την προσθήκη ενός επιπλέον συνελικτικού επιπέδου και ταυτόχρονα με μικρό αριθμό φίλτρων. Όπως παρατηρούμε και παραπάνω η προσθήκη του συνελικτικού επιπέδου με 16 φίλτρα, αν και καταλαμβάνει χώρο μνήμης με 4624 παραμέτρους κατάφερε να επιφέρει μείωση περίπου κατά 10 φορές στον αριθμό των παραμέτρων του πρώτου πλήρως συνδεδεμένου επιπέδου και κατ' επέκταση στον συνολικό αριθμό των παραμέτρων.

4.2 Μετατροπή παραμέτρων σε τύπο δεδομένων μισής ακρίβειας

Έχουμε αναφέρει και προηγουμένως ότι είναι πολύ σημαντικό η μικρή απαίτηση σε μνήμη για την υλοποίηση του μοντέλου σε επεξεργαστή FPGA. Ένας άλλος τρόπος να μειώσουμε την μνήμη είναι χρησιμοποιώντας float16 (μισή ακρίβεια) αντί για το κοινό float32 (απλή ακρίβεια), μαζί με την κατάλληλη υποστήριξη υλικού και λογισμικού.

Σε σύγκριση με την απλή ακρίβεια, η μισή ακρίβεια καταλαμβάνει 16 bits μνήμης αντί για 32 bits, γεγονός που υποδηλώνει μικρότερο χώρο αποθήκευσης, μικρότερο εύρος ζώνης μνήμης, λιγότερη κατανάλωση ρεύματος, χαμηλότερη καθυστέρηση επικοινωνίας και υψηλότερη αριθμητική ταχύτητα. Επίσης, μικρότερη καθυστέρηση εξυπηρέτησης που σημαίνει λιγότερα απαιτούμενα μηχανήματα για την ανάπτυξη [21]. Βέβαια, οι λειτουργίες μισής ακρίβειας πρέπει να υποστηρίζονται τόσο από το επίπεδο υλικού όσο και από το λογισμικό για την επίτευξη καλής απόδοσης. Για παράδειγμα, οι περισσότεροι επεξεργαστές γενικού σκοπού, δεν είναι βελτιστοποιημένοι για πράξεις μισής ακρίβειας αλλά για απλής και διπλής ακρίβειας. Αυτό έχει ως αποτέλεσμα πράξεις με μισή ακρίβεια να απαιτούν περισσότερο χρόνο εκτέλεσης σε σχέση με της απλής ακρίβειας, όπως θα δούμε και στην επόμενη υποενότητα.

Υπάρχουν διάφορα εργαλεία στο tensorflow για κβαντοποίηση ήδη εκπαιδευμένων μοντέλων. Βέβαια, για μετατροπή από float32 σε float16 ήδη εκπαιδευμένου μοντέλου δεν βρήκαμε κάποιο εργαλείο που να είναι συμβατό με το keras μοντέλο μας. Συνεπώς, για την μετατροπή ακολουθήσαμε την εξής μεθοδολογία.

Δημιουργήσαμε ένα μοντέλο τύπου float-16 αλλά ίδιας αρχιτεκτονικής και το αποθηκεύσαμε σε μορφή .sav καλώντας την συνάρτηση ***createModelF16(L1conv,L2conv,L3conv,dropout,dense_sz,lr,loss)***. Ο κώδικας της συνάρτησης ακολουθεί παρακάτω.

```
#call with: createModelF16(64,32,16,0.1,200,0.01,'categorical_crossentropy')
def createModelF16(L1conv,L2conv,L3conv,dropout,dense_sz,lr,loss):
        K.set_floatx('float16')
        kernel_sz=(3,3)
        model = Sequential()

        model.add(Conv2D(filters=L1conv, kernel_size=kernel_sz,strides=(1,1),
             padding='same', activation='relu',data_format="channels_first",
             input_shape=(3,80,80,)))
        model.add(Conv2D(filters=L2conv, kernel_size=kernel_sz,
                  strides=(1,1), padding='same', activation='relu'))

        model.add(MaxPooling2D(pool_size=(2,2)))
        model.add(Dropout(dropout))

        model.add(Conv2D(filters=L3conv, kernel_size=kernel_sz,
                  strides=(1,1), padding='same', activation='relu'))
        model.add(MaxPooling2D(pool_size=(2,2)))
        model.add(Dropout(dropout))

        model.add(Flatten(data_format='channels_last'))

        model.add(Dense(dense_sz, activation='sigmoid'))

        model.add(Dense(2, activation='softmax'))
```

```
        optimizer = SGD(lr=lr)
        model.compile(optimizer=optimizer,loss=loss,metrics=['accuracy'])

        # save the model to disk
        filename = 'model_f16.sav'
        joblib.dump(model, filename)
        #if we want to load the model from disk
        #model = joblib.load(filename)
```

Στη συνέχεια αλλάξαμε τις παραμέτρους του νέου μοντέλου με τις παραμέτρους του ήδη εκπαιδευμένου μοντέλου μας. Η μόνη διαφορά είναι ότι μετατρέψαμε πρώτα όλες τις παραμέτρους του μοντέλου από float-32 σε float-16. Για την υλοποίηση αυτής της μετατροπής δημιουργήσαμε τις συναρτήσεις change_layer_type(layer_name,model), change_all_layers(model) και την fill_empty_model(filename_f32,filename_f16).

Η συνάρτηση ***change_layer_type(layer_name,model)*** δέχεται το όνομα ενός επιπέδου και το αντίστοιχο μοντέλο που το περιέχει. Αλλάζει τον τύπο των παραμέτρων του συγκεκριμένου επιπέδου σε float16 και το επιστρέφει σε μορφή λίστας.

Η συνάρτηση ***change_all_layers(model)*** δέχεται ένα μοντέλο και μετατρέπει τις παραμέτρους κάθε επιπέδου σε float16. Επιστρέφει μία λίστα που αποτελείται από λίστες, μία για κάθε επίπεδο του μοντέλου.

Τέλος, η συνάρτηση ***fill_empty_model(filename_f32,filename_f16)*** δέχεται τα ονόματα των αποθηκευμένων μοντέλων σε μορφή .sav . Αλλάζει τον τύπο των παραμέτρων του μοντέλου τύπου float32 σε float16 και στη συνέχεια γεμίζει με αυτές τις τιμές το μοντέλο με παραμέτρους τύπου float16. Αυτό γίνεται με χρήση των παραπάνω συναρτήσεων. Αποθηκεύει το αλλαγμένο μοντέλο και συγχρόνως το επιστρέφει για να το αξιολογήσουμε όσον αφορά την ακρίβεια και την συνάρτηση κόστους.

Ακολουθεί ο κώδικας των συναρτήσεων.

```
# Convert layer_name's parameters of model to float16
def change_layer_type(layer_name,model):
        filters = model.get_layer(layer_name).get_weights()
        new_filters = []
        new_filters.append(filters[0].astype(np.float16))
        new_filters.append(filters[1].astype(np.float16))
        return new_filters

# Implements the above function to all layers
def change_all_layers(model):
        layers = []
        for layer in model.layers:
            # check for convolutional or dense layer
            if ('conv' in layer.name) or ('dense' in layer.name):
                new_layer_f16 = change_layer_type(layer.name,model)
                layers.append(new_layer_f16)
        return layers

# fills model with parameters float16 with converted float32 parameters of
# pre-trained model ( models saved as .sav)
def fill_empty_model(filename_f32,filename_f16):
        # Set float32 as default float type
```

```
        K.set_floatx('float32')
        #load model with float32 parameters
        model_f32 = joblib.load(filename_f32)
        # convert to float16 all filter's parameters
        new_filters = change_all_layers(model_f32)

        # Set float16 as default float type
        K.set_floatx('float16')
        #load model with float16 parameters
        model_f16 = joblib.load(filename_f16)

        i=0
        for layer in model_f16.layers:
        # check for convolutional or dense layers
            if ('conv' in layer.name) or ('dense' in layer.name):
                model_f16.get_layer(layer.name).set_weights(new_filters[i])
                i=i+1

        # save the filled model to disk
        filename = 'model_f16.sav'
        joblib.dump(model_f16, filename)

        return model_f16
```

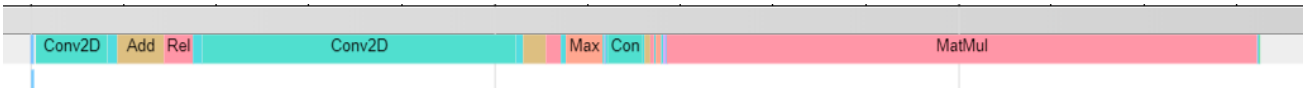
Τέλος, αξιολογήσαμε το νέο μοντέλο στα δεδομένα επαλήθευσης (test data). Από τα αποτελέσματα είδαμε ότι η μετατροπή των παραμέτρων σε μισή ακρίβεια δεν επηρέασε καθόλου την ακρίβεια και την συνάρτηση κόστους και ταυτόχρονα μείωσε την μνήμη του μοντέλου από τα 10.5MB στα 2.6MB.

4.3 Ανάλυση χρόνου πρόβλεψης των μοντέλων

Πριν προσπαθήσουμε να υλοποιήσουμε το μοντέλο με τον επεξεργαστή FPGA, είναι πολύ σημαντικό να αναλύσουμε τον χρόνο που απαιτεί κάθε λειτουργία του δικτύου για την διαδικασία της πρόβλεψης εικόνων RGB. Για την μέτρηση του χρόνου που απαιτεί κάθε λειτουργία χρησιμοποιήσαμε την λειτουργία timeline που διαθέτει το tensorflow. Η συγκεκριμένη λειτουργία καταγράφει τις λειτουργίες που εκτελούνται και τον χρόνο που απαιτούν κατά την εκπαίδευση ή στην δική μας περίπτωση κατά την πρόβλεψη εικόνων. Στη συνέχεια το αποθηκεύουμε σε JSON αρχείο και μπορούμε να το φορτώσουμε από την σελίδα “chrome://tracing” στο Google Chrome για την ανάλυση των αποτελεσμάτων. Επίσης, μετρήσαμε τον συνολικό χρόνο πρόβλεψης κάνοντας χρήση της συνάρτησης time.time() της python. Εφαρμόζοντας τα παραπάνω στα δύο μοντέλα με παραμέτρους float32 και float16, συγκεντρώσαμε τα παρακάτω αποτελέσματα.

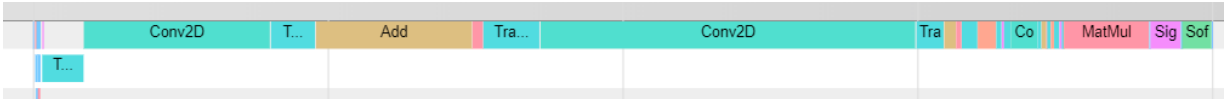
	Float32	Float16
Conv1	6.72	6.736
Conv2	8.055	15.812
MaxPooling1	0.57	1.787
Conv3	0.771	2.086
MaxPooling2	0.114	0.3
Dense1	1.971	25.496
Dense2	0.521	0.036
Total	19.857	52.824

Πίνακας 8 Απαιτούμενος χρόνος ανά λειτουργία για τα δύο μοντέλα Float32 και Float16



Εικόνα 34 Timeline από το chrome://tracing για το μοντέλο με παραμέτρους μισής ακρίβειας (float16)

Στον πίνακα 8 αναγράφεται ο απαιτούμενος χρόνος κάθε λειτουργίας κατά την διαδικασία της πρόβλεψης. Ο συνολικός χρόνος πρόβλεψης μίας εικόνας με το μοντέλο μισής ακρίβειας είναι 52.824ms. Η πιο χρονοβόρα λειτουργία είναι ο πολλαπλασιασμός στο πλήρως συνδεδεμένο επίπεδο με τους 200 νευρώνες (MatMul), η οποία καταλαμβάνει το 48% του χρόνου. Δεύτερη είναι η συνέλιξη των πυρήνων με την εικόνα στα τρία συνελικτικά επίπεδα (Conv2D), οι οποίες καταλαμβάνουν το 46% του συνολικού χρόνου. Συνολικά αυτές οι λειτουργίες καταλαμβάνουν το 94% του χρόνου που απαιτείται για την πρόβλεψη μίας εικόνας με το μοντέλο μισής ακρίβειας. Στην εικόνα 34 φαίνεται το Timeline όπως το αποτυπώνει το chrome://tracing για το συγκεκριμένο μοντέλο.

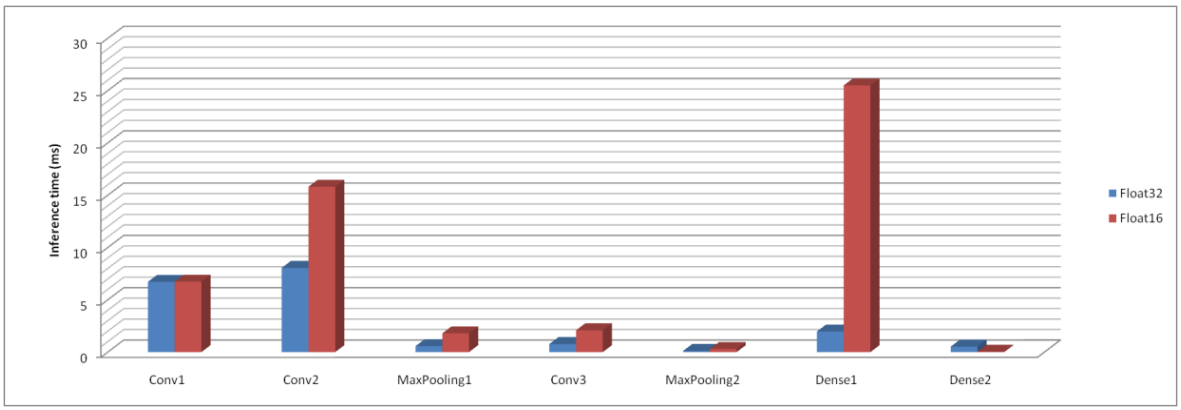


Εικόνα 35 Timeline από το chrome://tracing για το μοντέλο με παραμέτρους απλής ακρίβειας (float32)

Στον πίνακα 8 αναγράφεται, επίσης, ο απαιτούμενος χρόνος κάθε λειτουργίας κατά την διαδικασία της πρόβλεψης με το μοντέλο απλής ακρίβειας. Ο συνολικός χρόνος πρόβλεψης μίας εικόνας με το συγκεκριμένο μοντέλο είναι 19.857 ms. Σε αντίθεση με τα προηγούμενα αποτελέσματα η πιο χρονοβόρα λειτουργία είναι η συνέλιξη των πυρήνων με την εικόνα στα τρία συνελικτικά επίπεδα (Conv2D/_FusedConv2D), οι οποίες καταλαμβάνουν το 78% του συνολικού χρόνου. Στην εικόνα 35 φαίνεται το Timeline όπως το αποτυπώνει το chrome://tracing για το συγκεκριμένο μοντέλο.

Στα παραπάνω αποτελέσματα βλέπουμε ότι όσον αφορά τον πραγματικό χρόνο που απαιτούν οι λειτουργίες, το μοντέλο μισής ακρίβειας απαιτεί σχεδόν το διπλάσιο χρόνο σε σχέση με της απλής.

Αυτό μπορούμε να το δούμε καλύτερα και στο παρακάτω διάγραμμα στο οποίο αποτυπώνονται οι απαιτούμενοι χρόνοι σε ms των βασικών επιπέδων για τα δύο μοντέλα (εικόνα 36).



Εικόνα 36 Απαιτούμενοι χρόνοι σε ms των λειτουργιών για τα δύο μοντέλα

Από το συγκεκριμένο διάγραμμα μπορούμε να εξάγουμε τις εξής παρατηρήσεις.

- Ο χρόνος υλοποίησης της συνέλιξης στο δεύτερο συνελικτικό επίπεδο απαιτεί περισσότερο χρόνο σε σχέση με το πρώτο και το τρίτο επίπεδο. Αυτό συμβαίνει διότι ο αριθμός των απαιτούμενων πολλαπλασιασμών και προσθέσεων είναι πολύ μεγαλύτερος σε σχέση με τα άλλα δύο επίπεδα. Σε ένα συνελικτικό επίπεδο ο αριθμός των απαιτούμενων πολλαπλασιασμών καθορίζεται από τις διαστάσεις εξόδου (output size), τις διαστάσεις του πυρήνα (kernel_size) και τον αριθμό των φίλτρων (filters). Έτσι, περισσότερο χρόνο παίρνει το δεύτερο συνελικτικό επίπεδο με 18464 παραμέτρους και διαστάσεις εξόδου (,80,80) και 32 φίλτρα, μετά ακολουθεί το πρώτο συνελικτικό επίπεδο με 1792 παραμέτρους και διαστάσεις εξόδου (,80,80) και 64 φίλτρα και τέλος το τρίτο επίπεδο με 4624 παραμέτρους και διαστάσεις εξόδου (,40,40) και 16 φίλτρα.
- Το ίδιο συμβαίνει και για τα δύο πλήρως συνδεδεμένα επίπεδα. Το κρυφό πλήρως συνδεδεμένο επίπεδο με τους 200 νευρώνες έχει 1280200 παραμέτρους ενώ το επίπεδο εξόδου έχει μόνο 402 παραμέτρους. Συνεπώς, για την ολοκλήρωση των πράξεων του κρυφού επιπέδου απαιτείται περισσότερος χρόνος μιας και ο αριθμός των πράξεων είναι πολύ μεγαλύτερος.
- Επίσης, αξιοσημείωτο είναι το γεγονός ότι ο χρόνος για τους πολλαπλασιασμούς του πρώτου πλήρους συνδεδεμένου επιπέδου είναι κατά πολύ μεγαλύτερος στο μοντέλο μισής ακρίβειας CNN-f16 από ότι είναι στο μοντέλο απλής ακρίβειας CNN-f32. Ο λόγος που συμβαίνει αυτό είναι ότι ο επεξεργαστής ενός κοινού υπολογιστή, όπως αυτός στον οποίο έτρεξε το πρόγραμμα της πρόβλεψης, συνήθως έχει βελτιστοποιηθεί για τις πράξεις τύπου απλής ακρίβειας (float32) και διπλής ακρίβειας (float64), αφού είναι και οι πιο συνηθισμένες. Ταυτόχρονα, δεν είναι βελτιστοποιημένος για τις πράξεις μισής ακρίβειας, με αποτέλεσμα να απαιτούν πολύ περισσότερο χρόνο.

Στον παρακάτω πίνακα καταγράφεται η απαιτούμενη μνήμη για την αποθήκευση των αποτελεσμάτων κατά την κατηγοριοποίηση μίας εικόνας (υποθέτοντας μισή ακρίβεια για την αποθήκευσή τους), αλλά και ο αριθμός των πολλαπλασιασμών.

Layer (filters)	Parameters	Mults	Output Shape	Elements of Output	Total requirement memory for output (MB)
Conv2D_1 (64)	1792	11059200	64,80,80	1228800	2.4576
Conv2D_2 (32)	18464	117964800	32,80,80	39321600	78.6432
MaxPooling1	--	--	32,40,40	9830400	19.6608
Conv2D_3 (16)	4624	7372800	16,40,40	157286400	314.5728
MaxPooling2	--	--	16,20,20	39321600	78.6432
Dense1	1280200	1280000	200	1228800	2.4576
Dense2	402	400	2	12288	0.024576

Πίνακας 9 Απαιτούμενοι πολλαπλασιασμοί και μνήμη σε κάθε επίπεδο κατά την εξαγωγή των συμπερασμάτων

Στη συνέχεια μετρήσαμε τον συνολικό χρόνο πρόβλεψης μίας και εκατό εικόνων κάνοντας χρήση της συνάρτησης time.time() της python. Για να είναι οι μετρήσεις μας πιο αντικειμενικές επαναλάβαμε τις μετρήσεις των χρόνων τέσσερις φορές και κρατήσαμε τους μέσους όρους. Οι χρόνοι των δύο μοντέλων φαίνονται στον παρακάτω πίνακα.

MODEL	time.time() for 1pic (ms)	time.time() for 100pic (ms)
CNN-f16	119.177275	3687.5293
CNN-f32	100.8913	1443.239625

Πίνακας 10 Μέσοι όροι χρόνων εκτέλεσης πρόβλεψης μιας και εκατό εικόνων για τέσσερις επαναλήψεις

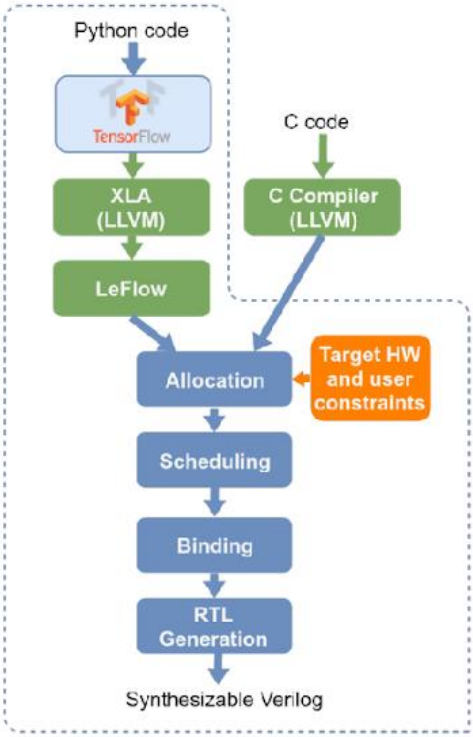
Στα αποτελέσματα παρατηρούμε ότι κατά την πρόβλεψη 100 φωτογραφιών δεν απαιτεί εκατονταπλάσιο χρόνο ολοκλήρωσης, αλλά πολύ λιγότερο αφού υλοποιούνται βελτιστοποιήσεις στον υπολογισμό των αποτελεσμάτων. Επίσης, ο χρόνος πρόβλεψης μίας εικόνας μετρημένος με την συνάρτηση time.time() είναι πιο μεγάλος σε σχέση με τον χρόνο που είδαμε ότι απαιτείται στο timeline. Αυτό συμβαίνει διότι μετριέται ο συνολικός χρόνος που είναι προς εκτέλεση σε αντίθεση με το timeine που καταγράφει μόνο τον απαιτούμενο χρόνο στον επεξεργαστή.

5 Χρήση εργαλείων υψηλού επιπέδου για τελική υλοποίηση νευρωνικού σε FPGA

Όπως αναφέραμε και προηγουμένως τα FPGAs προτιμούνται όλο και περισσότερο στον χώρο της μηχανικής μάθησης και ιδιαίτερα σε εφαρμογές με εικονικά δεδομένα. Αυτό συμβαίνει διότι η κατανάλωσή τους είναι χαμηλή, μπορούν να προσαρμόζονται κάθε φορά στην επιθυμητή εφαρμογή, καθώς επίσης οι εργασίες μπορούν να παραλληλοποιηθούν σε μεγάλο βαθμό μειώνοντας την συνολική καθυστέρηση. Σε αντίθεση με όλα αυτά τα πλεονεκτήματα έρχεται το γεγονός ότι ο προγραμματισμός τους απαιτεί μεγάλη εμπειρία και χρόνο ανάπτυξης. Αυτό προσπαθούν να το περιορίσουν τα εργαλεία που αναλύονται στην συνέχεια.

5.1 Εργαλείο LeFlow

Το LeFlow είναι ένα εργαλείο ανοικτού κώδικα το οποίο μετατρέπει αριθμητικά υπολογιστικά μοντέλα γραμμένα στο tensorflow, όπως είναι ένα συνελικτικό δίκτυο, σε μορφή που να μπορεί να γίνει σύνθεση για εφαρμογή σε επεξεργαστή FPGA.



Εικόνα 37 Προτεινόμενη ροή σε σύγκριση με την τυπική ροή HLS [55]

Το παραγόμενο αρχείο είναι σε κώδικα verilog. Η αυτόματη παραγωγή της verilog γίνεται με συνεργασία του μεταγλωττιστή υψηλού επιπέδου LegUp XLA της Google και του εργαλείου LeFlow. Παραπάνω φαίνεται η ροή εκτέλεσης του LeFlow με χρήση αρχείου σε python σε σύγκριση με την τυπική ροή HLS με χρήση αρχείου σε C κώδικα.

Το LeFlow κατασκευάστηκε για να είναι συμβατό με το LegUp 4.0., συνεπώς εγκαταστήσαμε την αντίστοιχη εικονική μηχανή. Στη συνέχεια εγκαταστήσαμε στην εικονική μηχανή το Tensorflow και το LeFlow. Το LeFlow κάνει κάποιες μικρές αλλαγές στο Tensorflow για να διασφαλίσει ότι οι πυρήνες που υποστηρίζονται από το LegUp χρησιμοποιούνται μόνο από το XLA της Tensorflow.

Οι παραπάνω εγκαταστάσεις ολοκληρώθηκαν με επιτυχία καθώς και κάποια από τα έτοιμα παραδείγματα που υπάρχουν στο αποθετήριο. Το μοντέλο που επιθυμούμε να μετατρέψουμε σε verilog, όπως αναφέρουμε παραπάνω, έχει αναπτυχθεί με Keras του Tensorflow συνεπώς απαιτείται να εγκατασταθεί και το keras στο εικονικό μηχάνημα. Για την εγκατάστασή του απαιτείται η εγκατάσταση κάποιων βιβλιοθηκών μέσα στις οποίες είναι και η βιβλιοθήκη ‘six’, της οποίας απαιτείται νεότερη έκδοση. Η συγκεκριμένη βιβλιοθήκη, όμως, είναι προστατευμένη στο εικονικό μηχάνημα και δεν μπορεί να απεγκατασταθεί προκειμένου να εγκατασταθεί η νεότερη έκδοσή της που απαιτείται για το keras. Αυτή η εξάρτηση εμπόδισε τελικά την εγκατάσταση του Keras με αποτέλεσμα να μην μπορεί να εφαρμοστεί το εργαλείο στο βελτιστοποιημένο μοντέλο των 2.6 MB και ακρίβειας 98.63%.

5.2 Εργαλείο hls4ml

Το hls4ml είναι ένα εργαλείο ανοικτού κώδικα για εξαγωγή συμπερασμάτων μηχανικής μάθησης σε FPGAs, στην δική μας περίπτωση ο εντοπισμός πλοίου σε μία εικόνα. Τα εξαγόμενα αρχεία αντιπροσωπεύουν αλγορίθμους μηχανικής μάθησης χρησιμοποιώντας γλώσσα σύνθεσης υψηλού επιπέδου (HLS). Στη συνέχεια τα αρχεία αυτά μπορούν να συνθεθούν με το πρόγραμμα Vivado HLS της Xilinx.

Ένα γνωστό ζήτημα στο Vivado HLS με την χρήση του παραπάνω εργαλείου είναι ότι κατά την σύνθεση του παραγόμενου κώδικα το ξετύλιγμα των βρόχων (loop unrolling) δημιουργεί προβλήματα μνήμης. Αυτό όπως αναφέρουμε παρακάτω αναλυτικά το ξεπεράσαμε αλλάζοντας κάποιες εντολές που απευθύνονται στον τρόπο υλοποίησης του προγράμματος σε επίπεδο υλικού.

5.2.1 Βασικές εντολές pragma

Το εργαλείο HLS παρέχει τις λεγόμενες pragma εντολές που μπορούν να χρησιμοποιηθούν για τη βελτιστοποίηση του σχεδιασμού: μείωση της καθυστέρησης, βελτίωση του ρυθμού με τον οποίο εξάγονται τα αποτελέσματα και μείωση του απαιτούμενου χώρου και των πόρων του προκύπτοντος κώδικα RTL. Αυτές οι pragma εντολές μπορούν να προστεθούν απευθείας στον πηγαίο κώδικα. Παρακάτω ακολουθεί μια σύντομη εξήγηση των εντολών που μας χρειάστηκαν.

- **Pragma hls allocation**

Καθορίζει και μπορεί να περιορίσει τον αριθμό των περιπτώσεων και των πόρων υλικού που χρησιμοποιούνται για την υλοποίηση συγκεκριμένων συναρτήσεων, βρόχων, λειτουργιών ή πυρήνων. Η εντολή pragma ALLOCATION προσδιορίζεται μέσα στο σώμα μιας συνάρτησης, ενός βρόχου ή μιας περιοχής κώδικα.

Οι λειτουργίες στον κώδικα C, όπως οι προσθέσεις, οι πολλαπλασιασμοί, το διάβασμα και γράψιμο σε πίνακα, μπορούν να περιοριστούν από την pragma ALLOCATION. Ταυτόχρονα, οι πυρήνες, στους οποίους χαρτογραφούνται οι λειτουργίες κατά τη διάρκεια της σύνθεσης, μπορούν να περιοριστούν με τον ίδιο τρόπο όπως οι συναρτήσεις. Αντί να περιοριστεί ο συνολικός αριθμός λειτουργιών πολλαπλασιασμού, περιορίζεται ο αριθμός των συνδυαστικών πυρήνων πολλαπλασιασμού, αναγκάζοντας οποιονδήποτε υπόλοιπο πολλαπλασιασμό να πραγματοποιηθεί με τους ήδη υπάρχοντες.

❖ **Pragma hls dataflow**

Η pragma DATAFLOW επιτρέπει την διοχέτευση (pipeline) σε επίπεδο εργασιών, επιτρέποντας στις λειτουργίες και τους βρόχους να επικαλύπτονται στη λειτουργία τους, αυξάνοντας την ταυτόχρονη υλοποίηση του επιπέδου μεταφοράς αρχείων (RTL) και αυξάνοντας τη συνολική απόδοση του σχεδίου.

Γενικά όλες οι λειτουργίες εκτελούνται διαδοχικά σε μια περιγραφή C. Κατά την απουσία οδηγιών που περιορίζουν τους πόρους (όπως η pragma hls allocation), το εργαλείο Vivado High-Level Synthesis (HLS) επιδιώκει να ελαχιστοποιήσει την καθυστέρηση. Ωστόσο, οι εξαρτήσεις δεδομένων μπορεί να το περιορίσουν. Για παράδειγμα, οι λειτουργίες ή οι βρόχοι που έχουν πρόσβαση σε πίνακες πρέπει να ολοκληρώσουν όλες τις προσβάσεις ανάγνωσης / εγγραφής στις συστοιχίες πριν ολοκληρωθούν. Αυτό αποτρέπει την επόμενη λειτουργία ή βρόχο που καταναλώνει τα δεδομένα από την έναρξη λειτουργίας. Η βελτιστοποίηση DATAFLOW επιτρέπει στις λειτουργίες μιας λειτουργίας ή ενός βρόχου να αρχίσουν να λειτουργούν πριν η προηγούμενη λειτουργία ή βρόχος ολοκληρώσει όλες τις λειτουργίες της.

❖ **Pragma hls stream**

Γενικά, οι μεταβλητές πινάκων υλοποιούνται ως μνήμη RAM:

- Οι παράμετροι πινάκων υψηλού επιπέδων συναρτήσεων υλοποιούνται ως θύρα διασύνδεσης RAM.
- Οι γενικοί πίνακες υλοποιούνται ως RAM για πρόσβαση ανάγνωσης και εγγραφής.
- Στις υπό-συναρτήσεις που εμπλέκονται σε βελτιστοποιήσεις DATAFLOW, οι παράμετροι πινάκων υλοποιούνται χρησιμοποιώντας ένα κανάλι RAM ping pong buffer.
- Οι πίνακες που εμπλέκονται σε βελτιστοποιήσεις DATAFLOW που βασίζονται σε βρόχο υλοποιούνται ως κανάλι προσωρινής αποθήκευσης ping pong RAM.

Εάν τα δεδομένα που αποθηκεύονται στον πίνακα καταναλώνονται ή παράγονται διαδοχικά, ένας αποτελεσματικότερος μηχανισμός επικοινωνίας είναι να χρησιμοποιεί δεδομένα ροής όπως καθορίζονται από το STREAM pragma, όπου χρησιμοποιούνται στοίβες FIFOs αντί προσωρινής μνήμης RAM.

❖ **Pragma hls pipeline**

Η εντολή αυτή μειώνει το διάστημα έναρξης (initiation interval - II) για μια λειτουργία ή βρόχο επιτρέποντας την ταυτόχρονη εκτέλεση των λειτουργιών. Η εφαρμογή διοχέτευσης σε μία λειτουργία ή βρόχο μπορεί να επεξεργαστεί νέες εισόδους σε κάθε <N> κύκλους ρολογιού, όπου <N> είναι το διάστημα έναρξης (II) του βρόχου ή της λειτουργίας. Η προεπιλεγμένη τιμή II για το pragma PIPELINE είναι 1, η οποία επεξεργάζεται μια νέα είσοδο σε κάθε κύκλο ρολογιού. Μπορούμε επίσης να ορίσουμε το διάστημα έναρξης μέσω της χρήσης της επιλογής II στην εντολή pragma.

❖ **Pragma hls unroll**

Η παραπάνω εντολή ξετυλίγει τους βρόχους για να δημιουργήσει πολλές ανεξάρτητες λειτουργίες και όχι μια ενιαία συλλογή λειτουργιών. Η pragma UNROLL μετασχηματίζει τους βρόχους δημιουργώντας πολλαπλά αντίγραφα του σώματος του βρόχου στο σχεδιασμό του επιπέδου μεταφοράς αρχείου (RTL), το οποίο επιτρέπει την πραγματοποίηση παράλληλων επαναλήψεων μερικών ή όλων των βρόχων.

Γενικά οι βρόχοι στις λειτουργίες C / C ++ δεν ξετυλίγονται. Όταν οι βρόχοι κρατούνται στην αρχική τους μορφή, η σύνθεση δημιουργεί τη λογική για μία επανάληψη του βρόχου και ο σχεδιασμός RTL εκτελεί αυτή τη λογική για κάθε επανάληψη του βρόχου στη σειρά. Ένας βρόχος εκτελείται για τον αριθμό των επαναλήψεων που καθορίζονται από τη μεταβλητή επαγωγής βρόχου. Ο αριθμός των επαναλήψεων μπορεί επίσης να επηρεαστεί από τη λογική εντός του σώματος του βρόχου (για παράδειγμα, συνθήκες διακοπής ή τροποποιήσεις σε μεταβλητή εξόδου βρόχου). Χρησιμοποιώντας την εντολή pragma UNROLL, μπορούμε να ξετυλίξουμε τους βρόχους με στόχο την αύξηση των προσβάσεων και την απόδοση των δεδομένων.

Η pragma UNROLL επιτρέπει στο βρόχο να ξετυλιχθεί πλήρως ή μερικώς. Η πλήρης εκτύλιξη του βρόχου δημιουργεί ένα αντίγραφο του σώματος βρόχου στο RTL για κάθε επανάληψη βρόχου, έτσι ώστε ολόκληρος ο βρόχος να μπορεί να τρέξει ταυτόχρονα. Για να ξετυλίξετε πλήρως έναν βρόχο, τα όρια βρόχου πρέπει να είναι γνωστά κατά το χρόνο σύνταξης. Αυτό δεν απαιτείται για μερική εκτύλιξη.

❖ **Pragma hls array partition**

Διαχωρίζει έναν πίνακα σε μικρότερους πίνακες ή μεμονωμένα στοιχεία και παρέχει τα εξής:

- Το τελικό αποτέλεσμα στο RTL είναι με πολλαπλές μικρές μνήμες ή πολλαπλούς πίνακες αντί για μία μεγάλη μνήμη.
- Αυξάνει αποτελεσματικά το μέγεθος των θυρών ανάγνωσης και εγγραφής για την αποθήκευση.
- Ενδεχομένως βελτιώνει την απόδοση του σχεδίου.
- Απαιτεί περισσότερα στιγμιότυπα ή καταχωρητές μνήμης.

5.2.2 Μεθοδολογία για σύνθεση

Πρέπει να αναφέρουμε ότι ο παραγόμενος κώδικας συνθέθηκε στο πρόγραμμα “Vivado HLS 2018.1” και για εφαρμογή του μοντέλου μας επιλέχθηκε η πλακέτα *ZYNQ-7 ZC702 Evaluation Board.* Τα χαρακτηριστικά της συγκεκριμένης πλακέτας φαίνονται παρακάτω.

Characteristics ZYNQ-7 ZC702 Evaluation Board	
Logic cells	85K
LUTs	53200
Flip-Flops	106400
Total Block RAM	4.9 MB
DSP slices	220

Πίνακας 11 Χαρακτηριστικά πόρων πλακέτας FPGA ZYNQ-7 ZC702

Αρχικά δημιουργήσαμε ένα αρχείο .yml το οποίο περιέχει διάφορα χαρακτηριστικά για την εξαγωγή του επιθυμητού κώδικα. Το όνομα του αρχείου με την αρχιτεκτονική του δικτύου σε μορφή .json. Το όνομα του αρχείου που περιέχει τα βάρη του εκπαιδευμένου μοντέλου σε μορφή .h5. Την συσκευή στην οποία επιθυμούμε να υλοποιήσουμε το τελικό κώδικα, στην δική μας περίπτωση είναι η συσκευή ZYNQ-7 ZC702 Evaluation Board με κωδικό xc7z020clg484-1. Ο τύπος της εισόδου και της εξόδου, εδώ επιλέχθηκε σειριακή υλοποίηση για εξοικονόμηση πόρων. Η περίοδος του ρολογιού επιλέχθηκε στα 10ns. Ο τύπος των βαρών για εξοικονόμηση πόρων επιλέχθηκε σταθερής υποδιαστολής ap_fixed<16,6> και τέλος ο παράγοντας επαναχρησιμοποίησης επιλέχθηκε στο 1. Παρακάτω φαίνεται ο κώδικας του αρχείου:

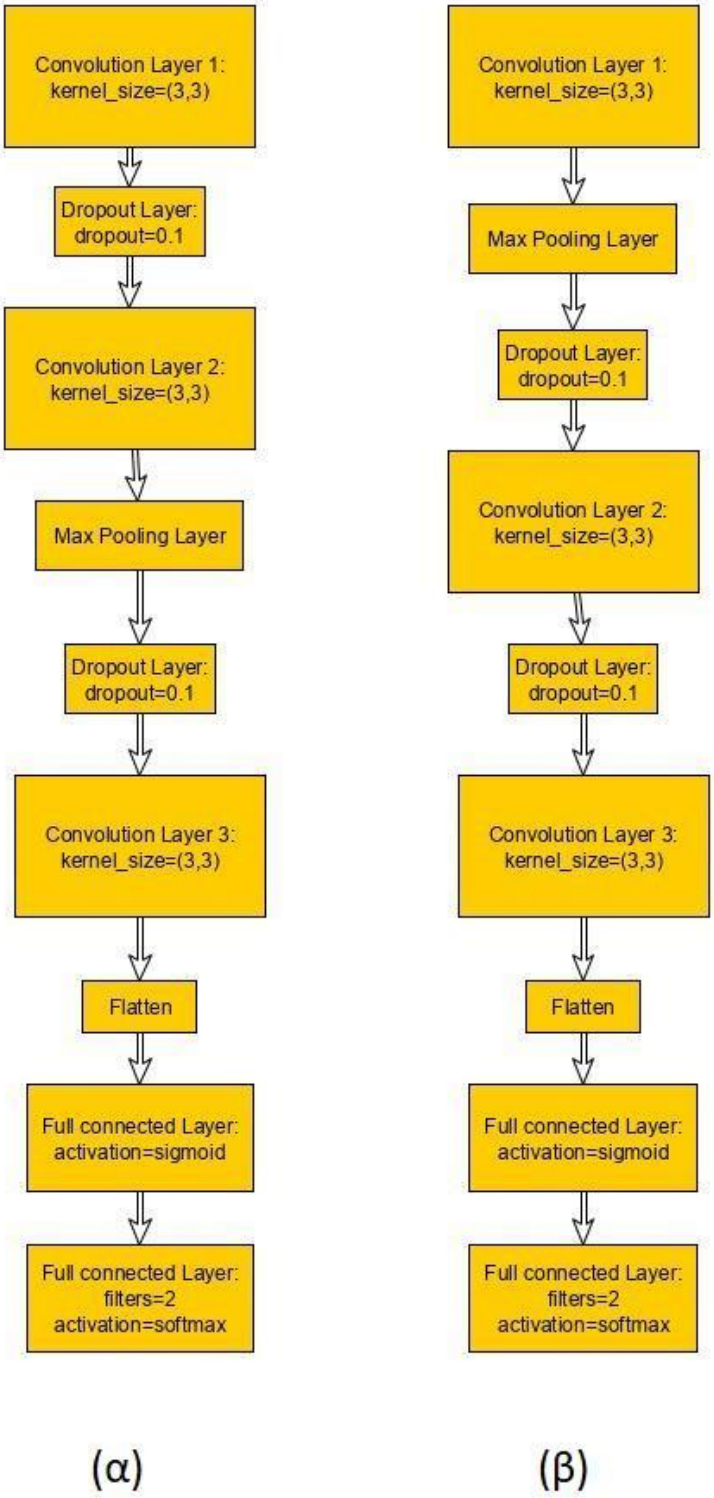
```
KerasJson: my_keras_models/model.json
KerasH5:   my_keras_models/model_weights.h5
OutputDir: dir
ProjectName: project_name
XilinxPart: xc7z020clg484-1
ClockPeriod: 10

IOType: io_serial
# options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,6>
    ReuseFactor: 1
```

Στη συνέχεια χρησιμοποιώντας το εργαλείο hls4ml εξάγαμε τον hls κώδικα από κάποια απλούστερα μοντέλα με λιγότερα επίπεδα. Παρατηρήσαμε ότι κατά την σύνθεση του παραγόμενου κώδικα το ξετύλιγμα των βρόχων (loop unrolling) δημιουργούσε πρόβλημα στην μνήμη. Το ξετύλιγμα των βρόχων γίνεται κατά την διοχέτευση ώστε να μπορέσουν να εκτελεστούν παράλληλα. Συνεπώς, αφαιρέσαμε την διοχέτευση από το αρχείο ‘nnet_conv2d.h’ που περιέχει βρόχους με τεράστιο αριθμό επαναλήψεων προκειμένου να εκτελεστούν οι συνελίξεις. Το πρόβλημα της μνήμης συνέχισε να υπάρχει στην υλοποίηση των πλήρως συνδεδεμένων επιπέδων. Συνεπώς, αφαιρέσαμε επιπλέον την εντολή pragma hls dataflow από το αρχείο ‘nnet_dense.h’ που κι αυτή απαιτεί ξετύλιγμα των βρόχων. Με αυτό τον τρόπο το πρόβλημα μνήμης λύθηκε.

Το επόμενο ζήτημα που προκλήθηκε κατά την σύνθεση ήταν ότι καθώς το vivado μετονομάζει αυτόματα κάποιες από τις συναρτήσεις δεν μπορούσε στην πορεία να αναγνωρίσει την συνάρτηση pool_op του αρχείου ‘nnet_pooling.h’. Η υλοποίηση της συγκεκριμένης συνάρτησης ήταν λίγες γραμμές και συνεπώς αντικαταστήσαμε κάθε κλήση της με τις απαραίτητες εντολές. Με αυτές τις αλλαγές η σύνθεση ολοκληρώθηκε με επιτυχία.

Καθώς προσπαθήσαμε να συνθέσουμε τον κώδικα κάποιων πιο μεγάλων αρχιτεκτονικών, παρατηρήσαμε ότι μόνο για τα μοντέλα με ένα επίπεδο μέγιστης συγκέντρωσης (max_pooling) ολοκληρωνόταν με επιτυχία η διαδικασία της σύνθεσης. Στα υπόλοιπα μοντέλα εμφανίζονταν λάθη στο δεύτερο κάλεσμα της συνάρτησης pooling2d και σε όλες τις κλήσεις που ακολουθούσαν. Συνεπώς, αυτό μας οδήγησε να δημιουργήσουμε δύο αρχιτεκτονικές δικτύου με τρία συνελικτικά επίπεδα αλλά με ένα επίπεδο συγκέντρωσης όπως απεικονίζονται παρακάτω.

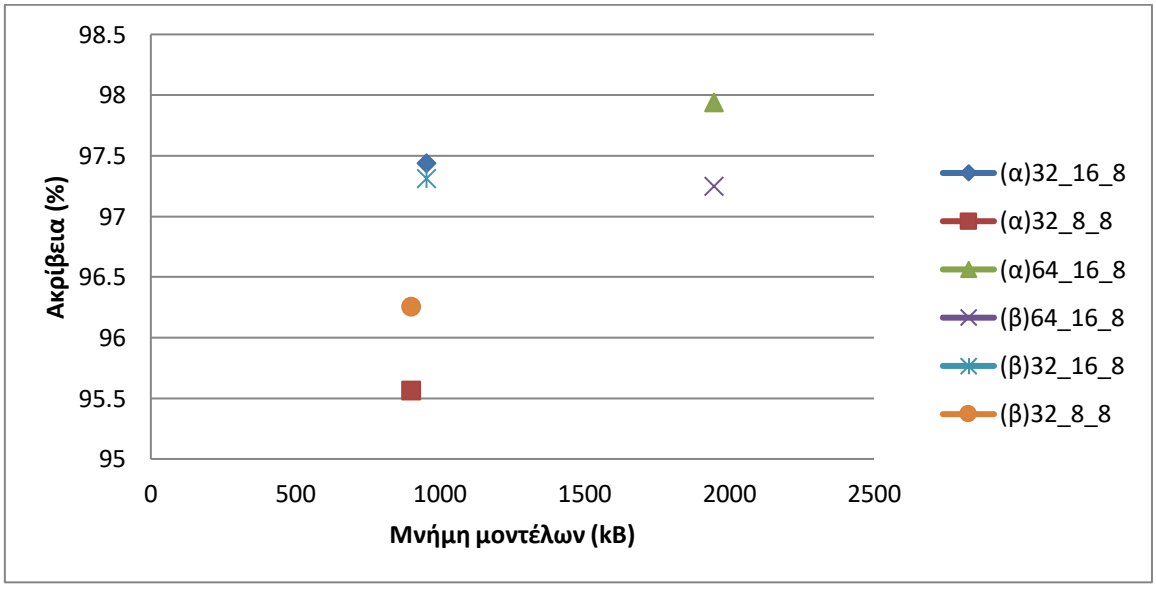


Εικόνα 38 Αρχιτεκτονικές δικτύων με ένα επίπεδο συγκέντρωσης και τρία συνελικτικά επίπεδα (α) επίπεδο συγκέντρωσης μετά το δεύτερο συνελικτικό επίπεδο, (β) επίπεδο συγκέντρωσης μετά το πρώτο συνελικτικό επίπεδο

Για τις παραπάνω αρχιτεκτονικές επιλέχθηκε ο SGD αλγόριθμος βελτιστοποίησης με 0.01 βαθμό εκμάθησης. Τα επίπεδα συγκέντρωσης εκτός ότι βοηθούν στην επιλογή των βασικών χαρακτηριστικών, μειώνουν και τις διαστάσεις εισόδου του επόμενου επιπέδου. Με την αφαίρεση ενός από τα επίπεδα συγκέντρωσης οι τελικές διαστάσεις εισόδου στο πλήρως συνδεδεμένο επίπεδο θα είναι αυξημένες. Αυτό συνεπάγεται αύξηση των παραμέτρων εκπαίδευσης και συνεπώς αύξηση της μνήμης του μοντέλου. Για αυτό τον λόγο αυξήσαμε τις διαστάσεις του επιπέδου συγκέντρωσης από [2,2] σε [3,3]. Επιπλέον, για τον ίδιο λόγο, στο πρώτο πλήρως συνδεδεμένο επίπεδο ο αριθμός των νευρώνων μειώθηκε από 200 σε 50. Οι υπερπαράμετροι προς βελτιστοποίηση ήταν ο αριθμός των φίλτρων στα συνελικτικά επίπεδα. Τα μοντέλα εκπαιδεύτηκαν για τριάντα πέντε εποχές και τα αποτελέσματα είναι συγκεντρωμένα αναλυτικά σε πίνακα αλλά και σε διάγραμμα για να μας διευκολύνει στην επιλογή του καλύτερου μοντέλου.

Αρχιτεκτονική μοντέλου	Φίλτρα 1ου συνελικτικού επιπέδου	Φίλτρα 2ου συνελικτικού επιπέδου	Φίλτρα 3ου συνελικτικού επιπέδου	Ακρίβεια (%)	Συνάρτηση κόστους	Απαιτήσεις σε μνήμη
(α)	32	16	8	97.44	0.09	951.5 kB
(α)	32	8	8	95.56	0.11	900.9 kB
(α)	64	16	8	97.94	0.07	1.9 MB
(β)	64	16	8	97.25	0.08	1.9 MB
(β)	32	16	8	97.31	0.08	951.7 kB
(β)	32	8	8	96.25	0.1	900.9 kB

Πίνακας 12 Αποτελέσματα μετά από εκπαίδευση 35 εποχών μοντέλων με νέα αρχιτεκτονική



Εικόνα 39 Ακρίβεια μοντέλων σε σχέση με την απαιτούμενη μνήμη

Για την επιλογή μοντέλου θέλουμε έναν συνδυασμό υπερπαραμέτρων με υψηλή ακρίβεια και χαμηλή μνήμη. Από το παραπάνω διάγραμμα μπορούμε πολύ εύκολα να καταλάβουμε ότι οι βέλτιστες λύσεις κατά παρέτο είναι τα μοντέλα (α)64_16_8 , (α)32_16_8 και (β)32_8_8. Επιλέγουμε λοιπόν για σύνθεση την λύση (α)32_16_8 που έχει αρκετά μικρή μνήμη 951.5kB και παράλληλα αρκετά μεγάλη ακρίβεια 97.44%.

Έτσι, προχωρήσαμε στην εφαρμογή του εργαλείου hls4ml για το συγκεκριμένο μοντέλο. Κάναμε τις αλλαγές που αναφέραμε παραπάνω στα εξαγόμενα αρχεία 'nnet_conv2d.h', 'nnet_pooling.h' και 'nnet_dense.h'. Επιπλέον, επειδή το συγκεκριμένο μοντέλο είναι αρκετά μεγάλο δημιουργήθηκε πρόβλημα με τις εντολές pragma hls partition, οι οποίες κατά την υλοποίηση της σύνθεσης ξεπερνούσαν το όριο διαίρεσης. Για αυτό το λόγο αφαιρέθηκαν από τα αρχεία 'nnet_conv2d.h' και 'nnet_dense.h'. Τέλος, συνθέσαμε τον hls κώδικα και λάβαμε τα εξής αποτελέσματα.

Latency		Interval		Type
min	max	min	max	
48600988	59660188	48600989	59660189	dataflow

Πίνακας 13 Χρονικές εκτιμήσεις απόδοσης

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4
FIFO	0	-	45	243
Instance	10252	5	2802	7171
Memory	0	-	32	13
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	10252	5	2879	7431
Available	280	220	106400	53200
Utilization (%)	3661	2	2	13

Πίνακας 14 Συνολικές απαιτήσεις σε πόρους

Instance	Module	BRAM_18K	DSP48E	FF	LUT
Block_proc_U0	Block_proc	0	0	2	11
conv_2d_U0	conv_2d	34	1	336	1076
conv_2d_1_U0	conv_2d_1	1933	1	578	1534
conv_2d_2_U0	conv_2d_2	8201	1	756	1665
dense_latency_0_0_U0	dense_latency_0_0	79	1	239	487
dense_latency_0_0_1_U0	dense_latency_0_0_1	1	1	156	415
pooling2d_U0	pooling2d	0	0	94	382
relu_U0	relu	0	0	33	150
relu_1_U0	relu_1	0	0	37	149
relu_2_U0	relu_2	0	0	38	154
sigmoid_U0	sigmoid	1	0	91	258
softmax_U0	softmax	3	0	442	890
Total	12	10252	5	2802	7171

Πίνακας 15 Απαιτήσεις σε πόρους κάθε στιγμιότυπου των συναρτήσεων που καλούνται στο κυρίως πρόγραμμα

Σύμφωνα με τις συνολικές απαιτήσεις σε πόρους καταλαβαίνουμε ότι αν και ολοκληρώθηκε η διαδικασία της σύνθεσης, δεν είναι δυνατόν η εφαρμογή του νευρωνικού με FPGA αφού απαιτεί BRAM_18K σε ποσοστό 3661% σε σχέση με τα διαθέσιμα μπλοκ. Παρατηρώντας τις απαιτήσεις σε πόρους κάθε στιγμιότυπου φαίνεται ότι την μεγαλύτερη απαίτηση σε BRAMs την έχουν οι υλοποιήσεις

των δύο τελευταίων συνελικτικών επιπέδων. Αυτό συμβαίνει διότι το πρόγραμμα αποθηκεύει στην μνήμη τα αποτελέσματα των πολλαπλασιασμών κάθε συνελικτικού επιπέδου, όπως βλέπουμε στην παρακάτω εικόνα με τις απαιτήσεις σε μνήμη του τρίτου συνελικτικού επιπέδου.

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
acc_V_U	conv_2d_2_acc_V	8	0	0	7680	16	1	122880
b2_V_U	conv_2d_2_b2_V	0	3	2	32	3	1	96
mult_V_U	conv_2d_2_mult_V	8192	32	0	5529600	16	1	88473600
w2_V_U	conv_2d_2_w2_V	1	0	0	864	9	1	7776
Total		4	8201	35	2	5538176	44	4 88604352

Πίνακας 16 Απαιτήσεις σε μνήμη τρίτου συνελικτικού επιπέδου

Μία μείωση στους πολλαπλασιασμούς μπορεί να πραγματοποιηθεί με αύξηση των διαστάσεων του επιπέδου συγκέντρωσης. Αυτή η σκέψη μας οδήγησε στην δημιουργία ενός δικτύου παρόμοιο με το προηγούμενο με την διαφορά ότι στο επίπεδο συγκέντρωσης αυξήσαμε τις διαστάσεις από [3,3] σε [4,4]. Κάνοντας τις απαραίτητες αλλαγές στα αρχεία 'nnet_dense.h', 'nnet_conv2d.h' και 'nnet_pooling.h' , προχωρήσαμε στην διαδικασία της σύνθεσης. Αυτή η αλλαγή στο μοντέλο οδήγησε στο ίδιο πρόβλημα που εμφανιζόταν με τα μοντέλα που περιέχουν δύο επίπεδα συγκέντρωσης.

Ένας άλλος τρόπος για μείωση των πολλαπλασιασμών είναι η επιπλέον μείωση των φίλτρων στα συνελικτικά επίπεδα. Καθώς επίσης η επιλογή της αρχιτεκτονικής (β), με το επίπεδο συγκέντρωσης αμέσως μετά το πρώτο συνελικτικό επίπεδο ώστε να μειωθούν οι διαστάσεις εισόδου των επιπέδων που ακολουθούν. Δημιουργήσαμε και εκπαιδεύσαμε ένα μοντέλο αρχιτεκτονικής (β) και με αριθμό φίλτρων 16,8,4 αντίστοιχα σε κάθε συνελικτικό επίπεδο. Η ακρίβεια για δέκα εποχές εκπαίδευσης ήταν 95.5%. Τα αποτελέσματα της σύνθεσης φαίνονται παρακάτω.

Latency		Interval		Type
min	max	min	max	
24301468	29831068	24301469	29831069	dataflow

Πίνακας 17 Χρονικές εκτιμήσεις απόδοσης

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4
FIFO	0	-	50	271
Instance	4204	5	2197	6246
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	4204	5	2247	6521
Available	280	220	106400	53200
Utilization (%)	1501	2	2	12

Πίνακας 18 Συνολικές απαιτήσεις σε πόρους

Instance	Module	BRAM_18K	DSP48E	FF	LUT
Block_proc_U0	Block_proc	0	0	2	11
conv_2d_U0	conv_2d	17	1	326	1070
conv_2d_1_U0	conv_2d_1	35	1	334	1082
conv_2d_2_U0	conv_2d_2	4101	1	726	1613
dense_latency_U0	dense_latency	47	1	234	505
dense_latency_1_U0	dense_latency_1	1	1	160	409
pooling2d_U0	pooling2d	0	0	94	382
relu_U0	relu	0	0	36	124
relu_1_U0	relu_1	0	0	36	124
relu_2_U0	relu_2	0	0	44	123
sigmoid_U0	sigmoid	1	0	30	217
softmax_U0	softmax	2	0	175	586
Total	12	4204	5	2197	6246

Πίνακας 19 Απαιτήσεις σε πόρους ανά στιγμιότυπο

Με τις παραπάνω αλλαγές μειώθηκε η χρησιμοποίηση των BRAMs στο 1501%. Τα τρίτο συνελικτικό επίπεδο απαιτεί 4101 αντί για 8201 BRAMs και το δεύτερο συνελικτικό επίπεδο 35 αντί για 1933 BRAMs σε σχέση με την αρχική υλοποίηση. Παρόλο την μεγάλη μείωση σε πόρους, συνεχίζει το μοντέλο να μην μπορεί να εφαρμοστεί στο FPGA.

Τα επόμενο μοντέλο που δοκιμάσαμε ήταν της ίδιας αρχιτεκτονικής (β) με 8 φίλτρα σε κάθε συνελικτικό επίπεδο. Εκπαιδεύτηκε για είκοσι εποχές και έφτασε 94.88% ακρίβεια και 0.14 συνάρτηση κόστους. Η χρησιμοποίηση των BRAMs κατέβηκε στο 760%.

Το επόμενο βήμα ήταν να μειώσουμε τα φίλτρα του δεύτερου επιπέδου στα 4 σε σχέση με το προηγούμενο νευρωνικό δίκτυο που είχε 8. Εκπαιδεύτηκε για είκοσι εποχές και έφτασε 94.44% ακρίβεια και 0.14 συνάρτηση κόστους. Η χρησιμοποίηση των BRAMs κατέβηκε στο 748%. Η τεράστια μείωση του μοντέλου επέτρεψε την προσθήκη όλων των εντολών pragma εκτός από το ξετύλιγμα βρόχων στο αρχείο ‘nnet_dense.h’. Παράλληλα στο αρχείο ‘nnet_conv2d.h’ προσθέσαμε την εντολή pragma hls dataflow για την συνάρτηση conv2d και pragma hls stream στον πίνακα mult για την αποθήκευση των πολλαπλασιασμών κατά την συνέλιξη σε FIFO αντί για BRAMs. Με αυτές τις αλλαγές οι απαιτήσεις σε πόρους είναι τέτοιες που μπορούν να καλυφθούν από τους διαθέσιμους πόρους του FPGA. Τα αναλυτικά αποτελέσματα της σύνθεσης φαίνονται παρακάτω.

Latency		Interval		Type
min	max	min	max	
4006586	6762727	4002255	6762728	dataflow

Πίνακας 20 Χρονικές απαιτήσεις απόδοσης

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4
FIFO	0	-	50	271
Instance	62	55	5295	22956
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	62	55	5345	23231
Available	280	220	106400	53200
Utilization (%)	22	25	5	43

Πίνακας 21 Συνολικές απαιτήσεις σε πόρους

Instance	Module	BRAM_18K	DSP48E	FF	LUT
Block_proc_U0	Block_proc	0	0	2	11
conv_2d_0_0_U0	conv_2d_0_0	2	1	247	896
conv_2d_0_0_1_U0	conv_2d_0_0_1	3	1	239	892
conv_2d_0_0_2_U0	conv_2d_0_0_2	3	1	352	1174
dense_latency_0_0_0_U0_s_s	dense_latency_0_0_0	0	2	278	656
dense_latency_0_0_0_3_U0	dense_latency_0_0_0_3	50	50	3450	17338
pooling2d_U0	pooling2d	0	0	92	380
relu_U0	relu	0	0	33	150
relu_1_U0	relu_1	0	0	32	150
relu_2_U0	relu_2	0	0	36	150
sigmoid_U0	sigmoid	1	0	92	269
softmax_U0	softmax	3	0	442	890
Total	12	62	55	5295	22956

Πίνακας 22 Απαιτήσεις σε πόρους ανά στιγμιότυπο

Επίσης, από την στιγμή που οι πόροι της πλατφόρμας χρησιμοποιούνται λιγότερο από το 50% προχωρήσαμε σε κάποιες επιπλέον προσθήκες pragmas εντολών στους κώδικες. Οι αλλαγές έγιναν στα αρχεία 'cnnet_conv2d.h', 'cnnet_dense.h'. Αρχικά ορίσαμε κάθε βρόχος των συναρτήσεων στα αρχεία να εκτελείται με διοχέτευση (pipeline). Επιπλέον, στην συνάρτηση conv2d και dense_latency προσθέσαμε την εντολή pragma hls inline. Η συγκεκριμένη εντολή αφαιρεί μια συνάρτηση ως ξεχωριστή οντότητα στην ιεραρχία. Η συνάρτηση ενσωματώνεται στη συνάρτηση από την οποία κλήθηκε και δεν εμφανίζεται πλέον ως ξεχωριστή οντότητα στο RTL αρχείο. Η ενσωμάτωση μιας συνάρτησης επιτρέπει την κοινή χρήση και βελτιστοποίηση των λειτουργιών εντός της συνάρτησης με τις περιβάλλουσες λειτουργίες. Τα αποτελέσματα από την σύνθεση ακολουθούν παρακάτω.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4
FIFO	0	-	50	273
Instance	59	67	5985	25604
Memory	4	-	0	0
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	63	67	6035	25881
Available	280	220	106400	53200
Utilization (%)	22	30	5	48

Πίνακας 23 Συνολικές απαιτήσεις σε πόρους

Latency		Interval		Type
min	max	min	max	
2797334	2797334	2766725	2766725	dataflow

Πίνακας 24 Χρονικές απαιτήσεις απόδοσης

5.3 Εργαλείο της Xilinx Machine Learning Suite

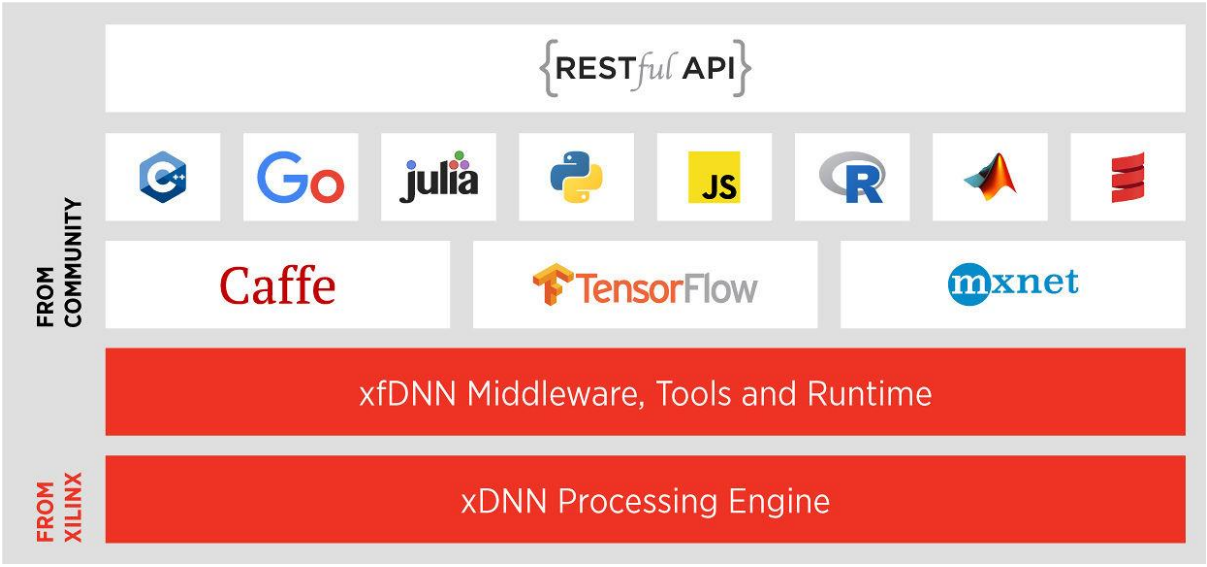
Το εργαλείο της Xilinx Machine Learning Suite (ML) παρέχει στους χρήστες τα εργαλεία για την ανάπτυξη εφαρμογών μηχανικής μάθησης για εξαγωγή συμπερασμάτων σε πραγματικό χρόνο. Παρέχει υποστήριξη για πολλά κοινά πλαίσια μάθησης μηχανών όπως το Caffe, MxNet και Tensorflow. Αποτελείται από τρία βασικά μέρη:

1. xDNN IP - Γενική μηχανή επεξεργασίας συνελικτικών νευρωνικών δικτύων (CNNs) υψηλής απόδοσης. Οι βασικές λειτουργίες που υποστηρίζει είναι οι εξής:

- Συνελικτικό επίπεδο (Convolution layer): Οι διαστάσεις του πυρήνα μπορούν να είναι οποιοσδήποτε συνδυασμός των αριθμών 1 έως 8.
- Συνάρτηση ενεργοποίησης ReLU
- Επίπεδο συγκέντρωσης (Pooling): Για επίπεδο μέγιστης συγκέντρωσης (max pooling) υποστηρίζονται πυρήνες [2,2] και [3,3]. Το επίπεδο συγκέντρωσης κατά μέση τιμή δεν υποστηρίζεται σε όλες τις εκδόσεις.

2. xfDNN Middleware - Βιβλιοθήκη λογισμικού και εργαλεία για τη διασύνδεση με ML πλαίσια και βελτιστοποίηση τους για συμπεράσματα πραγματικού χρόνου.

3. Πλαίσιο ML και Υποστήριξη ανοιχτού κώδικα



Εικόνα 40 Ακολουθία εφαρμογών και βιβλιοθηκών για την τελική εφαρμογή κάποιου μοντέλου στην πλακέτα με το εργαλείο ml-suite [56]

Όπως βλέπουμε και παραπάνω με το συγκεκριμένο εργαλείο μπορούμε να εφαρμόσουμε διάφορα μοντέλα, αναπτυγμένα με python στο tensorflow, στην πλακέτα της επιλογής μας χωρίς να χρειαστεί να αναπτύξουμε κάποιον vhdl ή verilog κώδικα.

Το μοντέλο για το οποίο δοκιμάστηκε το εργαλείο είναι το ίδιο με το τελευταίο εργαλείο που δημιουργήσαμε στην προηγούμενη υποενότητα με το εργαλείο hls4ml. Οι μόνες αλλαγές που έγιναν ήταν η αφαίρεση των επιπέδων dropout αφού δεν υποστηρίζονται και η αλλαγή της τροφοδότησης των δεδομένων στο μοντέλο από channels_first σε channels_last. Με αυτές τις αλλαγές το μοντέλο εκπαιδεύτηκε για 20 εποχές με τελική ακρίβεια 98.13% και απαιτήσεις σε μνήμη 2.2 MB. Αρχικά έπρεπε να μετατραπεί το μοντέλο σε μορφή συμβατή με το εργαλείο της xilinx, δηλαδή να αποθηκευτεί ως protobuf (.pb) αρχείο. Αυτή η μετατροπή έγινε με τον παρακάτω κώδικα.

```
import numpy as np
import tensorflow as tf
from sklearn.externals import joblib
import json
from tensorflow import keras
from keras import backend as K
from keras.optimizers import SGD
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

def freeze_session(session, keep_var_names=None, output_names=None,
clear_devices=True):
    """
    Freezes the state of a session into a pruned computation graph.

    Creates a new computation graph where variable nodes are replaced by
    constants taking their current value in the session. The new graph will be
    pruned so subgraphs that are not necessary to compute the requested
    outputs are removed.
    @param session The TensorFlow session to be frozen.
    @param keep_var_names A list of variable names that should not be frozen,
                          or None to freeze all the variables in the graph.
    @param output_names Names of the relevant graph outputs.
    @param clear_devices Remove the device directives from the graph for better
portability.
    @return The frozen graph definition.
    """
```

```
    graph = session.graph
    with graph.as_default():
        freeze_var_names = list(set(v.op.name for v in
tf.global_variables()).difference(keep_var_names or []))
        output_names = output_names or []
        output_names += [v.op.name for v in tf.global_variables()]
        input_graph_def = graph.as_graph_def()
        if clear_devices:
            for node in input_graph_def.node:
                node.device = ""
        frozen_graph = tf.graph_util.convert_variables_to_constants(
            session, input_graph_def, output_names, freeze_var_names)
        return frozen_graph

train = joblib.load("train.sav")
test = joblib.load("test.sav")
train_labels = joblib.load("train_labels.sav")
test_labels = joblib.load("test_labels.sav")

train = np.reshape(train,(6400,80,80,3))
test = np.reshape(test,(1600,80,80,3))

filename = 'ch_last8_4_8_50_lr0.01ep20.sav'
model = joblib.load(filename)

optimizer = SGD(lr=0.01)
model.compile(optimizer=optimizer,
loss='categorical_crossentropy',metrics=['accuracy'])

weights_file= 'weights.hdf5'
#  Train the model
checkpointer = ModelCheckpoint(filepath=weights_file, verbose=1,
save_best_only=True,monitor='val_acc')
model.fit(train,train_labels,validation_data=(test,test_labels),batch_size=100,epochs=
2,callbacks=[checkpointer])
model.load_weights(weights_file)

# inputs
print('inputs: ', [input.op.name for input in model.inputs])

# outputs
print('outputs: ', [output.op.name for output in model.outputs])

model.save('./my_model.h5')
K.set_learning_phase(0)
sess = K.get_session()
print("sess loaded")

frozen_graph = freeze_session(sess, output_names=[out.op.name for out in
model.outputs])
tf.train.write_graph(frozen_graph, './', 'my_model.pbtxt', as_text=True)
tf.train.write_graph(frozen_graph, './', 'my_model.pb', as_text=False)
```

Για τη εφαρμογή του python μοντέλου στην πλακέτα απαιτείται πρώτα η ολοκλήρωση κάποιων σταδίων τα οποία υποστηρίζονται από το εργαλείο. Το εργαλείο είναι διαθέσιμο σε docker container και όλα όσα θα αναφέρουμε παρακάτω ολοκληρώθηκαν σε αυτό.

Αρχικά ρυθμίσαμε το περιβάλλον της εφαρμογής να είναι συμβατό με τον επεξεργαστή alveo-u200 εκτελώντας την εντολή:

```
source $MLSUITE_ROOT/overlaybins/setup.sh alveo-u200
```

Στη συνέχεια ακολούθησε η βελτιστοποίηση του μοντέλου με το εργαλείο DECENT_Q. Αυτό απαιτείται, επειδή τα FPGAs θα επωφεληθούν από την ακρίβεια Fixed Point Precision, για να επιτύχουν μεγαλύτερο παραλληλισμό σε χαμηλότερη ισχύ. Μετά την συμπίεση του μοντέλου η ακρίβεια παρέμεινε σε υψηλό επίπεδο και ήταν 97.66%. Αυτό πραγματοποιήθηκε με την εντολή:

```
python run.py --quantize --model /opt/ml-suite/share/ML_Suit/my_model.pb --
output_dir work --input_nodes conv2d_1_input --output_nodes dense_2/Softmax -
-c_input_nodes conv2d_1_input --c_output_nodes dense_2/Softmax --input_shapes
?,80,80,3
```

Επίσης έγιναν αλλαγές στο αρχείο utils.py για να υποστηρίζει το δικό μας μοντέλο. Αλλάξαμε το όνομα της εισόδου του μοντέλου σε “conv2d_1_input” και τις δύο συναρτήσεις “input_fn” και “top5_accuracy”. Ο αλλαγμένος κώδικας ακολουθεί παρακάτω. Η πρώτη συνάρτηση φορτώνει κάθε φορά έως 300 εικόνες (λόγω του μεγάλου όγκου των δεδομένων) και τις αντίστοιχες επιθυμητές εξόδους και επιστρέφει την επιθυμητή, η οποία ορίζεται από την μεταβλητή iter. Με αυτό τον τρόπο καταφέραμε να φορτώσουμε για έλεγχο όλα τα δεδομένα ελέγχου (test data), τα οποία ήταν 1500 εικόνες. Η δεύτερη συνάρτηση υπολογίζει την ακρίβεια του μοντέλου για τόσες εικόνες όσο η μεταβλητή iter_cnt.

```
def input_fn(iter):
  if (iter <= 300):
      import test
      import test_labels
      images=test.data
      labels=test_labels.labels
  if (iter > 1200):
      import test4
      import test_labels4
      iter = iter-1200
      images=test4.data
      labels=test_labels4.labels
  if (iter > 900):
      import test3
      import test_labels3
      iter = iter-900
      images=test3.data
      labels=test_labels3.labels
  if (iter > 600):
      import test2
      import test_labels2
      iter = iter-600
      images=test2.data
      labels=test_labels2.labels
  if (iter > 300 ):
      import test1
      import test_labels1
      iter = iter-300
      images=test1.data
      labels=test_labels1.labels
  images = np.asarray(images[iter])
  images = np.reshape(images,(1,80,80,3))
  labels = np.array(labels[iter])
```

```
  if INCLUDE_LABELS:
    return {INPUT_NODES: images, 'labels': labels}
  else:
    return {INPUT_NODES: images}



def top5_accuracy(graph, input_nodes, output_nodes, iter_cnt,
batch_size, label_offset=0):
  global BATCH_SIZE, INPUT_NODES, INCLUDE_LABELS, LABEL_OFFSET
  INPUT_NODES    = input_nodes
  INCLUDE_LABELS = True
  LABEL_OFFSET   = label_offset
  BATCH_SIZE     = 1
  with tf.Session(graph=graph) as sess:
    input_tensors = {node:
sess.graph.get_operation_by_name(node).outputs[0] for node in
make_list(input_nodes)}
    output_tensor =
sess.graph.get_operation_by_name(output_nodes).outputs[0]
    top5_acc = 0
    progress = ProgressBar()

    for iter in progress(range(iter_cnt)):
      inputs = input_fn(iter)
      correct_labels = inputs['labels']
      predictions = sess.run(output_tensor, feed_dict={tensor:
inputs[name] for name, tensor in input_tensors.items()})
      top5_prediction = np.argsort(predictions, axis=1)[:,-5:]
      top5_accuracy=0
      if(top5_prediction[0][0] == correct_labels[0] and
top5_prediction[0][1] == correct_labels[1]):
          top5_accuracy=1
      top5_acc += top5_accuracy

    total_samples = float(iter_cnt)
    final_top5_acc = top5_acc/total_samples
    print ('top5_acc:{}'.format(final_top5_acc))
```

Τέλος, γίνεται χωρισμός (partition), μεταγλώττιση (compile) και εκτέλεση συμπερασμάτων (run inference). Γενικά για να εκτελέσουμε ένα μοντέλο tensorflow στο FPGA, πρέπει να μεταγλωττιστεί και να δημιουργηθεί ένα νέο γράφημα. Το νέο γράφημα είναι παρόμοιο με το πρωτότυπο, χωρίς το υπογράφημα που θα εκτελεστεί στο FPGA και αντικαθιστώντας το με ένα προσαρμοσμένο επίπεδο Python. Το μοντέλο χωρίζεται σε ένα μέρος προς εκτέλεση στην πλακέτα FPGA, συνελικτικά επίπεδα με επίπεδα συγκέντρωσης, και σε ένα δεύτερο μέρος προς εκτέλεση στην CPU, πλήρως συνδεδεμένα επίπεδα. Όλα αυτά γίνονται με την εντολή:

```
python run.py --validate --model /opt/ml-suite/share/ML_Suit/my_model.pb --
output_dir work --input_nodes conv2d_1_input --output_nodes dense_2/Softmax -
-c_input_nodes conv2d_1_input --c_output_nodes dense_2/Softmax --input_shapes
?,80,80,3
```

Στο παραπάνω βήμα εκτελέστηκαν όλα τα στάδια με επιτυχία μέχρι πριν την σύνδεση με την πλακέτα alveo-u200. Το πρόβλημα που συναντήσαμε ήταν ότι το συγκεκριμένο εργαλείο τρέχει κατευθείαν στην πλακέτα μιας και έχει pre-build τα αρχεία που αντιπροσωπεύουν τα διάφορα επίπεδα των νευρωνικών. Προκειμένου να παραχθούν τα συγκεκριμένα αρχεία χρησιμοποιήθηκε το XRT 2018.2. Συνεπώς, για να

γίνει με επιτυχία η σύνδεση στην πλακέτα και να τρέξει το νευρωνικό μας θα πρέπει να έχουμε την συγκεκριμένη έκδοση. Το εργαστήριο, όμως, έχει την ενημερωμένη έκδοση του 2019 και συνεπώς δεν μπορεί να πραγματοποιηθεί με επιτυχία η εξαγωγή των συμπερασμάτων από την πλατφόρμα alveo-u200 με το εργαλείο ML-Suite.

5.4 Σύγκριση με αποτελέσματα προηγούμενων διπλωματικών εργασιών

Πριν προχωρήσουμε στην σύγκριση των αποτελεσμάτων με προηγούμενες διπλωματικές εργασίες παραθέτουμε τον παρακάτω πίνακα με την σύγκριση των τριών εργαλείων που χρησιμοποιήσαμε.

Εργαλείο	Χρόνος ενασχόλησης	Απαιτούμενες γνώσεις	Κατηγορία εργαλείου	Λειτουργικότητα
LeFlow	2 εβδομάδες	Βασικές γνώσεις για virtual machines	Ανοικτού κώδικα	Δεν υποστηρίζει μοντέλα αναπτυγμένα σε keras του tensorflow
Hls4ml	2 εβδομάδες	Βασικές γνώσεις HLS για μετατροπές στον παραγόμενο κώδικα Βασικές λειτουργίες του προγράμματος Vivado HLS της xilinx	Ανοικτού κώδικα	Υποστηρίζει keras μοντέλα, απαιτούνται αλλαγές στους εξαγόμενους κώδικες
Xilinx ML Suit	2 εβδομάδες	Βασικές γνώσεις χρήσης εφαρμογών σε κοντέινερ (docker container)	Εμπορικό	Υποστηρίζει keras μοντέλα σε μορφή pb αρχείου, απαιτεί νεότερη έκδοση XRT

Πίνακας 25 Σύγκριση τριών εργαλείων

Στη συνέχεια, εξετάσαμε τα αποτελέσματά μας με δύο προηγούμενες διπλωματικές εργασίες που είχαν ασχοληθεί επίσης με εφαρμογή συνελικτικών μοντέλων σε ειδικού σκοπού επεξεργαστές. Στην πρώτη διπλωματική εργασία αναπτύχθηκαν FPGA αρχιτεκτονικές σχεδιασμένες σε VHDL για την υλοποίηση συνελικτικών νευρωνικών δικτύων χρησιμοποιώντας αποκλειστικά την μνήμη της προγραμματιζόμενης λογικής. Ταυτό χρονα κατασκευάστηκε το "Modified Cifar-10 Full", ένα βαθύ συνελικτικό νευρωνικό δίκτυο λίγων bit, και εκπαιδεύτηκε με το Caffe framework σε εικόνες του SAT-6 airborne dataset [35]. Η δεύτερη διπλωματική εργασία είχε στόχο την ανάπτυξη ενός συστήματος εκτέλεσης συνελικτικών νευρωνικών δικτύων, για τον ενσωματωμένο πολυεπεξεργαστή Myriad2 [34]. Αυτό το πέτυχε αναπτύσσοντας κάθε στοιχείο ενός συνελικτικού δικτύου σε συμβολική γλώσσα (assembly). Από την δεύτερη διπλωματική για την σύγκριση με το δικό μας μοντέλο πήραμε τα αποτελέσματα για είσοδο 80,80,3 και μοντέλο το “CIFAR 10 Quick CNN” που είναι παρόμοιο με το μοντέλο της πρώτης διπλωματικής.

Board	ZYNQ-7 ZC702		Myriad2 SoC
Model	(b)8_4_8 , with inline and pipeline	Modified Cifar 10 Full CNN	CIFAR 10 Quick CNN
LUTs (%)	48	75.81	--
Slice Registers-FF (%)	5	52.36	--
RAM Blocks (%)	22	57.12	--
DSP (%)	30	99.09	--
Clock (ns)	10	10	--
Frequency (MHz)	100	100	--
Latency (ms)	27.9	0.551	3.6
Throughput (Images/sec)	35.84229391	4650	277.8
Input Size (RGB images)	80,80,3	28,28,3	80,80,3
Total input pixels	19200	2352	19200
Throughput (Mpixels/sec)	0.688172043	4.268602541	5.333333333
Word Length	16 fixed point	I/O: 4bits Conv: 8bits FC: 2bits	16-bit floating point
CNN Accuracy (%)	94.44	94.89	--

Πίνακας 26 Σύγκριση αποτελεσμάτων του δικού μας μοντέλου, που εξάγαμε με το εργαλείο hls4ml, με τα αποτελέσματα δύο παλαιότερων διπλωματικών

Παρατηρώντας τα αποτελέσματα καταλαβαίνουμε ότι από άποψη χρόνου εκτέλεσης δεν πλησιάζει τους χρόνους των υπόλοιπων υλοποιήσεων. Αυτό όμως που πετύχαμε ήταν να βελτιστοποιήσουμε ως προς μνήμη το μοντέλο με το Keras του TensorFlow, να το εφαρμόσουμε στην πλακέτα χρησιμοποιώντας αποκλειστικά την μνήμη της προγραμματιζόμενης λογικής και σε πολύ λίγο χρόνο ανάπτυξης (περίπου μισό μήνα). Παρόλο τους μεγάλους χρόνους εκτέλεσης πιστεύουμε ότι στο άμεσο μέλλον τέτοιου είδους εργαλεία θα καταφέρουν να φτάσουν σε επίπεδο να ανταγωνίζονται τους χρόνους εκτέλεσης FPGA εφαρμογών αναπτυγμένες με τους παραδοσιακούς τρόπους.

6. Συμπεράσματα

Με το πέρασμα των χρόνων όλο και περισσότερες εφαρμογές αναπτύσσονται με γνώμονα την μηχανική μάθηση. Προκειμένου μία εφαρμογή να είναι επιτυχημένη θα πρέπει να έχει χαμηλή καθυστέρηση, χαμηλή κατανάλωση ενέργειας και παράλληλα υψηλή προσαρμοστικότητα. Έτσι, η χρήση των FPGAs σε τέτοιου είδους εφαρμογές αποτελεί την καλύτερη επιλογή.

Στην συγκεκριμένη διπλωματική εργασία, λόγω του περιορισμένου αριθμού πόρων των FPGA, κληθήκαμε να κατασκευάσουμε συνελικτικά νευρωνικά δίκτυα με χαμηλές απαιτήσεις σε μνήμη και παράλληλα κρατώντας σε υψηλά επίπεδα την ακρίβεια. Ο σχεδιασμός των νευρωνικών δικτύων έγινε με την βοήθεια του keras στο tensorflow. Χρησιμοποιήθηκαν και εκπαιδεύτηκαν δεδομένα από δύο διαγωνισμούς της kaggle.Ο πρώτος διαγωνισμός λέγεται “Statoil iceberg classifier challenge” και αποσκοπεί στην κατηγοριοποίηση των εικόνων σε αυτές που περιέχουν παγόβουνα και σε αυτές που περιέχουν πλοία [3]. Για τα συγκεκριμένα δεδομένα κατασκευάστηκε συνελικτικό νευρωνικό δίκτυο με ακρίβεια 88.47% και μνήμης 15MB.Τα δεδομένα αντιπροσώπευαν σήματα από ραντάρ οπότε δεν είχαμε το κατάλληλο υπόβαθρο για να τα εκμεταλλευτούμε πλήρως και συνεπώς επικεντρωθήκαμε στα δεδομένα αναγνώρισης πλοίων. Ο δεύτερος διαγωνισμός λέγεται “Ships in Satellite Imagery” και αποσκοπεί στην αναγνώριση πλοίων από δορυφορικές εικόνες [4]. Για αυτά τα δεδομένα καταφέραμε να κατασκευάσουμε συνελικτικό νευρωνικό δίκτυο ακρίβειας 98.63% και μνήμης 2.6MB. Αυτό επιτεύχθηκε σχεδιάζοντας μία αρχιτεκτονική με περιορισμένο αριθμό παραμέτρων προς εκπαίδευση και αλλάζοντας την ακρίβεια των βαρών από απλή ακρίβεια (float32) σε μισή ακρίβεια (float16). Αξιοσημείωτο είναι ότι με την αλλαγή του τύπου των βαρών η ακρίβεια πρόβλεψης δεν επηρεάστηκε καθόλου, ενώ η μνήμη του μοντέλου μειώθηκε σε ικανοποιητικό βαθμό.

Ένα μεγάλο ζήτημα στην υλοποίηση νευρωνικών μοντέλων με FPGAs είναι ότι απαιτείται πολύ μεγάλος χρόνος ανάπτυξης καθώς και μεγάλη εμπειρία από την μεριά των προγραμματιστών. Για την επίλυση αυτών των ζητημάτων μεγάλο ερευνητικό ενδιαφέρον έχει προσανατολιστεί στην δημιουργία εργαλείων για την εύκολη μετατροπή νευρωνικών μοντέλων σε μορφή που να μπορεί να υλοποιηθεί σε πλακέτα FPGA. Συνεπώς, στο δεύτερο κομμάτι της διπλωματικής, δοκιμάστηκαν τρία τέτοιου είδους εργαλεία. Τα δύο πρώτα ανοικτού κώδικα εργαλεία ήταν το LeFlow που παράγει κώδικα verilog [22] και το hls4ml που παράγει κώδικα hls [23]. Το τρίτο εμπορικό εργαλείο ήταν το ML-Suite της Xilinx. Τελικά, μεγαλύτερη έμφαση δόθηκε στο εργαλείο hls4ml με το οποίο καταφέραμε να εφαρμόσουμε το μοντέλο στην πλακέτα ZYNQ-7 ZC702. Λόγω της φύσης του εργαλείου, σε πολύ μικρό χρόνο ανάπτυξης (περίπου μισό μήνα), πετύχαμε να υλοποιήσουμε το νευρωνικό δίκτυο χρησιμοποιώντας αποκλειστικά την μνήμη της προγραμματιζόμενης λογικής και με τελικό throughput 35 Images/sec.

Τέλος, μελετήσαμε παλαιότερες υλοποιήσεις συνελικτικών νευρωνικών δικτύων στην πλακέτα ZYNQ-7 ZC702 και στην Myriad-2 SoC, οι οποίες αναπτύχθηκαν με χρήση vhdl και assembly. Αν και πετύχαιναν καλύτερο throughput, ο χρόνος ανάπτυξής τους ήταν πολύ μεγαλύτερος από τον δικό μας. Βέβαια, παρόλο τους μεγάλους χρόνους εκτέλεσης πιστεύουμε ότι στο άμεσο μέλλον τέτοιου είδους εργαλεία θα καταφέρουν να φτάσουν σε επίπεδο να ανταγωνίζονται τους χρόνους εκτέλεσης FPGA εφαρμογών αναπτυγμένες με τους παραδοσιακούς τρόπους.

FPGA-oriented deep learning for earth observation image classification

1. Introduction

The satellites record a huge amount of information every day. Satellite imagery can provide useful information on various market segments. In particular, they can be used in agriculture, the economy but also for protection by the early discovery of fire, oil spill, icebergs, etc. This flood of new imagery is outgrowing the ability for organizations to manually look at each image that gets captured, and there is a need for machine learning and computer vision algorithms to help automate the analysis process.

In this thesis, data from two kaggle competitions were used and trained. The first competition is called "Ships in Satellite Imagery" and aims to identify ships from satellite images [4]. The second competition is called the "Statoil iceberg classifier challenge" and aims to identify if a remotely sensed target is a ship or iceberg [3]. What we are interested in is the design of neural networks with Keras in TensorFlow to maximize accuracy, optimize memory requirements of the models and time profile the inference.

At the same time, many studies have shown that the use of FPGA processors in the field of machine learning has led to a huge acceleration of applications [24, 25, 26]. The most common way to create a neural network is by using a high-level tool such as TensorFlow, Keras or Caffe frameworks using python. Therefore, to synthesize the model and implement it to the platform requires manual translation of the network components in C / C++ or HDL. This process is very time consuming and requires experience, limiting the application of FPGAs in this field. So much research interest has turned to create tools that transform pre-trained models, created with TensorFlow, Keras, Caffe or some other framework, into one of the design languages mentioned above. Recently, several such commercial and open source tools have been developed. In this thesis, we have tested three such tools. Two open-source tools LeFlow [22] and hls4ml [23] were first tested and then the commercial tool Machine Learning Suite from Xilinx [22, 23, 27].

2. Background

2.1 Layers of Convolutional Neural Network

Convolution layer

The convolution layer is the core building block of a convolutional neural network or CNN. This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. The kernel is spatially smaller than an image but is more in-depth. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.

During the forward pass, the kernel slides across the height and width of the image, producing the image representation of that receptive region. This produces a two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.

Pooling Layer

The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually. There are several pooling functions but the most popular process is max pooling, which reports the maximum output from the neighborhood.

Dropout Layer

Deep neural networks with a large number of parameters are very powerful machine learning systems. However, over-fitting is a serious problem in such networks. That occurs because models learn training data to such an extent that it cannot be generalized to unknown data [15]. Dropout is a technique to deal with this problem. The basic idea is to randomly reject neurons (along with their connections) from the entire network during training. This prevents the neurons from adapting too much to the specific training data and at the same time, we receive a sparser network [16].

Fully Connected Layer

Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular Fully Connected Neural Networks or FCNNs. This is why it can be computed as usual by a matrix multiplication followed by a bias effect. The FC layer helps map the representation between the input and the output.

Activation Functions

Since convolution is a linear operation and images are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map. These layers are called activation functions. There are several activation functions but the most popular are:

- Sigmoid: The sigmoid non-linearity has the mathematical form $f(\kappa) = 1/(1+e^{-\kappa})$. It takes a real-valued number and "squashes" it into a range between 0 and 1.
- Tanh: Tanh has the mathematical form $f(x)=2(1 + e^{-2x})^{-1} – 1$ and squashes a real-valued number to the range [-1, 1]. Like sigmoid, the activation saturates, but—unlike the sigmoid neurons—its output is zero centered.
- Relu: The Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function $ƒ(\kappa)=\max(0,\kappa)$. In other words, the activation is simply a threshold at zero.

Optimizer Algorithms

Whenever a neural network completes a batch of training and generates prediction results, it has to decide how to use the difference between the results it received and the values it knows are true to properly adjust the weights to the nodes. The algorithm that defines this step is known as the optimization algorithm [10]. In our models, we use three optimization algorithms: SGD (Stochastic Gradient Descent), RMSprop (Root Mean Square Propagation) and Adam (Adaptive Moment Estimation).

2.2 Keras framework in TensorFlow

TensorFlow is a free and open-source software library for data flow and differentiable programming across a range of tasks. Humanly speaking, TensorFlow is an open-source library for building Machine learning models at large scale. It is by far the most popular library for building deep learning models. It also has the strongest and a huge community of developers, researchers, and contributors [29]. TensorFlow builds a graph for every kind of calculation, from adding two numbers, to building a complex convolutional network. Once the graph is created, it runs in a so-called session.

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano [28]. Keras is being used for easy and fast prototyping (through user-friendliness, modularity, and extensibility). Also, keras supports convolutional networks CNNs and recurrent networks RNNs, as well as combinations of the two. Moreover, it is very important the fact that runs seamlessly on CPU and GPU. When running with a CPU, TensorFlow is wrapped with a low-level tensor library called Eigen. When running on a GPU, TensorFlow is linked to a library of well-optimized deep learning features called the NVIDIA CUDA Deep Neural Network (cuDNN) library. Some advantages of Keras framework are:

- User-friendliness. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

- Modularity. A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as few restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions and regularization schemes are all standalone modules that you can combine to create new models.
- Easy extensibility. New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
- Work with Python. No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

The integration of Keras into TensorFlow created a high-level application tf.keras for the construction of neural networks and other machine learning models [30]. This provides all the benefits of Keras and also allows us to access all the low-level TensorFlow classes. All of these make keras a very popular application in the research community and this is the reason that we have chosen tf.keras framework for the development of our neural networks.

2.3 Introduction to FPGAs

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to the desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application-Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM based which can be reprogrammed as the design evolves [36]. Every FPGA chip is made up of a finite number of predefined resources with programmable interconnects to implement a reconfigurable digital circuit and I/O blocks.

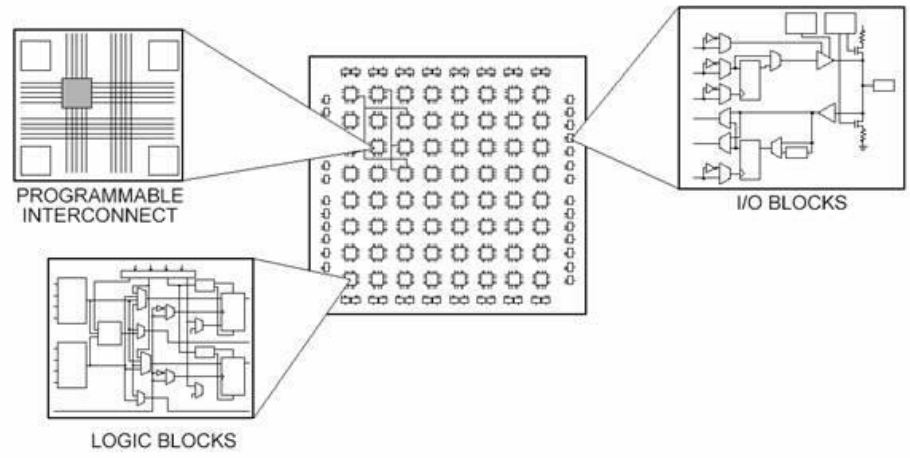


Figure 1 Different recourses of an FPGA platform [52]

The configurable logic blocks (CLBs) are the basic logic unit of an FPGA. Sometimes referred to as slices or logic cells, CLBs are made up of two basic components: flip-flops and lookup tables (LUTs). Various FPGA families differ in the way flip-flops and LUTs are packaged together.

Flip-flops are binary shift registers used to synchronize logic and save logical states between clock cycles within an FPGA circuit. On every clock edge, a flip-flop latches the 1 or 0 (TRUE or FALSE) value on its input and holds that value constant until the next clock edge.

Much of the logic in a **CLB** is implemented using very small amounts of RAM in the form of Lookup Tables or LUTs. It is easy to assume that the number of system gates in an FPGA refers to the number of NAND gates and NOR gates in a particular chip. But, in reality, all combinatorial logic (ANDs, ORs, NANDs, XORs, and so on) is implemented as truth tables within LUT memory. A truth table is a predefined list of outputs for every combination of inputs.

FPGAs have prebuilt multiplier circuitry to save on LUT and flip-flop usage in math and signal processing applications. Many signal processing algorithms involve keeping the running total of numbers being multiplied, and, as a result, higher-performance FPGAs like Xilinx Virtex-5 FPGAs have prebuilt multiplier-accumulate circuitry. These prebuilt processing blocks, also known as **DSP48** slices, integrate a 25-bit by 18-bit multiplier with adder circuitry.

Memory resources are another key specification to consider when selecting FPGAs. User-defined **RAM**, embedded throughout the FPGA chip, is useful for storing data sets or passing values between parallel tasks. Depending on the FPGA family, you can configure the onboard RAM in blocks of 16, 18 or 36 kb. You still have the option to implement data sets as arrays using flip-flops. However, large arrays quickly become expensive for FPGA logic resources. The inherent parallel execution of FPGAs allows for independent pieces of hardware logic to be driven by different clocks. Passing data between logic running at different rates can be tricky, and onboard memory is often used to smooth out the transfer using first-in-first-out (**FIFO**) memory buffers.

2.4 Challenges in implementation of a neural network to an FPGA

Many studies have shown that the use of FPGA processors in the field of machine learning has led to a huge acceleration of applications [24, 25, 26]. This is because FPGAs due to their reprogramming nature can be adapted to the desired application at any time. They can parallelize tasks which reduces delay. At the same time, all of this can be implemented with much lower energy consumption.

Of course, when implementing neural networks with FPGA there are challenges and limitations that we need to keep in mind. The first challenge concerns the FPGA memory available. FPGAs often have less than 10 MB on-chip memory and have no additional memory or storage. This makes it very difficult to design a network that satisfies the memory requirements on the one hand and achieves high prediction accuracy on the other. Therefore, we focus on the design of small convergent models, with a small number of parameters to be trained and at the same time the network architecture and parameters to be suitable for our data in order to achieve high accuracy. Our ultimate goal is to design a model that meets the FPGA's memory requirements and does not need to devote a great deal of predictive time to communication to update each parameter.

A second challenge when implementing neural networks in FPGAs is the limited resources of the application development platform. Depending on the network that we want to implement, the elements of the neuron to be implemented should be appropriately set for low latency and energy consumption.

Finally, a very big challenge is the implementation of neuronal in HDL. The most common way to create a neural network is by using a high-level tool such as TensorFlow or Keras in python. Therefore, to design the model and implement it with FPGA requires the development of network layers in either C / C ++ or HDL, as mentioned above. This process is has increased development time and requires experience limiting the application of FPGAs in this field. So much research interest has turned to create tools that transform created models, with TensorFlow, Keras, Caffe or some other framework, into one of the design languages mentioned above. Recently, several such commercial and other open-source tools have been developed. In this thesis, we have tested three such tools. Two open-source tools were first tested, LeFlow [22] and hls4ml [23] and then the Xilinx Machine Learning Suite commercial tool [22, 23, 27].

3. Iceberg classification

An iceberg is a large piece of ice that has been cut off by a glacier. Each iceberg can have a different shape and size. Because most of the icebergs lie beneath the surface of the water, it is engulfed by ocean currents. This poses risks to the navigation and infrastructure of ships. Currently, many companies and institutions are using aircraft to track ice skating onboard routes. However, in remote areas with particularly adverse weather conditions, these methods are not feasible and the only viable monitoring option is through satellites [3], [5], [6].

The competition data was given in JSON format. The files consist of a list of 1604 images, and for each image, you can find the following fields:

- id - the id of the image
- band_1, band_2 – the flattened image data. Each band has 75x75 pixel values in the list, so the list has 5625 elements. Note that these values are not the normal non-negative integers in image files since they have physical meanings - these are float numbers with unit being dB. Band 1 and Band 2 are signals characterized by radar backscatter produced from different polarizations at a particular incidence angle. The polarizations correspond to HH (transmit/receive horizontally) and HV (transmit horizontally and receive vertically).
- inc_angle - the incidence angle of which the image was taken. This field has missing data marked as "na" so we didn't use it to the training input.
- is_iceberg - the target variable, set to 1 if it is an iceberg, and 0 if it is a ship.

In order to train our neural networks, we split the data into training data (80% - 1283 images) and test data (20% - 321 images). We also apply two of the mechanisms of augmentation data rotation and flip. This resulted in a threefold increase in training and test data but did not lead to a further increase in the accuracy of the models.

When training a neural network, there are several hyper-parameters for optimization, including the number of hidden layers, the number of neurons for each filter, the learning rate, the activation function and the optimization algorithm. Creating the best combination of such hyperparameters is a difficult task. In this thesis, we try to optimize a fully connected neural network and then a convolutional neural network.

3.1 Exploration fully connected neural networks

For these data, we created three neural networks of three, four and five fully connected layers. We trained the models for various hyper-parameter combinations on the number of filters in each hidden layer, the activation function and the optimization algorithm. The activation functions tested were sigmoid and tanh. Accordingly, the optimization algorithms tested were SGD, Adam and RMSprop. The results are shown in the table below.

Optimizer	Number of layers	Filters in each layer	Activation Function	Loss Function	Accuracy (%)
Adam	3	100_50_2	sigmoid	0.7	49.22
Adam	3	300_100_2	sigmoid	0.7	49.22
SGD	3	100_50_2	sigmoid	0.71	48.29
SGD	3	300_80_2	sigmoid	0.66	63.24
RMSprop	3	100_50_2	sigmoid	0.7	49.22
RMSprop	3	250_90_2	sigmoid	0.7	49.22
SGD	**3**	**300_80_2**	**tanh**	**0.53**	**74.14**
SGD	3	300_50_2	tanh	0.61	65.42
SGD	3	600_200_2	tanh	0.78	60.44
Adam	4	500_200_80_2	tanh	0.7	54.21
SGD	4	500_200_80_2	tanh	0.64	67.6
RMSprop	4	500_200_80_2	tanh	0.76	45.79
SGD	4	400_150_50_2	tanh	0.58	69.16
SGD	4	600_200_50_2	tanh	0.55	70.4
Adam	5	200_100_60_30_2	sigmoid	0.69	54.21
SGD	5	200_100_60_30_2	sigmoid	0.69	54.21
RMSprop	5	200_100_60_30_2	sigmoid	0.69	54.21
Adam	5	500_200_100_50_2	sigmoid	0.69	54.21
SGD	5	500_200_100_50_2	sigmoid	0.69	54.21
RMSprop	5	500_200_150_50_2	tanh	0.69	64.17
SGD	5	400_300_200_70_2	tanh	1.51	48.29
SGD	5	600_400_200_70_2	tanh	0.56	69.47

Table 1 Training results of full connected neural networks

From the above results the higher accuracy achieved by the model with SGD optimizer, three full connected layers, 300 and 80 filters in each hidden layer, and tanh activation function.

3.2 Exploration of convolutional neural networks

Three network architectures were selected for convolutional neural network training. The architectures are shown below.

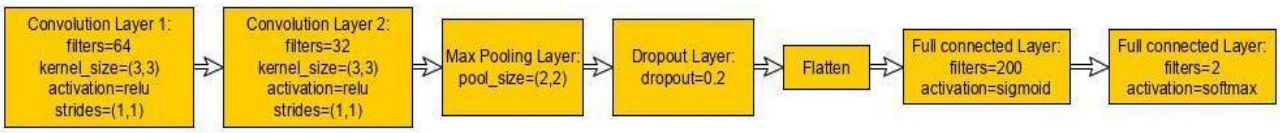


Figure 2 Architecture with two convolutional layers

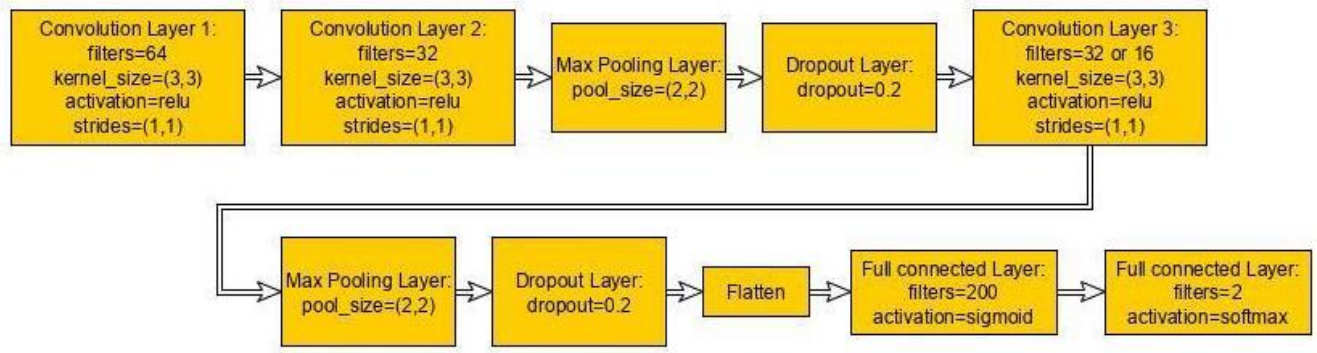


Figure 3 Architecture with three convolutional layers

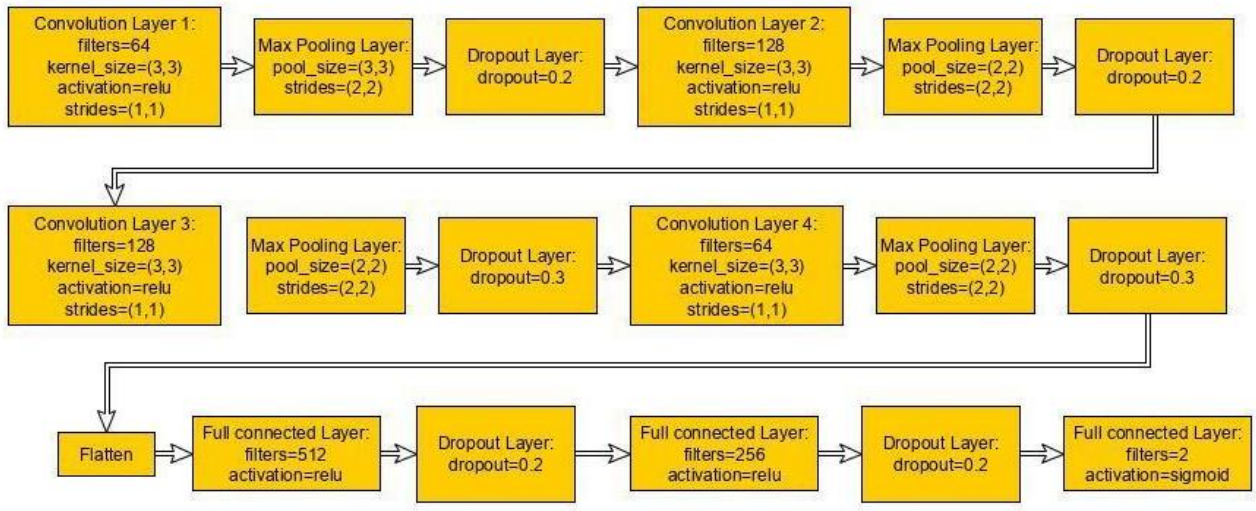


Figure 4 Architecture with four convolutional layers

This time the hyper-parameters for optimization were the learning rate (lr), the optimization algorithm and the number of filters at certain convolutional layers. The other values of the parameters remained constant, as shown in the pictures above. The results are shown in the diagrams below.

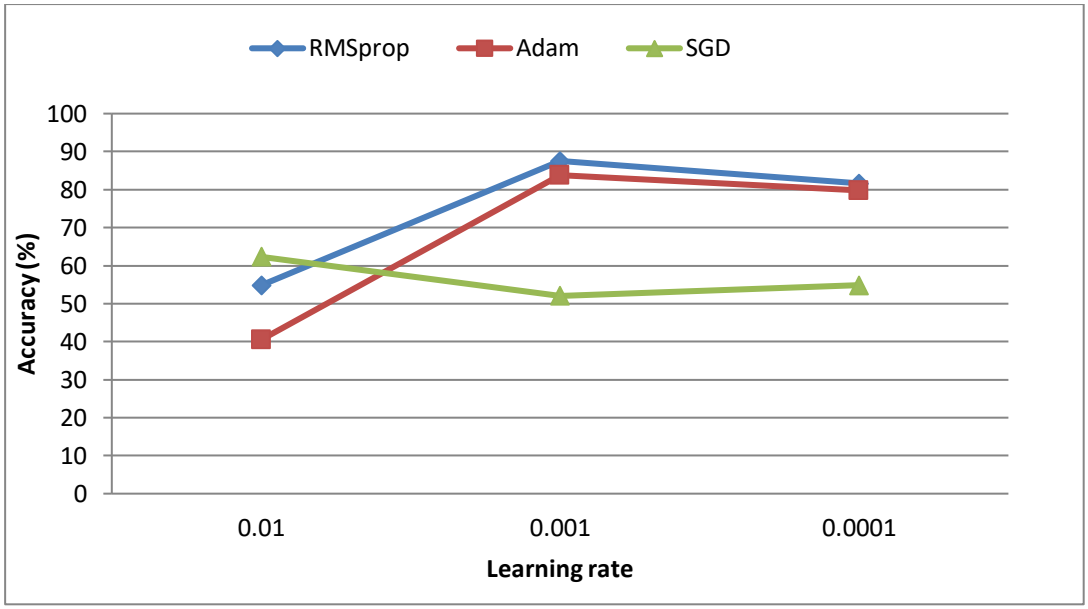


Figure 5 The accuracy of the models in relation to the learning rate and the optimizer for the architecture with two convolutional layers [64 and 32 filters each]

In the first architecture the model with optimizer RMSprop and learning rate 0.001 gave the highest accuracy of 87.54%.

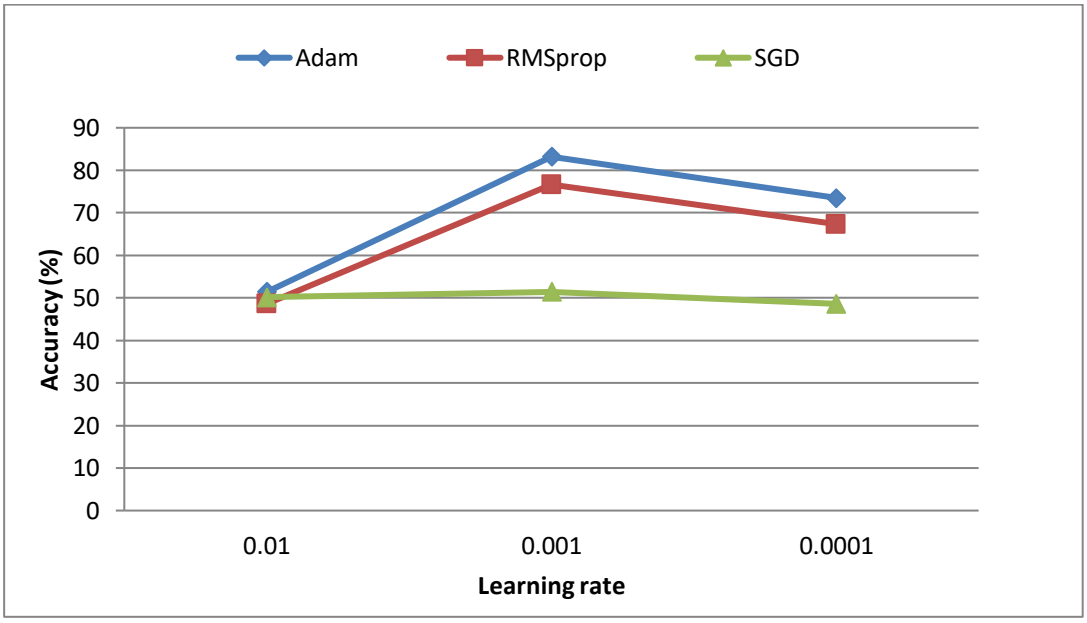


Figure 6 The accuracy of the models in relation to the learning rate and the optimizer for the architecture with three convolutional layers [64, 32 and 16 filters each]

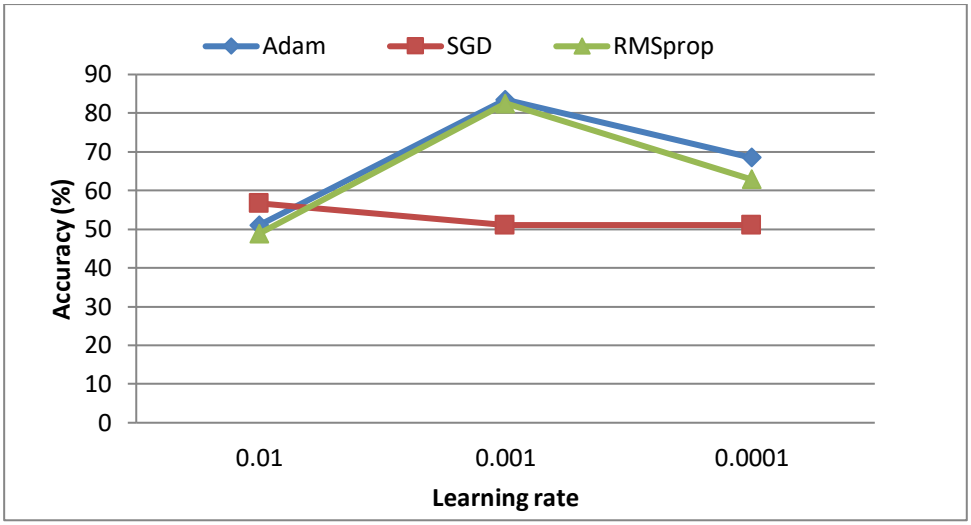


Figure 7 The accuracy of the models in relation to the learning rate and the optimizer for the architecture with three convolutional layers [64, 32 and 32 filters each]

In the second architecture, we also tested different values of filters (32 and 16) on the third convolutional layer. The highest accuracy of 83.49% was given by the model with Adam optimization algorithm, 0.001 learning rate and 32 filters at the third convolutional layer.

Unlike the first two architectures that had optimization parameters, the third architecture was fixed. This is because it is one of the solutions to the competition and exists in the GitHub repository [18]. Therefore, it was included for comparison of the results with the models we created and not for optimization. The model accuracy was 81.93% with Adam's optimization algorithm and 0.001 learning rate.

We then changed the input data, creating an additional table with the sum of zone_1 and zone_2. This time we only trained models with 0.001 learning rate and Adam and RMSprop optimization algorithms, due to the low accuracy of the other models. In the second architecture, the third convolution layers filters are 32 for the Adam optimization algorithm and 16 for the RMSprop. The overall results for the three architectures are shown in the diagram below.

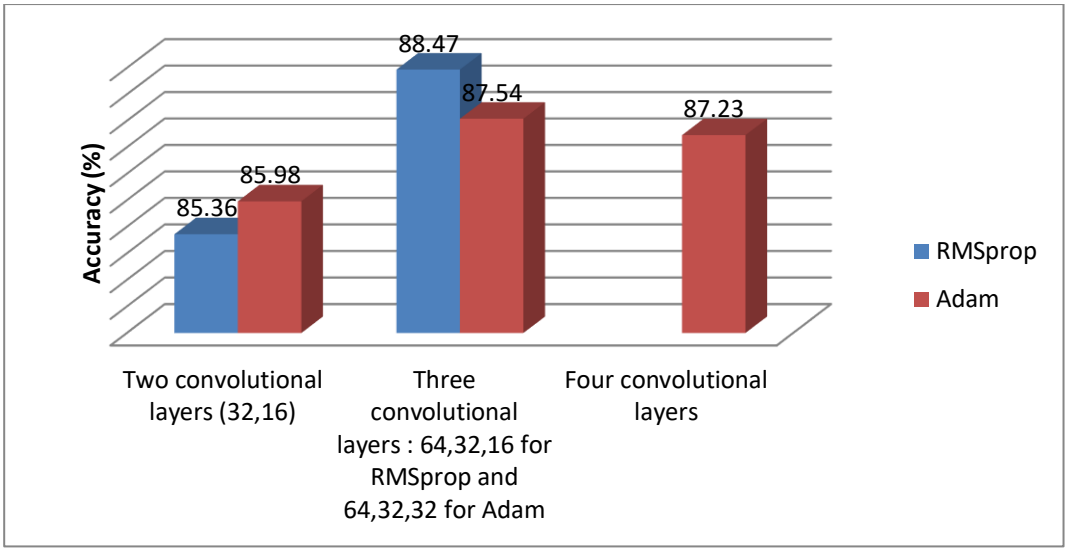


Figure 8 The accuracy of the models in relation to the learning rate and the optimizer for the architecture with three convolutional layers [64, 32 and 32 filters each]

When developing a neural network, we are interested in achieving as higher accuracy as possible and at the same time minimizing memory requirements. The following diagram shows the accuracy and the memory requirements for the basic models we tested. The optimal solutions are two in number. The one model is with three convolution layers, RMSprop optimizer, learning rate 0.001, 32 filters in the third convolution level, accuracy of 88.47%, cost function 0.29 and total memory requirement of 15MB. The other model is the one with four convolutional layers, Adam's optimization algorithm, 0.001 learning rate, 87.23% accuracy, 0.31 cost function and total memory requirement of 6.9MB.

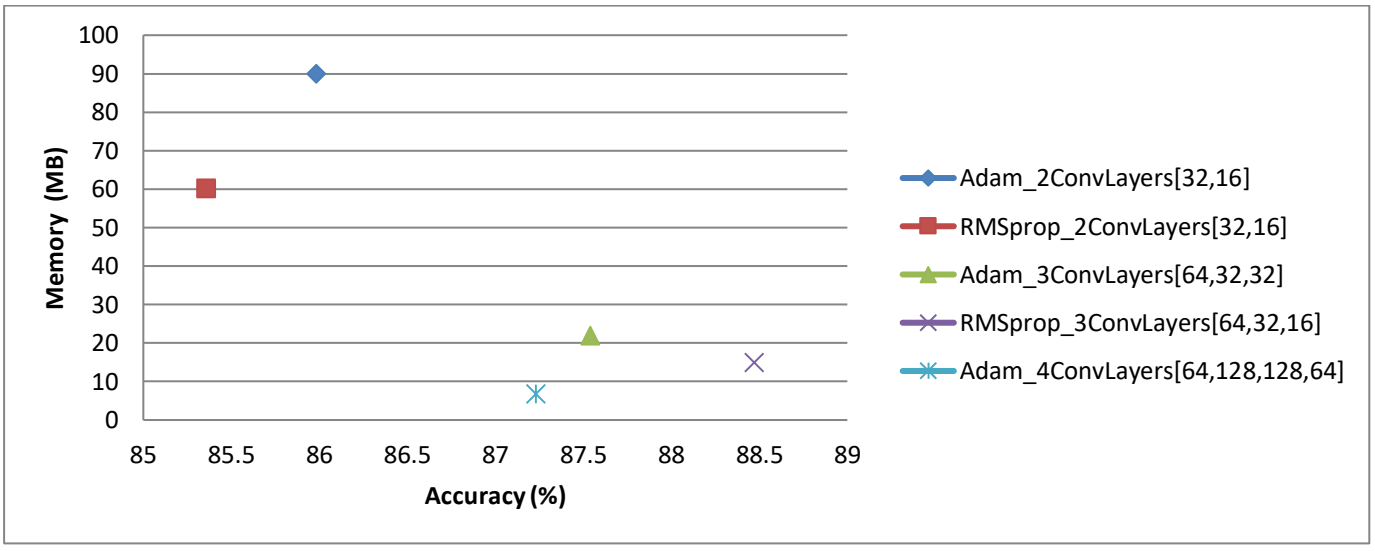


Figure 9 Accuracy and memory requirements of the models

4. Ships classification

The data was captured by the US Company Planet in the San Francisco Bay Area and San Pedro Bay, California. There are 4000, 80 * 80 RGB photos, which are classified as "ships" and "non-ships". The resolution of the images is such that each pixel corresponds to three meters.

The "ships" class includes 1000 images. The ships included have different orientations and sizes, as well as differences in their atmospheric collection conditions.

The "non-ships" class includes 3000 images, which are categorized into three subgroups. One group includes various landscapes such as sea, land, cities, etc. The second group consists of parts of ships that are not sufficient to fit into the category of "ships". The last group includes images that have previously failed from other machine learning models, usually caused by bright pixels or strong linear features.

In the particular data, the hyper-parameters for optimization were the number of hidden layers, the number of neurons for each filter, the activation function, the learning rate, the optimization algorithm as well as the percentage of dropout. As in the previous data, 80% of the data (3200 images) were used for training and the remaining 20% for testing (800 images).

4.1 Exploration fully connected neural networks

We create the following neural network architecture with five fully connected neural layers, which was trained for various values of its hyper-parameters.

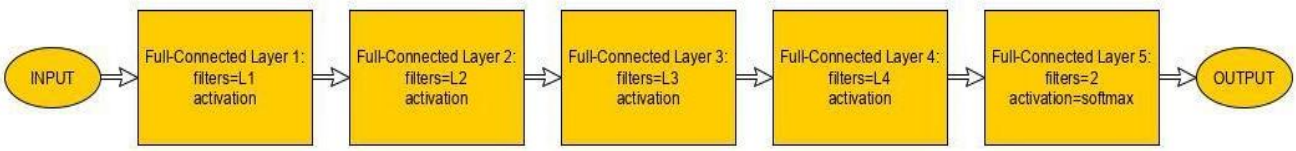


Figure 10 Architecture of fully connected network

The following methodology was used to optimize the hyper-parameters. First of all, we trained the network for various value combinations on the filters of each L1, L2, L3 and L4 layers. The chosen optimizer was Adam with learning rate 0.001 and activation function sigmoid.

Number of filters (L1_L2_L3_L4)	Loss function	Accuracy (%)
200_100_60_30	0.34	85.75
500_200_100_50	0.28	89.25
150_100_50_20	0.36	86.5
170_110_70_30	0.32	81.25
120_90_60_25	0.33	89
100_80_40_20	0.37	85.38
600_300_150_80	**0.27**	**90.88**
400_250_100_40	0.26	88.38
700_350_200_90	0.26	86.62
800_400_150_70	0.31	84.38

Table 2 Training results with different values of filters

For the best combination of filter (L1 = 600, L2 = 300, L3 = 150, L4 = 80) we change the activation functions to hard_sigmoid and tanh.

Activation function	Number of filters	Loss function	Accuracy (%)
sigmoid	**600_300_150_80**	**0.27**	**90.88**
hard_sigmoid	600_300_150_80	0.32	89.38
tanh	600_300_150_80	0.55	76.5

Table 3 Training results with different activation functions

Finally, for the best combination of filters (L1 = 600, L2 = 300, L3 = 150, L4 = 80) and best activation function sigmoid, we change the optimizer from Adam to SGD and to RMSprop.

Optimizer	Activation function	Filters	Loss function	Accuracy (%)
Adam	**sigmoid**	**600_300_150_80**	**0.27**	**90.88**
RMSprop	sigmoid	600_300_150_80	0.28	86.25
SGD	sigmoid	600_300_150_80	0.54	76.38

Table 4 Training results with different optimizers

Observing the results of the training for the various combinations we achieve to create a network with accuracy 90.88% and cost function 0.27. The characteristics of the model are optimization algorithm Adam, activation function sigmoid and number of filters L1 = 600, L2 = 300, L3 = 150, L4 = 80 at each hidden layer.

4.2 Exploration of convolutional neural networks

We first create the following architecture with four convolutional layers. The chosen optimization algorithm was Adam with learning rate 0.001.

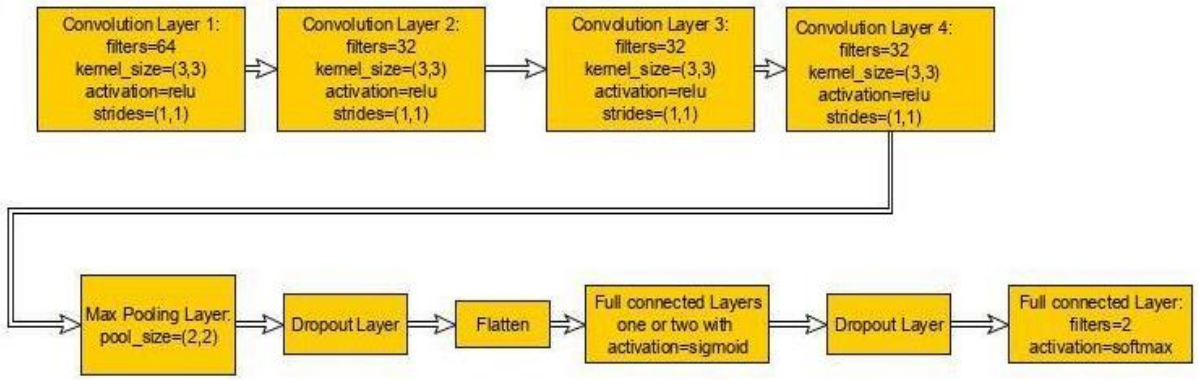


Figure 11 CNN architecture with four convolutional layers

The hyper-parameters that were examined were the number of layers and the number of filters of the fully connected layers, as well as the percentage of the dropout layer. The values of the dropout tested were 0, 0.1 and 0.15. The model with a hidden layer of fully connected neurons was trained for 100 and 200 filters, whereas the model with two hidden layers was trained with combinations of 200_80 and 300_100 neurons. The resulting models ranged from 73.25% to 76.75%. The accuracy rate was not satisfactory, so we constructed the following architecture with two convolutional layers.

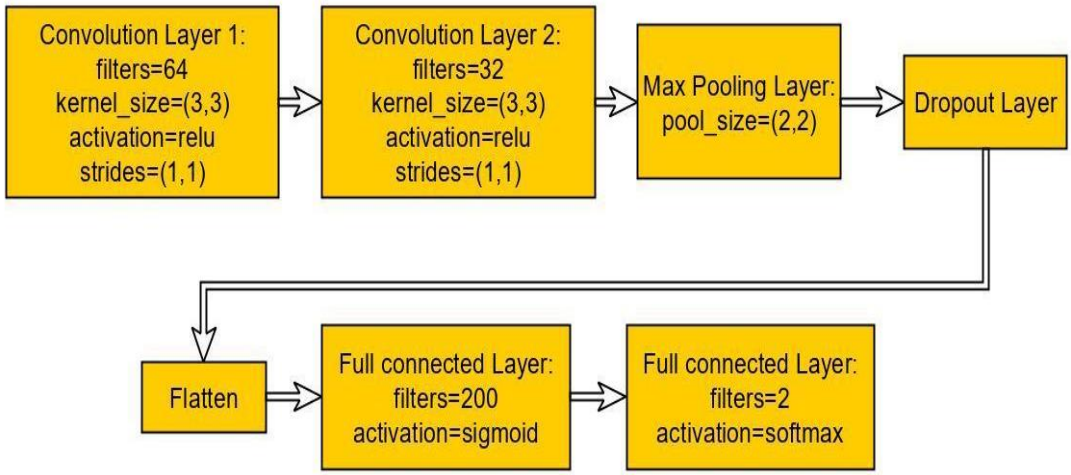


Figure 12 CNN architecture with two convolutional layers

The hyper-parameters examined were the optimization algorithm and the learning rate. Dropout was selected at 10%. We first trained the network for the three optimization algorithms Adam, SGD, RMSprop and three values for learning rate 0.01, 0.001, 0.0001. The results of 15 epochs training are shown in the diagrams below.

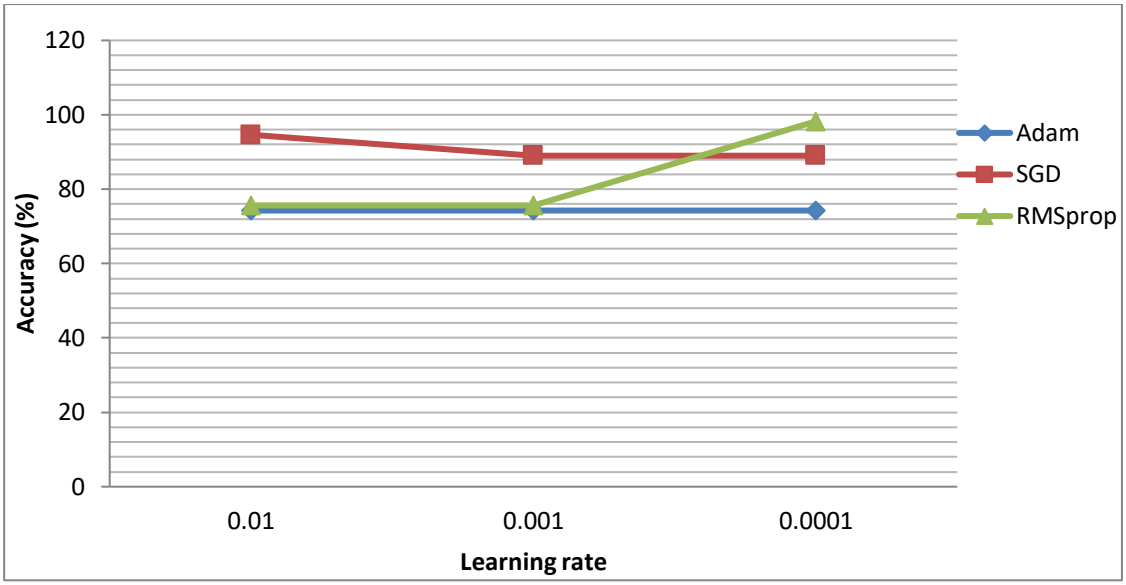


Figure 13 Accuracy for different optimizers and learning rate

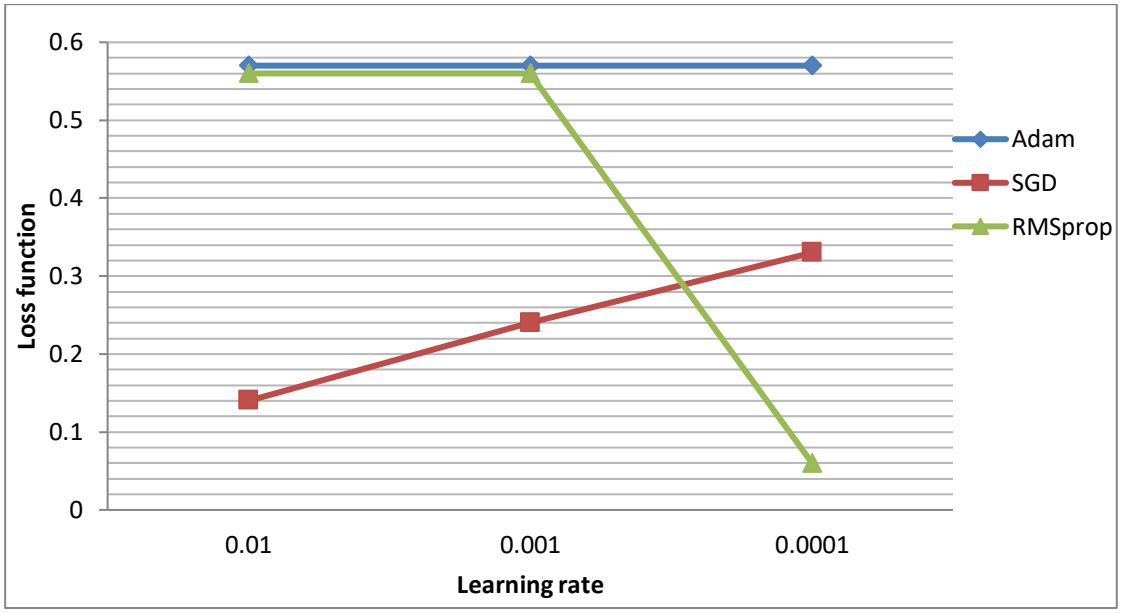


Figure 14 Loss function for different optimizers and learning rate

We then trained the best models of each optimizer for 40 epochs, collecting the following results.

Optimizer	Convolutional layer filters	Learning rate	Loss function	Accuracy (%)	Memory (MB)
Adam	64_32	0.01	0.56	75.25	98.6
SGD	64_32	0.01	0.07	97.5	65.7
RMSprop	64_32	0.0001	0.56	75.25	65.7

Table 5 Best models of each optimizer results for 40 epochs training

With the above results we conclude that the most suitable for this data optimizer is the SGD with a learning rate 0.01. This neural network achieves high prediction accuracy and at the same time has low memory requirement compared to other models. Of course, although prediction accuracy may consider satisfactory, memory requirements are not. Ways to further reduce memory while maintaining high prediction accuracy will be discussed in the next section.

5. Memory optimization and time profiling

Many of the recent research into deep convolutional neural networks (CNNs) have focused on increasing accuracy in virtual datasets. For a given level of accuracy, there are usually multiple architectures that succeed. Given equivalent accuracy, a complex architecture network (CNN) with fewer parameters has several advantages.

This thesis emphasized the reduction of parameters and the model memory in general so that it can be implemented in an FPGA platform. FPGA platforms often have less than 10MB on-chip memory and have no additional memory or storage. For inferences and predictions, a small model could be stored directly in the FPGA instead of devoting a large proportion of the forecast time to communication to update the parameters each time.

5.1 Redundancy of training parameters

In order to reduce the model memory we built the following network architecture by adding an extra convolutional layer.

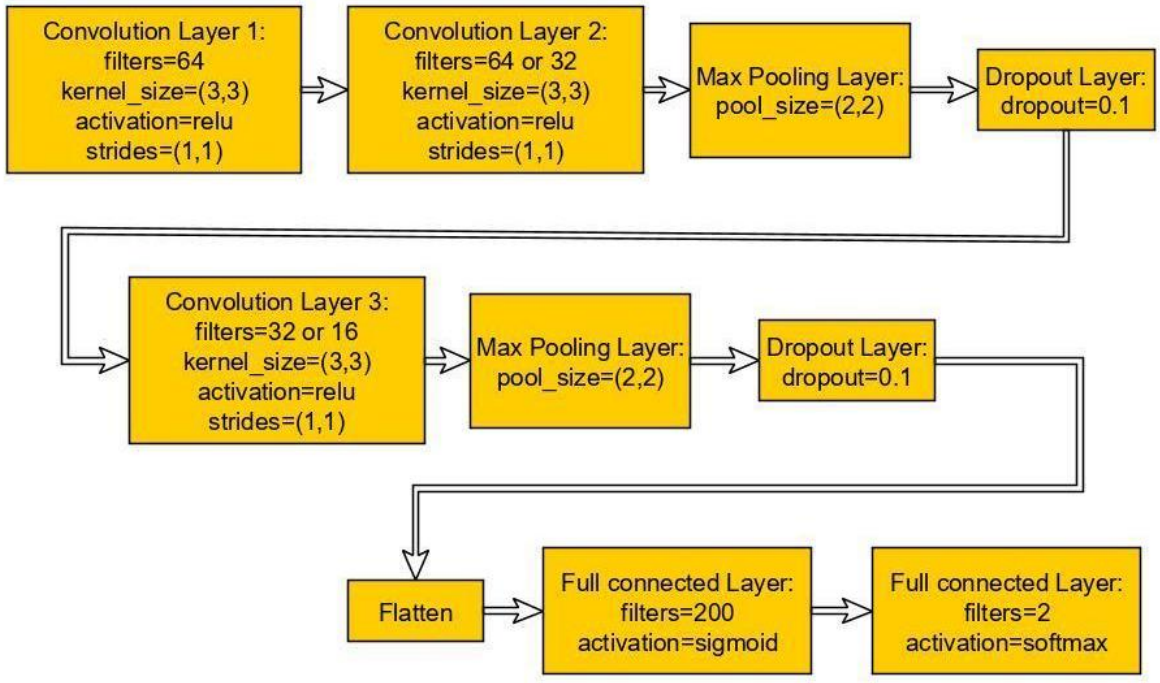


Figure 15 Network architecture with three convolutional layers and one full connected hidden layer

From the optimized model of the previous two-level convolution architecture, we kept the SGD optimization algorithm as well as the learning rate at 0.01. This time the hyper-parameters for optimization were the number of filters at each convolutional layer. Three combinations were tested, the results of which were satisfactory, as shown below, so no more were tested.

Optimizer	Filters on each convolutional layer	Learning rate	Loss function	Accuracy (%)	Memory (MB)
SGD	64_32_32	0.01	0.06	98.5	16.7
SGD	64_32_16	0.01	0.06	98.25	10.5
SGD	64_64_32	0.01	0.06	98.38	16.9

Table 6 Training results of model with three convolutional layers

It can be seen from the above results that by keeping the prediction accuracy at high levels we managed to reduce the model's memory from 82.1 MB to 10.5 MB, which is 87% decrease in memory requirements. Additionally, for the network with 64, 32 and 16 filters at each convolutional layer, we applied data augmentation with rotation mechanism. As a result of that both training and test data were doubled. After retraining, the model reached an accuracy of 98.63% and a cost function of 0.04.

Layer (type)	Output Shape	Parameters
Conv2D_1	(,64,80,80)	1792
Conv2D_2	(,32,80,80)	18464
Max_Pooling2D	(,32,40,40)	0
Dropout	(,32,40,40)	0
Flatten	(,51200)	0
Dense_1	(,200)	10240200
Dense_2	(,2)	402
	Total:	10260858

Table 7 Training parameters of model with two convolutional layers [64, 32 filters each]

Layer (type)	Output Shap	Parameters
Conv2D_1	(,64,80,80)	1792
Conv2D_2	(,32,80,80)	18464
Max_Pooling2D	(,32,40,40)	0
Dropout	(,32,40,40)	0
Conv2D_3	(,16,40,40)	4624
Max_Pooling2D	(,16,20,20)	0
Dropout	(,16,20,20)	0
Flatten	(,6400)	0
Dense_1	(,200)	1280200
Dense_2	(,2)	402
	Total:	1305482

Table 8 Training parameters of model with three convolutional layers [64, 32, 16 filters each]

Studying the number of parameters at each layer of the network for the model of 10.5MB and 82.1MB we obtain the above results. In general, the total amount of parameters at a convolutional layer is:

(Number of input channels) * (Number of filters + 1) * (Kernel size width * Kernel size height)

To maintain a small total number of parameters in a convolutional neural network with kernel size [3,3], it is important to reduce either the number of input channels in 3x3 filters or the number of filters. We can see in the above results that a larger volume of parameters occupies the first fully connected plane. The parameters of each fully connected layer are:

(Number of filters + 1) * (Input dimensions)

In order to maintain a small number of parameters at these layers, it is very important to reduce the input dimensions. A reduction in input dimensions can be very easily accomplished by adding an extra convolutional layer with a small number of filters and at the same time followed by a max pooling layer. As we have seen, the addition of convolutional and max pooling layers, although they increase memory space with 4624 parameters, managed to reduce about 10 times the number of parameters of the first fully connected layer and thus the total number of parameters.

5.2 Converting parameters to half precision type

We have mentioned before that the small memory requirement for implementing the model in an FPGA platform is very important. Another way to reduce memory is by using half precision float16 instead of the common simple precision float32, along with proper hardware and software support.

Compared to simple precision, half precision occupies 16 bits of memory instead of 32 bits, which means less storage space, less memory bandwidth, less power consumption, lower communication delay and higher arithmetic speed [21]. Of course, float16 functions must be supported by both hardware and software level to achieve good performance. Most general-purpose processors, for example, are not optimized for operations with float16 but for float32 and float64. This results in half-precision operations requiring more execution time than simple precision, as we will see in the next subsection.

The conversion of the model achieved with the following method.

At first, we created a float-16 model with the same architecture and saved it in .sav format by calling the createModelF16 function. The function code is followed below.

```
#call with: createModelF16(64,32,16,0.1,200,0.01,'categorical_crossentropy')
def createModelF16(L1conv,L2conv,L3conv,dropout,dense_sz,lr,loss):
        K.set_floatx('float16')
        kernel_sz=(3,3)
        model = Sequential()

        model.add(Conv2D(filters=L1conv, kernel_size=kernel_sz,strides=(1,1),
             padding='same', activation='relu',data_format="channels_first",
             input_shape=(3,80,80,)))
        model.add(Conv2D(filters=L2conv, kernel_size=kernel_sz,
                  strides=(1,1), padding='same', activation='relu'))

        model.add(MaxPooling2D(pool_size=(2,2)))
```

```
        model.add(Dropout(dropout))

        model.add(Conv2D(filters=L3conv, kernel_size=kernel_sz,
                  strides=(1,1), padding='same', activation='relu'))
        model.add(MaxPooling2D(pool_size=(2,2)))
        model.add(Dropout(dropout))

        model.add(Flatten(data_format='channels_last'))

        model.add(Dense(dense_sz, activation='sigmoid'))

        model.add(Dense(2, activation='softmax'))

        optimizer = SGD(lr=lr)
        model.compile(optimizer=optimizer,loss=loss,metrics=['accuracy'])

        # save the model to disk
        filename = 'model_f16.sav'
        joblib.dump(model, filename)
        #if we want to load the model from disk
        #model = joblib.load(filename)
```

Then we changed the parameters of the new model to the parameters of our already trained model. The only difference is that we first converted all the parameters of the model from float-32 to float-16. To implement this conversion we created the functions change_layer_type (layer_name, model), change_all_layers (model) and fill_empty_model (filename_f32, filename_f16).

The **change_layer_type (layer_name, model)** function receives the name of a layer and its corresponding model. It changes the type of the parameters of that layer to float16 and returns it in a list format.

The **change_all_layers (model)** function receives a model and converts the parameters of each layer into float16. This function returns a list consisting of lists, one for each layer of the model.

Finally, the **fill_empty_model function (filename_f32, filename_f16)** accepts the names of the models saved in .sav format. It changes the type of float32 model parameters to float16, and then fills the model with float16 pre-trained parameters. This is done using the above functions. It saves the changed model and returns it at the same time to evaluate it for accuracy and loss function.

The following is the code of the functions.

```
# Convert layer_name's parameters of model to float16
def change_layer_type(layer_name,model):
        filters = model.get_layer(layer_name).get_weights()
        new_filters = []
        new_filters.append(filters[0].astype(np.float16))
        new_filters.append(filters[1].astype(np.float16))
        return new_filters

# Implements the above function to all layers
def change_all_layers(model):
        layers = []
        for layer in model.layers:
            # check for convolutional or dense layer
            if ('conv' in layer.name) or ('dense' in layer.name):
```

```
                new_layer_f16 = change_layer_type(layer.name,model)
                layers.append(new_layer_f16)
        return layers

# fills model with parameters float16 with converted float32 parameters of
# pre-trained model ( models saved as .sav)
def fill_empty_model(filename_f32,filename_f16):
        # Set float32 as default float type
        K.set_floatx('float32')
        #load model with float32 parameters
        model_f32 = joblib.load(filename_f32)
        # convert to float16 all filter's parameters
        new_filters = change_all_layers(model_f32)

        # Set float16 as default float type
        K.set_floatx('float16')
        #load model with float16 parameters
        model_f16 = joblib.load(filename_f16)

        i=0
        for layer in model_f16.layers:
        # check for convolutional or dense layers
            if ('conv' in layer.name) or ('dense' in layer.name):
                model_f16.get_layer(layer.name).set_weights(new_filters[i])
                i=i+1

        # save the filled model to disk
        filename = 'model_f16.sav'
        joblib.dump(model_f16, filename)

        return model_f16
```

Finally, we evaluated the new model in the test data. The results showed that the conversion of the parameters to half precision did not affect the accuracy and loss function at all and at the same time reduced the model memory requirements from 10.5MB to 2.6MB.

5.3 Time profiling

Before attempting to implement the model with the FPGA platform, it is very important to analyze the time it takes each network function to process the RGB image at prediction. To measure the time required for each operation we used the TensorFlow timeline function. This function records the operations performed and the time they require during training or in our case when predicting images. We then save it to a .json file and can load it from the "chrome: // tracing" page in Google Chrome to analyze the results. We also measured the total prediction time using the python time.time () function. The above were applied to the models with float32 and float16 parameters and we obtained the following results.

	Float32	Float16
Conv1	6.72	6.736
Conv2	8.055	15.812
MaxPooling1	0.57	1.787
Conv3	0.771	2.086
MaxPooling2	0.114	0.3
Dense1	1.971	25.496
Dense2	0.521	0.036
Total	19.857	52.824

Table 9 Time per layer for float32 and float16 model

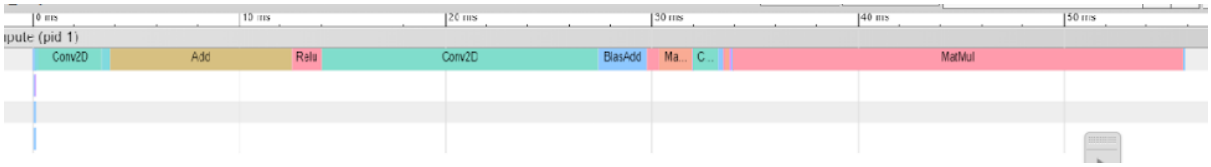


Figure 16 Timeline from chrome://tracing for float16 model

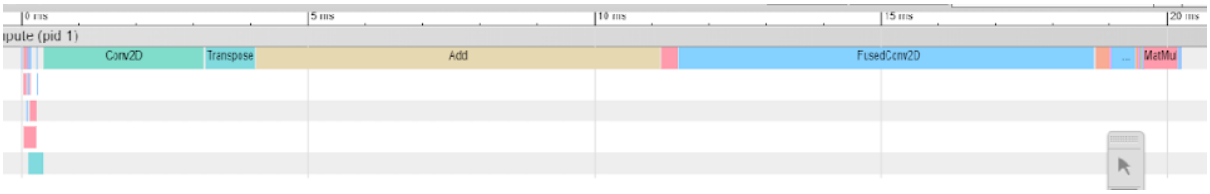


Figure 17 Timeline from chrome://tracing for float32 model

The total prediction time of an image with the half-precision model is 52.824ms. The most time-consuming layer is the first fully connected that takes almost the 50% of the total inference time. Second is the second convolutional layer that takes another 30% of the inference time.On the other hand, the total inference time of an image with the simple precision model is 20.461 ms. In contrast to the previous results, the most time-consuming operations are the first and the second convolutional layers, that occupies almost the 75% of total inference time. In the above results we can see that in terms of the real time required by these functions, the half-precision model requires almost twice as long as with simple precision. We can see this better in the diagram below, which depicts the time required in ms per layer for both models.

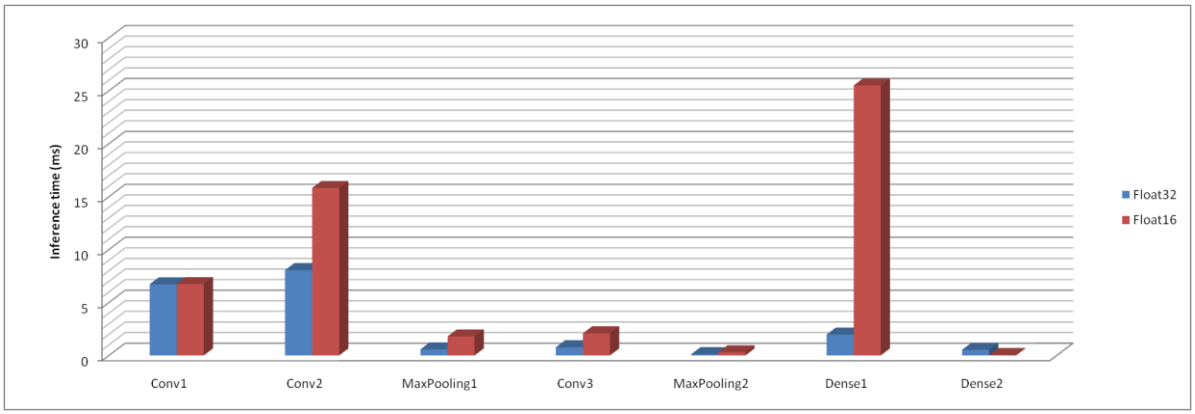


Figure 18 Inference time per layer for both FP32 and FP16 models

From this diagram we can extract the following observations.

- Convolution at the second convolutional layer takes longer than the first and third convolutional layers. This is because the number of multiplications and additions required is much greater than the in other two levels. The number of operations required is determined by both input dimensions and the number of parameters for each layer. Thus, it takes more time for the second convolutional layer with 18464 parameters and input dimensions (,64,80,80), followed by the first convolutional layer with 1792 parameters and input dimensions (,3, 80,80) and finally the third layer with 4624 parameters and input dimensions (, 32,40,32).
- The same is happening for both fully connected layers. The hidden fully connected layer with 200 neurons has 1280200 parameters while the output layer has only 402 parameters. Consequently, the hidden layer takes more time to complete since the number of operations is much higher.
- Also the time of multiplications at the first fully connected layer is much longer in the CNN-f16 half precision model than in the CNN-f32 single precision model. The reason for this is that the CPU of a computer, such as the one running the prediction program, has usually been optimized for single-precision (float32) and double-precision (float64) operations, since they are the most common used. At the same time, it is not optimized for half-precision operations, which results in much more execution time.

The number of multiplications and the required memory for the results (for float16 precision) have been calculated and are shown to the below table.

Layer (filters)	Parameters	Mults	Output Shape	Elements of Output	Total requirement memory for output (MB)
Conv2D_1 (64)	1792	11059200	64,80,80	1228800	2.4576
Conv2D_2 (32)	18464	117964800	32,80,80	39321600	78.6432
MaxPooling1	--	--	32,40,40	9830400	19.6608
Conv2D_3 (16)	4624	7372800	16,40,40	157286400	314.5728
MaxPooling2	--	--	16,20,20	39321600	78.6432
Dense1	1280200	1280000	200	1228800	2.4576
Dense2	402	400	2	12288	0.024576

Table 10 Multiplications and Memory per layer at inference time

We then measured the total prediction time of one and one hundred images using the python time.time () function. To make our measurements more objective we repeated the time measurements four times and kept the averages. The execution times of the two models are shown in the table below.

MODEL	time.time() for 1pic (ms)	time.time() for 100pic (ms)
CNN-f16	119.177275	3687.5293
CNN-f32	100.8913	1443.239625

Table 11 Average execution time for one and one hundred image predictions

The results show that predicting 100 photos does not require a hundredfold completion time but much less because of the optimizations that performed during execution. Also, the prediction time of an image measured with the time.time () function is longer than the time we saw it required in the timeline. This is because this function measures the total time that the prediction required to complete on our computer. On the other hand the timeline measures only the time that an operation requires to complete on our CPU.

6. High level tools for network implementation to FPGA

Traditional FPGA design tools

Hardware description languages (HDL) such as VHDL and Verilog are the first languages to design algorithms that run on FPGA chips. HDL languages reflect some of the properties of other text languages, but they differ significantly because they are based on a data flow model where inputs and outputs are connected to a series of modules via signals.

After creating and verifying a FPGA design using HDL, it is powered by a compilation tool that takes the text-based logic and through many complex steps composes the HDL code into a configuration or bitstream file containing information on how to connect the various elements between them.

High-level synthesis tools

Another way to design an application in FPGA is through a high level language such as C / C ++. Tools such as Vivado HLx, for example, have been developed to accelerate IP generation, enabling C / C ++ and System C specifications to be implemented directly on Xilinx programmable devices without the need for HDL code generation.

Recent machine learning tools for directly applying models to the FPGA

Many studies have shown that the use of FPGA platforms in the field of machine learning has led to a huge acceleration of applications [24, 25, and 26]. The most common way to create a neural network is by using a high-level tool such as TensorFlow or Keras with python. Therefore, to synthesize the model and apply it to the processor requires manual translation of the network components in either C / C ++ or HDL, as mentioned above. This process is very time consuming and requires experience, limiting the application of FPGAs in this field. So much research interest has turned to creating tools that transform pre-trained models, with TensorFlow, Keras, Caffe or some other framework, into one of the design languages mentioned above. Recently, several such commercial and other open source tools have been developed. In this thesis we have tested three such tools. Two open source tools were first tested, LeFlow [22] and hls4ml [23] and then the commercial tool of Xilinx Machine Learning Suite [22, 23, 27], which are presented below.

6.1 LeFlow tool

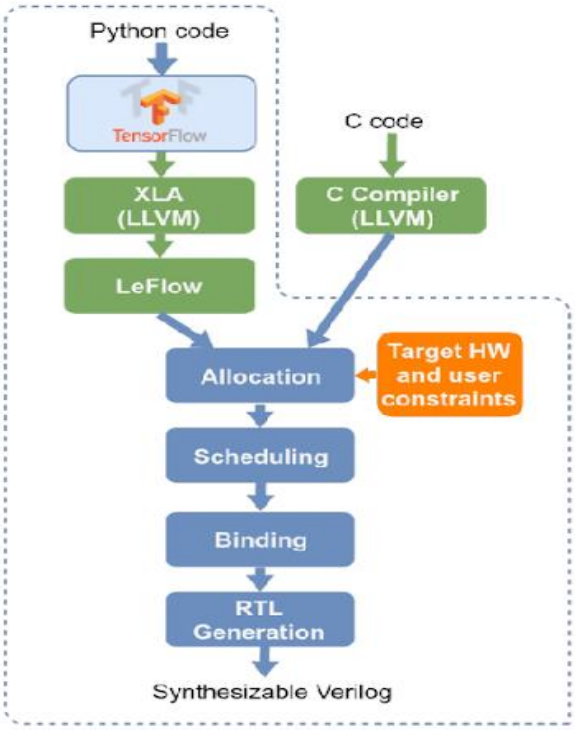


Figure 19 Execution flow of LeFlow using a file in python compared to the standard HLS flow using a C code file [55]

LeFlow is an open source tool that converts numerical computational models written in TensorFlow, such as a convolutional neural network, into a form that can be synthesized for application to an FPGA platform. Has been developed by Daniel H. Noronha, Bahar Salehpour, and Steven J.E. Wilton at the Department of Electrical and Computer Engineering, University of British Columbia, and published last year (2018) [38].

The generated file is in verilog code. Verilog is automatically produced in collaboration with Google's high-level LegUp XLA compiler and LeFlow tool. The above picture shows the execution flow of LeFlow using a file in python compared to the standard HLS flow using a C code file.

LeFlow was built to be LegUp 4.0 compatible, so we installed the corresponding virtual machine. Then we installed TensorFlow and LeFlow on the virtual machine. LeFlow makes some minor changes to TensorFlow to ensure that LegUp-supported kernels are only used by Tensorflow's XLA.

The above installations were successfully completed. The model we want to convert to verilog, as mentioned above, has been developed with Keras in TensorFlow so it is also necessary to install the Keras on the virtual machine. Installation requires the installation of some libraries, including the 'six' library, which requires a newer version. This library, however, is protected on the virtual machine and cannot be uninstalled in order to install the latest version required for Keras. This dependency eventually prevented Keras from being installed so the tool could not be implemented on the optimized 2.6 MB model and 98.63% accuracy.

6.2 Tool hls4ml

Hls4ml is an open source tool for creating firmware implementations of machine learning algorithms using high level synthesis language (HLS). These files can then be synthesized with Xilinx's Vivado HLS program. A known issue with Vivado HLS using the above tool is that loop unrolling creates memory problems when compiling the generated code. As mentioned in section 5.2.2 we have overcome this by changing some commands that address how to implement the program at the hardware level.

The HLS tool provides the so-called pragma commands that can be used to optimize the design: reducing delay, improving the throughput, and reducing the required space and resources of the resulting RTL code. These pragma commands can be added directly to the source code.

6.2.1 Methodology for synthesizability

It should be noted that the generated code was synthesized in the program "Vivado HLS 2018.1" and the ZYNQ-7 ZC702 Evaluation Board was selected to apply our model. The features of this board are shown below.

Characteristics ZYNQ-7 ZC702 Evaluation Board	
Logic cells	85K
LUTs	53200
Flip-Flops	106400
Total Block RAM	4.9 MB
DSP slices	220

Table 12 Recourses of ZYNQ-7 ZC702 evaluation board

We originally created an .yml configuration file containing various attributes to extract the desired code. Contains:

- The file name with the network architecture in .json format [KerasJson].
- The name of the file containing the weights of the trained model in .h5 format [KerasH5].
- The device where we want to implement the final code, in our case, is the ZYNQ-7 ZC702 Evaluation Board with code xc7z020clg484-1 [XilinxPart].
- The type of input and output here was chosen at first parallel mode and after a serial implementation to save resources [IOType].
- The clock period, that was selected at 10ns [ClockPeriod].
- Also, contains the output dir, the project name and a reuse factor in case of parallel I/O type.

Then using the hls4ml tool we extracted the hls code from some simpler models with less layers. We noticed that loop unrolling created a problem in memory when compiling the generated code. The loops are unwrapped during the passage so that they can be run in parallel. Therefore, we removed the pipeline from the file 'nnet_conv2d.h' that contains loops with a huge number of iterations in order to execute convolution. The problem of memory continued to exist in the implementation of fully connected layers. Therefore, we also removed the pragma hls dataflow command from the file 'nnet_dense.h' which also requires looping. This way the memory problem was solved.

The next issue encountered during the compilation was that as vivado automatically renamed some of the functions it could not later recognize the pool_op function of the file 'nnet_pooling.h'. The implementation of this function was only a few lines of code, so we replaced every call with the necessary commands. With these changes the synthesis was successfully completed.

As we tried to compile the code for some more complex architecture, we noticed that only models with one max_pooling layer successfully completed the synthesis process. The other models had errors in the second call of the pooling2d function and in all subsequent calls. Therefore, this led us to create two network architectures with three convolutional layers, as before, but with one max pooling layer as illustrated below.

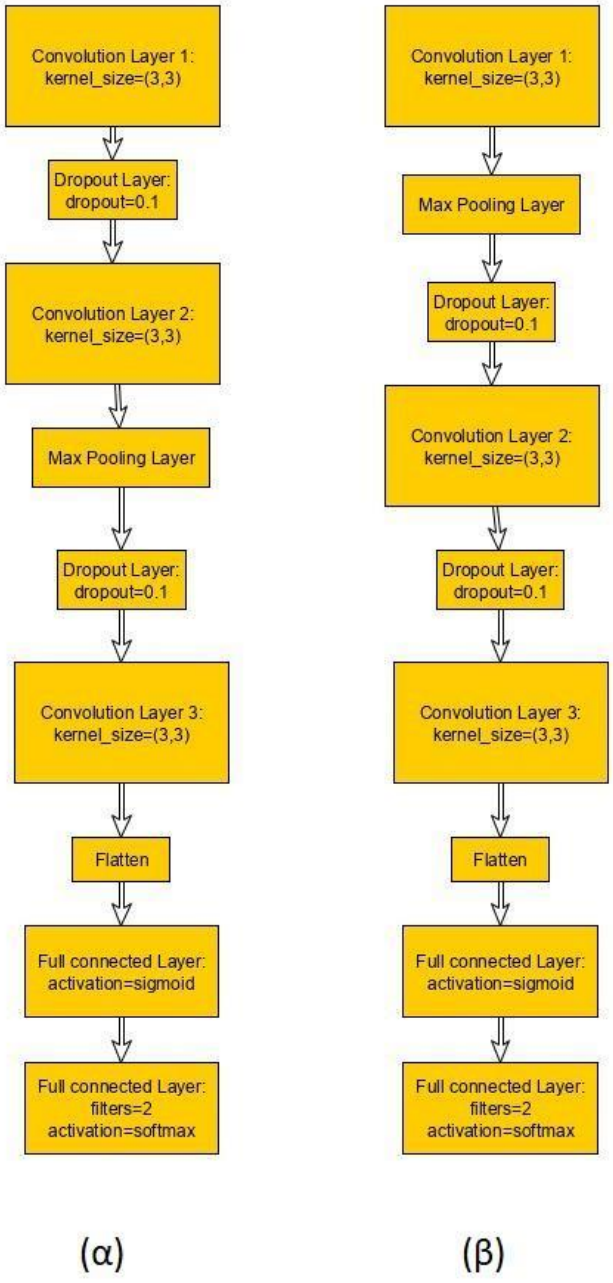


Figure 20 Network architectures with one max pooling layer and three convolutional layers (a) max pooling layer is after the second convolutional layer, (b) max pooling layer is after first convolutional layer

For the above architectures was selected the SGD optimization algorithm with 0.01 learning rate. Max pooling layers not only help to select key features, they also reduce the input dimensions of the next layer. By removing one of them, the final input dimensions to the fully connected layers will be increased. This implies an increase in training parameters and thus an increase in model memory. For this reason we increased the max pooling layer from [2, 2] to [3, 3]. In addition, for the same reason, at the first fully connected layer we decreased the number of neurons from 200 to 50. The hyper-parameters for optimization were the number of filters at convolutional layers. The models were trained for thirty-five epochs and the results follow in the table and the diagram to help us select the best model.

Model architecture	Filters of 1rst conv layer	Fileters of 2nd conv layer	Filters of 3rd conv layer	Accuracy (%)	Loss function	Memory
(α)	32	16	8	97.44	0.09	951.5 kB
(α)	32	8	8	95.56	0.11	900.9 kB
(α)	64	16	8	97.94	0.07	1.9 MB
(β)	64	16	8	97.25	0.08	1.9 MB
(β)	32	16	8	97.31	0.08	951.7 kB
(β)	32	8	8	96.25	0.1	900.9 kB

Table 13 Results after 30 epochs training

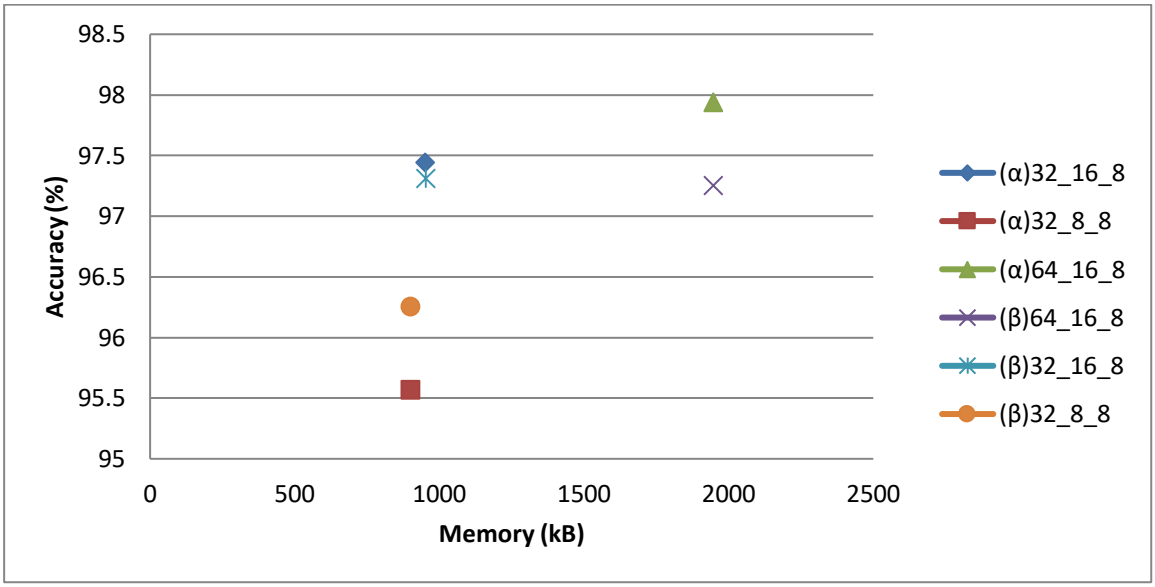


Figure 21 Model accuracy compared to required memory

For model selection we want a combination of high accuracy and low memory requirements. From the above diagram we can easily understand that the best solutions are the models (a)64_16_8, (a)32_16_8 and (b)32_8_8. So we choose the solution (a)32_16_8 that has a fairly small memory of 951.5kB and a rather high accuracy of 97.44%.

So we proceeded to implement the hls4ml tool for this model. We made the changes mentioned above in the exported files 'nnet_conv2d.h', 'nnet_pooling.h' and 'nnet_dense.h'. In addition, because this model is quite large, there was a problem with the pragma hls partition commands, which exceeded the division threshold when executing the synthesis. They were therefore removed from the files 'nnet_conv2d.h' and 'nnet_dense.h'. Finally, we synthesized the hls code and got the following results.

LUTs (%)	13
Slice Registers-FF (%)	2
RAM Blocks (%)	3661
DSP (%)	2
Clock (ns)	10
Frequency (MHz)	100
Latency (ms)	48.6
Throughput (Images/sec)	20.57613169
Input Size (RGB images)	80,80,3
Word Length	16 fixed point
CNN Accuracy (%)	97.44

Table 14 Time and recourse requirements of model (a)32_16_8

According to total resource requirements we understand that although the synthesis process has been completed, the HLS program cannot be implemented to a FPGA, as it requires BRAM_18K 3661% of the available blocks. Observing the resource requirements of each instance it appears that the most demanding requirements in BRAMs have been the implementation of the last two convolutional layers. This is because the program stores in memory the results of the multiplications of each convolutional layer.

A decrease in the multiplications can be made by increasing the dimensions of the max pooling layer. This led us to create a network similar to the previous with the exception that at max pooling layer we increased the dimensions from [3, 3] to [4, 4]. Making the necessary changes to the files 'nnet_dense.h', 'nnet_conv2d.h' and 'nnet_pooling.h', we proceeded with the synthesis process. This change in model resulted in the same problem as in models containing two max pooling layers.

Another way to reduce multiplications is to further reduce filters to convolutional layers. Moreover we chose the architecture (b) with the max pooling layer immediately after the first convolutional layer to reduce the input dimensions of following layers. So we created and trained a model with 16, 8, 8 filters respectively at each convolutional layer of (b) architecture. The accuracy for ten epochs training was 95.5%. The results of the synthesis are shown below.

LUTs (%)	12
Slice Registers-FF (%)	2
RAM Blocks (%)	1501
DSP (%)	2
Clock (ns)	10
Frequency (MHz)	100
Latency (ms)	24.3
Throughput (Images/sec)	41.15226337
Input Size (RGB images)	80,80,3
Word Length	16 fixed point
CNN Accuracy (%)	95.5

Table 15 Time and recourse requirements of model (b)16_4_4

These changes reduced the use of BRAMs to 1501%. The BRAMs requirements of the third convolutional layer reached 4101 instead of 8201 and of the second convolutional layer reached 35 instead of 1933 BRAMs in relation to the initial implementation. Despite the significant reduction in resources, the model still cannot be implemented in the FPGA.

The next model we tested was the same architecture (b) with 8 filters at each convolutional layer. The model was trained for twenty epochs and achieved 94.88% accuracy and 0.14 loss function. The utilization of BRAMs was reduced to 760% but the model still cannot be implemented in the FPGA.

Then, we reduced the filters from 8 to 4 of the second convolutional layer. The new model was trained for twenty epochs and achieved 94.44% accuracy and 0.14 loss function. The use of BRAMs was reduced to 748%. The huge reduction in the model allowed us to add all pragmas commands except loop unroll to the file 'nnet_dense.h'. At the same time in the file 'nnet_conv2d.h' we added the pragma hls dataflow command for conv2d and pragma hls stream in the mult table to store multipliers from convolution to FIFO instead of BRAMs. With these changes the resource requirements are such that they can be covered by the available FPGA resources. The detailed results of the synthesis are shown below.

LUTs (%)	43
Slice Registers-FF (%)	5
RAM Blocks (%)	22
DSP (%)	25
Clock (ns)	10
Frequency (MHz)	100
Latency (ms)	40
Throughput (Images/sec)	25
Input Size (RGB images)	80,80,3
Word Length	16 fixed point
CNN Accuracy (%)	94.44

Table 16 Time and recourse requirements of model (b)8_4_8

Also, as less than 50% of platform resources were used, we added some additional command pragmas to the code. Changes have been made to the files 'cnnet_conv2d.h' and 'cnnet_dense.h'. First set every loop of functions in the files to be executed pipelined. In addition, we added the pragma hls inline command to conv2d and dense_latency functions. This command removes a function as a separate unit in the hierarchy. The function is integrated into the calling function and no longer appears as a separate level of hierarchy in the RTL file. Inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations. The results of the synthesis are as follows.

LUTs (%)	48
Slice Registers-FF (%)	5
RAM Blocks (%)	22
DSP (%)	30
Clock (ns)	10
Frequency (MHz)	100
Latency (ms)	27.9
Throughput (Images/sec)	35.84229391
Input Size (RGB images)	80,80,3
Word Length	16 fixed point
CNN Accuracy (%)	94.44

Table 17 Time and recourse requirements of model (b)8_4_8 , with inline and pipeline

6.3 Machine Learning Suite

The Xilinx Machine Learning Suite (ML) tool provides users with the tools to develop machine learning applications for real-time inference. It provides support for many common machine learning frameworks such as Caffe, MxNet and Tensorflow. It consists of three main parts:

1. xDNN IP - Generic High Performance Convolutional Neural Network (CNNs) Processing Machine. The main layers that supports are:

- Convolution layer: The dimensions of the kernel can be any combination of numbers 1 to 8.
- ReLU activation function
- Pooling: Kernels [2,2] and [3,3] are supported for maximum pooling level. The average pooling is not supported in all versions.

2. xfDNN Middleware - Software library and tools to interface with ML frameworks and optimize them for real-time conclusions.

3. ML Framework and Open Source Support

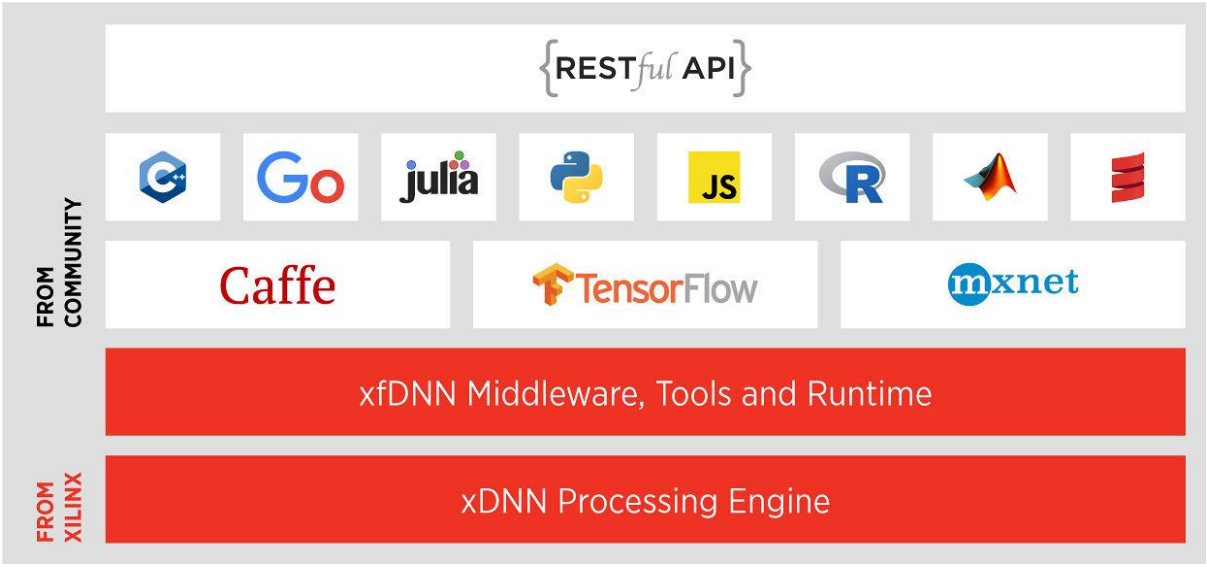


Figure 22 Main parts of Xilinx ML-Suite [56]

As we have seen above with this tool we can implement various models, developed with python on tensorflow, on the board of our choice without having to develop any vhdl or verilog code.

In order to proceed we had to make some changes to the previous model (b)8_4_8. First, we removed the dropout layers as they are not supported and secondly we change the way that model receives the data from channels_first to channels_last. With these two changes, the model was trained for 20 epochs with a final accuracy of 98.13% and memory requirements of 2.2 MB. Initially, the model had to be converted to a format compatible with the xilinx tool; therefore it was saved as a protobuf (.pb) file.

Applying the python model to the board first requires the completion of some stages supported by the tool. The tool is available in a docker container and all that we will mention below was completed in it.

We initially configured the application environment to be compatible with the alveo-u200 processor by executing the command:

```
source $ MLSUITE_ROOT / overlaybins / setup.sh alveo-u200
```

The model was then optimized with the DECENT_Q tool. This is required because FPGAs will benefit from Fixed Point Precision to achieve greater parallelism at lower power. After compression of the model the accuracy remained high and was 97.66%. This was accomplished by the command:

```
python run.py --quantize --model /opt/ml-suite/share/ML_Suit/my_model.pb --
output_dir work --input_nodes conv2d_1_input --output_nodes dense_2/Softmax -
-c_input_nodes conv2d_1_input --c_output_nodes dense_2/Softmax --input_shapes
?,80,80,3
```

Also, changes required to the utils.py file to support our own model. We changed the model input name to "conv2d_1_input" and both "input_fn" and "top5_accuracy" functions. The first function loads up to 300 images at a time (due to the large amount of data) and the corresponding outputs, and returns the desired one, which is specified by the iter variable. This way we were able to load all the test data, which was 1500 images, for testing. The second function calculates the model accuracy for as many images as the iter_cnt variable.

Finally, partition, compile and run inference are performed. In general, to run a tensorflow model in FPGA, a new graph must be compiled and created. The new graph is similar to the original, without the sub-graph to be executed in FPGA and replaced with a custom Python layer. The model is split into one part to run on the FPGA board, convolutional layers with pooling layers, and another part to run on the CPU, fully connected layers. All this is done by the command:

```
python run.py --validate --model /opt/ml-suite/share/ML_Suit/my_model.pb --
output_dir work --input_nodes conv2d_1_input --output_nodes dense_2/Softmax -
-c_input_nodes conv2d_1_input --c_output_nodes dense_2/Softmax --input_shapes
?,80,80,3
```

In the above step, all steps were performed successfully before connecting to the alveo-u200 board. In order to run on this platform, was needed the xrt 2018.2 version and the lab has the newer version of 2019.

6.2.2 Comparison with results of previous theses

Tool	Developent time	Additional knowledge	Tool category	Functionality
LeFlow	2 weeks	Basic knowledge on virtual machines	open source	No support of Keras models
Hls4ml	2 weeks	Basic knowledge of HLS (pragma commands) Basic knowledge using Vivado HLS of xilinx	open source	Support keras models at json and h5 files
Xilinx ML Suit	2 weeks	Basic knowledge of docker containers	commercial	Support keras models at pb file, not compatible with xrt 2019 version

Table 18 Compare of the tools

After the comparison of the tools, we examined our final results with two previous theses that had also dealt with applying CNNs models to special-purpose processors. In the first thesis, FPGA architectures were developed in VHDL designed to implement CNNs using only programmable logic memory. Simultaneously, the "Modified Cifar-10 Full", a deep CNN of few bits, was built and trained with the Caffe framework on images of the SAT-6 airborne dataset [35]. The second thesis aimed at developing a CNN execution system for the integrated Myriad2 multiprocessor [34]. He did this by developing every element of a convolutional network into assembly. For comparison with our model we got the results with input dimensions 80,80,3 and the model "CIFAR 10 Quick CNN" which is similar to the model of the first thesis.

Board	ZYNQ-7 ZC702		Myriad2 SoC
Model	(b)8_4_8 , with inline and pipeline	Modified Cifar 10 Full CNN	CIFAR 10 Quick CNN
LUTs (%)	48	75.81	--
Slice Registers-FF (%)	5	52.36	--
RAM Blocks (%)	22	57.12	--
DSP (%)	30	99.09	--
Clock (ns)	10	10	--
Frequency (MHz)	100	100	--
Latency (ms)	27.9	0.551	3.6
Throughput (Images/sec)	35.84229391	4650	277.8
Input Size (RGB images)	80,80,3	28,28,3	80,80,3
Word Length	16 fixed point	I/O: 4bits Conv: 8bits FC: 2bits	16-bit floating point
CNN Accuracy (%)	94.44	94.89	--

Table 19 Comparison with results of previous theses

Observing the above results we realize that in terms of execution time it does not come close to other implementation times. But what we did succeed was to memory optimize the model with Keras, apply it to the board using only programmable memory and most important at small development time (half a month). Despite the long execution time, we believe that in the near future such tools will be able to compete with the execution times of implemented neural networks, which have been developed with traditional ways (HDL/assembly).

7. Conclusion

Over the years, more and more applications are being developed for machine learning. To be successful, it must have low latency, low power consumption and at the same time high adaptability. Thus, the use of FPGAs in such applications is the best option.

In this thesis, due to the limited number of FPGA resources, we were called upon to construct convolutional neural networks with low memory requirements while maintaining high accuracy. The neural networks were designed with the help of keras in tensorflow. At first, data from two kaggle competitions were used and trained. The first competition is called the "Statoil iceberg classifier challenge" and aims to determine if an image contains iceberg or ship [3]. For this data, a convolutional neural network with an accuracy of 88.47% and memory requirements of 15MB was constructed.The data represented radar signals so we did not have the proper background to make the most of them and so we focused on ship identification data. The second competition is called "Ships in Satellite Imagery" and aims to identify ships from satellite images [4]. For these data we have been able to construct a convolutional neural network of 98.63% accuracy and only 2.6 MB of memory requirements. This was achieved by designing an architecture with a limited number of trainable parameters and changing the data type of weights from simple precision (float32) to half precision (float16).

Although the advantages, a big issue in implementing neural models with FPGAs is that it requires a lot of development time as well as a lot of developer experience. To address these issues, much research interest has focused on the creation of tools to easily convert neural models into a form that can be applied to FPGA boards. Therefore, in the second part of this thesis, three such tools were tested. The first two open source tools were LeFlow which generates verilog code [22] and hls4ml that generates hls code [23]. The third commercial tool was Xilinx's ML-Suite. More emphasis was placed on the hls4ml tool with which we were able to apply the model to the ZYNQ-7 ZC702 board. Due to the nature of the tool, in a very short development time (two weeks), we were able to implement the neural network using only programmable logic memory and with a final throughput of 35 Images / sec.

Finally, we studied older implementations of convolutional neural networks in the ZYNQ-7 ZC702 and Myriad-2 SoC boards, which were developed using vhdl and assembly. Although they achieved better throughput, their development time was much longer than ours. Despite the long execution time, we believe that in the near future such tools will be able to compete with the execution times of implemented neural networks, which have been developed with traditional ways (HDL/assembly).

Περιεχόμενα Εικόνων

Περιεχόμενα Πινάκων

Contents of Figures

Contents of Tables

References

[1] **Imagenet classification with deep convolutional neural networks**
by A. Krizhevsky, I. Sutskever, G.E. Hinton
Site: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] **GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification** by Maayan Frid Adar , Idit Diamant, Eyal Klang, Michal Amitai, Jacob Goldberger, Hayit Greenspan
Site:https://www.sciencedirect.com/science/article/pii/S0925231218310749#bib001

[3] **Statoil /C-CORE Iceberg Classifier Challenge**
Site:https://www.kaggle.com/c/statoil-iceberg-classifier-challenge/overview

[4] **Ships in Satellite Imagery**
Site: https://www.kaggle.com/rhammell/ships-in-satellite-imagery

[5] Site:https://www.mnn.com/money/sustainable-business-practices/stories/drifting-icebergs-threaten-ships

[6] Site:https://nunatsiaq.com/stories/article/110810_giant_iceberg_could_threaten_ships_oil_platforms/

[7] **Satellite image processing for precision agriculture and agroindustry using convolutional neural network and genetic algorithm** by Firdaus , Y. Arkeman , A. Buono and I. Hermadi
Site: https://iopscience.iop.org/article/10.1088/1755-1315/54/1/012102/pdf

[8] **TensorFlow for Deep Learning** by Reza Bosagh Zadeh, Bharath Ramsundar (Chapter 4)
Site: https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html

[9] **Understanding Activation Functions in Neural Networks** by Avinash Sharma
Site: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

[10] **Keras Optimizers**
Site: https://www.kaggle.com/residentmario/keras-optimizers

[11] **A Look at Gradient Descent and RMSprop Optimizers**
Site: https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b

[12] **Gentle Introduction to the Adam Optimization Algorithm for Deep Learning**
Site: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[13] Site: https://www.datascience.com/blog/convolutional-neural-network

[14] Site: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[15] Site: https://elitedatascience.com/overfitting-in-machine-learning

[16] **Dropout: A Simple Way to Prevent Neural Networks from Overfitting**
by Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov
Site: https://www.datopia.ir/wp-content/uploads/2018/12/srivastava14a.pdf

[17] **Data Augmentation | How to use Deep Learning when you have Limited Data - Part 2**
Site: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

[18] Site: https://github.com/jaejin1/Statoil-C-CORE-Iceberg-Classifier-Challenge

[19] **Memory-Centric Accelerator Design for Convolutional Neural Networks**
by Peemen, M. C. J., Setio, A. A. A., Mesman, B., & Corporaal, H.
Site: https://pure.tue.nl/ws/files/3997958/580711006684801.pdf

[20] **SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND < 0.5MB MODEL SIZE** by Forrest N. Iandola , Song Han , Matthew W. Moskewicz , Khalid Ashraf , William J. Dally , Kurt Keutzer
Site: https://arxiv.org/pdf/1602.07360.pdf

[21] **Post-Training Quantization of TensorFlow model to FP16**
Site: https://medium.com/@fanzongshaoxing/post-training-quantization-of-tensorflow-model-to-fp16-8d66b9dfa77f

[22] **LeFlow repository**
Site: https://github.com/danielholanda/LeFlow

[23] **hls4ml**
Site: https://fastmachinelearning.org/hls4ml/

[24] Jiantao Qiu, et al. **Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. FPGA** 2016 Site: http://www.isfpga.org/fpga2016/index_files/Slides/1_2.pdf

[25] **Throughput-Optimized OpenCL-based FPGA Acceler-ator for Large-Scale Convolutional Neural Networks. FPGA** by Naveen Suda et al. Site: http://isfpga.org/fpga2016/index_files/Slides/1_1.pdf

[26] **DLAU: A Scalable Deep Learning Accelerator Unit on FPGA** by Chao Wang, Member, IEEE, Qi Yu, Lei Gong, Xi Li, Member, IEEE Yuan Xie, Fellow, IEEE and Xuehai Zhou, Member, IEEE Site: https://arxiv.org/pdf/1605.06894.pdf

[27] **Xilinx Machine Learning Suite** Site: https://www.xilinx.com/products/acceleration-solutions/xilinx-machine-learning-suite.html

[28] **Keras**
Site: https://keras.io

[29] Site: https://towardsdatascience.com/tensorflow-is-in-a-relationship-with-keras-introducing-tf-2-0-dcf1228f73ae

[30] Site: https://www.tensorflow.org/guide/keras/overview

[31] **Xilinx Machine Learning Suite**
https://www.xilinx.com/products/acceleration-solutions/xilinx-machine-learning-suite.html

[32] **Xilinx DNNDK user guide** https://www.xilinx.com/support/documentation/user_guides/ug1327-dnndk-user-guide-190201.pdf

[33] **Repository Xilinx ML Suite**
Site: https://github.com/Xilinx/ml-suite

[34] **Implementation of Convolutional Neural Networks on Embedded Architectures** by Αθανάσιος Α. Ξύγκης Σούντρης Δημήτριος
Site:http://artemis.cslab.ece.ntua.gr:8080/jspui/bitstream/123456789/13550/1/DT2017-0208.pdf

[35] **FPGA Architectures of Deep Convolutional Neural Networks for Satellite Image Classification** by Ρέππας Χρυσοβιτσινός, Δημήτριος and Σούντρης Δημήτριος
Site:http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/17062

[36] Site: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[37] Site: https://www.ni.com/en-us/innovations/white-papers/08/fpga-fundamentals.html

[38] **LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks** by Daniel H. Noronha, Bahar Salehpour, and Steven J.E. Wilton
Site: https://arxiv.org/ftp/arxiv/papers/1807/1807.05317.pdf

[39] Image URL: https://qph.fs.quoracdn.net/main-qimg-330e8b2941bc0164211bbdc7d5c693f3

[40] Image URL:
https://www.google.gr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=2ahUKEwiOpqTA57LlAhXDx4UKHd_PB9EQjRx6BAgBEAQ&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FActivation_function&psig=AOvVaw0TTIex1Z0Vm294nFU19ior&ust=1571934707270662

[41] Image URL:
https://www.google.gr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=2ahUKEwjYicn857LlAhWOzYUKHTJhCWoQjRx6BAgBEAQ&url=https%3A%2F%2Ftowardsdatascience.com%2Fintroduction-to-artificial-neural-networks-ann-1aea15775ef9&psig=AOvVaw0TTIex1Z0Vm294nFU19ior&ust=1571934707270662

[42] Image URL:
https://www.google.gr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=2ahUKEwiKuc6s6LLlAhWKxoUKHV6NDQIQjRx6BAgBEAQ&url=https%3A%2F%2Fmedium.com%2Fthe-theory-of-everything%2Funderstanding-activation-functions-in-neural-networks-9491262884e0&psig=AOvVaw0vji43LnqNql6byXVwgy3v&ust=1571934992173224

[43] Image URL:
https://www.google.gr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=2ahUKEwiyg4Hd6LLlAhUS6RoKHe3JBycQjRx6BAgBEAQ&url=https%3A%2F%2Fwww.researchgate.net%2Ffigure%2FFigure-316-Rectified-Linear-Unit-94_fig17_328048988&psig=AOvVaw3XXx_pUAzQXBxOgKG79P8z&ust=1571935063947457

[44] Image URL: https://miro.medium.com/max/596/1*KQIEqhxzICU7thjaQBfPBQ.png

[45] Image URL: https://www.oreilly.com/library/view/neural-networks-with/9781788397872/assets/2009c470-759a-4fb7-a1a3-982c4bae841c.png

[46] Image URL: https://help.sketchup.com/sites/help.sketchup.com/files/images/fg016_1.png

[47] Image URL: https://cdn-images-1.medium.com/max/720/1*i_F6aNKj3yggkcNXQxYA4A.jpeg

[48] Image URL: https://cdn-images-1.medium.com/max/720/1*z8_8gq5zgA_9peaTfyx2gQ.jpeg

[49] Image URL: https://cdn-images-1.medium.com/max/720/1*INLTn7GWM-m69GUwFzPOaQ.jpeg

[50] Image URL: https://nanonets.com/blog/content/images/2018/11/1_L07HTRw7zuHGT4oYEMlDig.jpeg

[51] Image URL: https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSF0DYW34dIWZ9LYMFCDYKPBd1K2oD3gfYSrJiPnq8BogaMWXGW&s

[52] Image URL: https://ni.scene7.com/is/image/ni/swvvifhq55851?scl=1

[53] Image URL: https://i.imgur.com/tLsSoTz.png

[54] Image URL: https://i.imgur.com/Q3daQMC.png

[55] Image URL: http://3s81si1s5ygj3mzby34dq6qf-wpengine.netdna-ssl.com/wp-content/uploads/2018/07/FPGATens1.png

[56] Image URL: https://images.exxactcorp.com/CMS/technologies/fpga-solutions/xilinx-alveo-accelerator-solutions/Xilinx-Stack-Chart.jpg

[57] Site: https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#specifications

[58] Site: https://www.apriorit.com/dev-blog/586-fpgas-for-ai