Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Design and Evaluation of Approximation Techniques using Approximate Multipliers on Deep Neural Networks

Διπλωματικη Εργασια

του

ΓΕΩΡΓΙΟΥ ΜΑΚΡΗ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθήνα, Ιούνιος 2021

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

Design and Evaluation of Approximation Techniques using Approximate Multipliers on Deep Neural Networks

Διπλωματικη Εργασια

του

ΓΕΩΡΓΙΟΥ ΜΑΚΡΗ

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24η Ιουνίου 2021.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π

Αθήνα, Ιούνιος 2021

(Υπογραφή)

.....

ΓΕΩΡΓΙΟΥ ΜΑΚΡΗ

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

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

Περίληψη

Την τελευταία δεκαετία τα Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ) αναδείχθηκαν ως μία από τις καλύτερες προσεγγίσεις για την αντιμετώπιση ορισμένων προκλήσεων της Όρασης Υπολογιστών, όπως η ταξινόμηση εικόνων. Οι τελευταίες έρευνες δείχνουν σαφώς ότι τα νευρωνικά δίκτυα διαθέτουν μια εγγενή ιδιότητα ανθεκτικότητας στα σφάλματα. Απο τη στιγμή που οι χρήστες διατίθενται να ανεκτούν ορισμένα σφάλματα σε κάποιες περιπτώσεις, οι αρχές του approximate computing μπορούν να επιστρατευτούν για την σχεδιάση εφαρμογών αποδοτικών ως προν την κατανάλωση ενέργειας.

Στη παρούσα διπλωματική, στοχεύουμε στην ανάπτυξη νέων προσεγγιστικών τεχνικών που προσφέρουν ένα καλό trade-off μεταξύ ενέργειας και σφάλματος στην ακρίβεια, πραγματοποιώντας μια διεξοδική εξερεύνηση χώρου για να βρούμε τις βέλτιστες δυνατές λύσεις. Επεκτείναμε την βιβλιοθήκη [1], που παρέχει συνελικτικά layer με μειωμένη ακρίβεια χρησιμοποιώντας προσεγγιστικούς πολλαπλασιαστές, σχεδιάζοντας τέσσερις νέες προσεγγιστικές τεχνικές. Στην πρώτη τεχνική ακολουθήσαμε μια μη ομοιόμορφη δομή ανά layer. Στην δεύτερη τεχνική, διαιρέσαμε το πλήθος των φίλτρων σε κάθε layer σε k ισοδύναμα μέρη και αναθέσαμε σε κάθε μέρος ένα διαφορετικό προσεγγιστικό πολλασιαστή. Στην τρίτη τεχνική, πραγματοποιήσαμε προσεγγίσεις μέσα στα φίλτρα είτε αντικαθιστώντας τους πολλαπλασιασμούς με διαφορετικούς προσεγγιστικούς πολλαπλασιαστές είτε παραλείποντας τους. Στην τέταρτη και τελευταία τεχνική παρατηρήσαμε ότι τα βάρη των φίλτρων ακολουθούν κανονική κατανομή ανα layer και παραλείψαμε κάποιους πολλαπλασιασμούς βασιζόμενοι σε αυτό. Η αξιολόγηση των τεχνικών πραγματοποιήθηκε στο Tensorflow πάνω στο νευρωνικό δίκτυο το Resnet-8 και χρησιμοποιώντας το validation set του CIFAR-10 καθώς και τρεις προσεγγιστικούς πολλαπλασιαστές. Για να αξιολογήσουμε τις τεχνικές, εξετάσαμε την ακρίβεια και την ενέργεια που απαιτείται για το inference μίας εικόνας. Τα τελικά αποτελέσματα έδειξαν ότι η τρίτη και η δεύτερη τεχνική είναι οι καλύτερες καθώς παρέχουν σημαντική εξοικομόνηση στην ενέργεια ως και 33.5% ανδ 30.4% αντίστοιχα, συγκρινόμενες με την υλοποιήση με τον ακριβή πολλαπλασιαστή, έχοντας παράλληλα πολύ μικρή πτώση στην ακρίβεια.

Λέξεις Κλειδιά

Προσεγγιστικοί υπολογισμοί, Συνελικτικά Νευρωνικά Δίκτυα, Προσεγγιστικοί Πολλαπλασιαστές, Tensorflow, Resnet-8, Ενεργειακή Απόδοση, Ακρίβεια Πρόβλεψης

Abstract

Over the past decade Convolutional Neural Networks (CNNs) emerged as the state-of-the-art approach to tackle certain Computer Vision problems such as image classification and object detection.The state-of-the-art works clearly indicate that neural networks feature an intrinsic error-resilience property. Since they often process noisy or redundant data and their users are willing to accept certain errors in many cases, the principles of approximate computing can be employed in the design of their energy efficient implementations.

In this thesis, we target the development of novel approximation techniques that provide a good trade-off between between energy consumption, and inference accuracy, by performing an in-depth design space exploration to find optimal solutions. We extended the Tensorflow Approximate Layers library [1], which provides convolutional layers with reduced precision implemented using approximate multipliers, by designing and developing four new approximation techniques in an effort to find the optimal solutions. In the first technique we followed a non-uniform structure per layer. In the second technique, we split the number of filters in each layer into k equivalent parts and assign in each of these parts a different approximate multiplier. In the third technique, we performed approximations inside the filters by either replacing the multiplications i.e., partial products, with diverse approximate components or simply skipping this operations i.e, not executing them at all. In the fourth and final approximation technique we observed that the filter weights of each layer follow a normal distribution and based on this we proposed to execute only the multiplications that have filter weights that belong in either this range $[\mu - \sigma, \mu + \sigma]$ or this $[\mu - 2\sigma, \mu + 2\sigma]$. The evaluation of our proposed techniques is performed in Tensorflow with Resnet-8 using the validation set from CIFAR-10 and three inexact multipliers with different perforation, by examining the inference accuracy and energy for the inference of one input image. The final results show that the third and second technique are the best since they provide a significant energy saving up to 33.5% and 30.4% compared to the accurate implementation respectively with a negligible drop in the inference accuracy.

Keywords

Approximate Computing, Deep Neural Network, Resnet-8, CNN, Approximate Multipliers, Tensorflow, CIFAR-10, Energy Efficiency, Inference Accuracy

Ευχαριστίες

Στο σημείο αυτό ολοκληρώνεται ένας πολύ σημαντικός κύκλος της ζωής μου και κλείνει το ιδιαίτερα όμορφο κεφάλαιο των προπτυχιακών σπουδών στο Εθνικό Μετσόβιο Πολυτεχνείο. Η πορεία αυτή εμπλουτίστηκε από πολλούς ανθρώπους, τους οποίους νιώθω την ανάγκη να ευχαριστήσω από τα βάθη της καρδιάς μου. Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της παρούσας διπλωματικής εργασίας, τον κύριο Δημήτριο Σούντρη που μου έδωσε τη δυνατότητα και την ευκαιρία να εκπονήσω ένα θέμα διπλωματικής άκρως ενδιαφέρον για εμένα.

Στην συνέχεια θα ήθελα να εκφράσω την ευγνομωσύνη μου προς τον μεταδιδακτορικό ερευνητή Σωτήρη Ξύδη και τον υποψήφιο διδάκτορα Βασίλη Λέων, οι οποίοι συνέβαλαν σε σημαντικό βαθμό τοσο στην καλλιέργεια του τρόπου σκέψης μου ως προς την προσέγγιση σχετικών ζητημάτων, όσο και στην καλλιέργεια και ανάπτυξη νέων ιδεών και κατευθύνσεων που αφορούσαν την εκπόνηση αυτης της εργασίας. Θέλω να τους ευχαριστήσω ιδιαιτέρως για τον χρόνο που αφιέρωσαν ώστε να ανταλλάξουμε σκέψεις και ιδέες σχετικά με ζητήματα που προέκυψαν κατά την διάρκεια της διπλωματικής.

Θα ήθελα να ευχαριστήσω τους φίλους μου που βρίσκονται στο πλευρό μου χρόνια και ομορφαίνουν τη ζωή μου. Τα παιδιά από το σχολείο, τον Γιάννη και τον Φάνη που μεγαλώσαμε μαζί και έχουμε μοιραστεί τόσες ωραίες στιγμές. Ακόμη, τους ανθρώπους που γνώρισα στα φοιτητικά μου χρόνια, τον Βασίλη και τον Βασίλη, το Στέλιο,το Νίκο, τον Τόλη, τον Λευτέρη, τον Αστεριο, τον Θανάση και φυσικά την κοπέλα μου την Τζένη. Τους ευχαριστώ πολύ ιδιαίτερα για την τελευταία περίοδο που υπέμειναν τις ιδιοτροπίες μου και τις ανησυχίες μου σε καθημερινή βάση και με βοήθησαν με τις πολύτιμες συμβουλές τους.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου που από μικρή ηλικία έθεσαν τα θεμέλια για την πορεία μου, με παρότρυναν να αδράξω ευκαιρίες και προσέφεραν καθοδήγηση και ενθάρρυνση σε κάθε βήμα. Στέκονται δίπλα μου στα καλά και τα κακά, πάντα με υπομονή και αγάπη. Μέσα από προσωπικές και καθημερινές θυσίες καλλιέργησαν το περιβάλλον που μου επιτρέπει να κυνηγήσω τα όνειρά μου.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Εισαγωγή

Την τελευταία δεκαετία, τα Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ) αναδείχθηκαν ως ένας από τους καλύτερους τρόπους για την επίλυση ορισμένων προβλημάτων από το πεδίο της Όρασης Υπολογιστών, όπως ειναι η ταξινόμηση εικόνων και η αναγνώριση αντικειμένων. Τα Συνελικτικά Νευρωνικά Δίκτυα είναι μία υποκατηγορία των Τεχνητών Νευρωνικών Δικτύων και ειναι εμνευσμένα απο τον τρόπο με τον οποίο λειτουργεί ο ανρθώπινος εγκέφαλος. Συνιστούν ένα πολυ σημαντικό κομμάτι της Μηχανικής Μάθησης.

Πολλές εφαρμογές οι οποίες ειναι πολύ δαπανηρές όσον αφορά του υπολογιστικούς πόρους (για παράδειγμα η αναγνώριση εικόνων, η εξόρυση δεδομένων και η επεξεργασία εικόνας και βίντεο) διαθέτουν μια εσωτερική ιδιότητα ανθεκτικότητας σε σφάλματα. Απο τη στιγμή που οι χρήστες δέχονται σε ορισμένες περιπτώσεις ορισμένα σφάλματα στις εφαρμογές τους, οι αρχές των προσσεγιστικών υπολογισμώνapproximate computing μπορούν να αξιοποιηθούν.

Το approximate computing ειναι μία ενναλακτική προσεγγιση σχεδιασμού που εκμευταλλεύεται την εσωτερική ιδιότητα ανθεκτικότητας σε σφάλματα των εφαρμογών αυτών και μπορεί να χαλαρώσει την ακρίβεια στους υπολογισμούς προκειμένου να παρέχει σημαντικά κέρδη στην κατανάλωση ενέργειας.

Η διπλωματική αυτη αποσκοπεί στα εξής:

- Να μελετήσουμε και να καταλάβουμε πως λειτουργεί η βιβλιοθήκη [1]. Αυτή ειναι η βιβλιοθήκη πάνω στην οποία βασίστηκε αυτή η διπλωματική.

- Να επεκτείνουμε αυτή τη βιβλιοθήκη προκειμένου να μπορεί να υποστηρίξει πολλαπλούς προσεγγιστικούς πολλαπλασιαστές ταυτόχρονα.

- Να επεκτείνουμε ξανά αυτη τη βιβλιοθήκη αναπτύσοντας και σχεδιάζοντας νέες προσεγγιστικές τεχνικές σε διαφορετικά επίπεδα μέσα στο Βαθύ Νευρωνικό Δίκτυο.

- Να αναλύσουμε τα πειραματικά αποτελέσματα που προκύπτουν απο καθε μία απο τις προτεινόμενες τεχνικές μας προκειμένου να ελέγξουμε την αποτελεσματικότητα της καθεμίας μετρώντας την ακρίβεια του νευρωνικού δικτύου καθώς και την ενέργεια που απαιτείται για μία εικόνα, χρησιμοποιώντας διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιασμών.

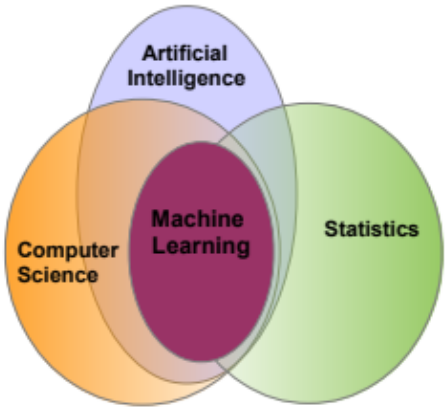
- Να ελέγξουμε την αποτελεσματικότητα της καθεμίας και να συμπεράνουμε ποιες ειναι οι καλύτερες τεχνικές συγκρίνοντας τις με τα αποτελεσματα που παίρνουμε όταν χρησιμοποιούμε τον ακριβή πολλαπλασιαστή.

- Να συγκρίνουμε τις προτεινομένες προσεγγιστικές τεχνικές μας με προσεγγιστικούς πολλαπλαστές απο την EvoApproxLib.

Θεωρητικό Υπόβαθρο

Μηχανική Μάθηση

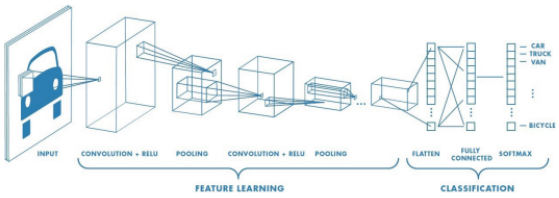
Η Μηχανική Μάθηση συναντιέται σε αρκετές πτυχές της καθημερινότητας των ανθρώπων. Η κατηγοριοποίηση αλληλογραφίας, το σύστημα προτάσεων διαφημίσεων ή προτάσεων σχετικών με τα ενδιαφέροντα του χρήστη, στην αναγνώριση κειμένου και φωνής ακόμα και σε αυτόνομα αυτοκίνητα ή αυτοματοποιήσεις διάφορων εργασιών που εκτελούνται από τον άνθρωπο. Η τεχνητή νοημοσύνη θέτει τα ερώτηματα ¨Τι είναι η ευφυΐα και πώς αυτή δουλέυει' και ¨Μπορούμε να φτιάξουμε έξυπνες μηχανές¨. Όπως υποδεικνύει και το όνομα, με την μηχανική μάθηση προσπαθούμε να εκπαιδεύσουμε τους υπολογιστές ώστε να μάθουν να λύνουν προβλήματα χωρίς να είναι εκτενώς προγραμματισμένοι για αυτόν τον σκοπό. Ένας πιο επίσημος ορισμός για την μάθηση είναι ο εξής : ' ¨Ενα πρόγραμμα υπολογιστή, λέγεται οτι μαθαίνει από την εμπειρία E με δεδομένη κάποια εργασία T και κάποια μέτρηση της απόδοσης Π, αν η απόδοση του στην T όπως μετριέται απο την Π βελτιώνεται με την εμπειρία E¨. Στον πυρήνα της υπάρχει η υπόθεση οτι η γνώση μπορεί να αναχθεί από τα δεδομένα. Η Μηχανική Μάθηση προχωράει ένα βήμα μπροστά από τις ¨έξυπνες' μηχανές και υπόσχεται μια μεγαλύτερου εύρους και βάθους αυτοματοποίηση στις ανθρώπινες δραστηριότητες. Οι μονότονες και επαναληπτικές εργασίες σε έναν σύγχρονο κόσμο θα εκτελούνται από μηχανές.



Σχήμα 1: Συσχέτιση της Μηχανικής Μάθησης σε σχέση με άλλα επιστημονικά πεδία

Συνελικτικά Νευρωνικά Δίκτυα

Τα Συνελικτικά Νευρωνικά Δίκτυα είναι μια ιδιαίτερη κατηγορία νευρωνικών δικτύων για επεξεργασία δεδομένων τα οποία έχουν μια γνωστή τοπολογία πλέγματος. Είναι φτιαγμένα από νευρώνες οι οποίοι έχουν εκπαιδεύσιμα βάρη και πολώσεις. Κάθε νευρώνας δέχεται κάποια είσοδο, εκτελεί ένα εσωτερικό γινόμενο το οποίο προαιρετικά ακολουθείται από μια μη γραμμικότητα. Αυτά τα δίκτυα έχουν εμπνευστεί από το μοντέλο το οποίο τα θηλαστικά λαμβάνουν την πληροφορία του κόσμου γύρω τους χρησιμοποιόντας μια κατάλληλη συστοιχία βιολογικών νευρώνων στον εγκέφαλο τους για να αναγνωρίσουν το αντικείμενο αυτό. Ως παράδειγμα μπορούμε να θεωρήσουμε ένα αυτοκίνητο και να εξετάσουμε τον τρόπο που ένας άνθρωπος το αγαγνωρίζει. Ο άνθρωπος ψάχνει για χαρακτηριστικά τα οποία ξεχωρίζουν ένα αυτοκίνητο από άλλα μέσα μεταφοράς όπως είναι οι τροχοί, τα μπροστινά φώτα, οι πόρτες, το ντεπόζιτο, το καπό, οι καθρέφτες και το παρμπρίζ και άλλα. Παρόμοια όταν αναγνωρίζει έναν τροχό ψάχνει για αντικείμενα κυκλικού σχήματος, σκούρου χρώματος τα οποία βρίσκονται κάτω από την κύρια δομή του αυτοκινήτου. Όλες αυτές οι μικρές πληροφορίες συνδυάζονται μαζί για να σχηματίσουν ένα συγκεκριμένο χαρακτηριστικό το οποίο είναι μοναδικό σε ένα αντικείμενο το οποίο αναγνωρίζουμε. Κάθε επίπεδο ενός συνελικτικού νευρωνικού δικτύου σχετίζεται με την παραγωγή πληροφορίας από τιμές οι οποίες έρχονται από προηγούμενα επίπεδα σε μια ακόμα πιο σύνθετη πληροφορία και την περαιτέρω διάδοση της σε επόμενα επίπεδα για να γινει περαιτέρω γενικοποίηση.



Σχήμα 2: Παράδειγμα δομής συνελικτικού δικτύου για την αναγνώριση μέσων μεταφοράς

Προσεγγιστικοί Υπολογισμοί

Η φορητή και ενσωματωμένη φύση των υπολογιστικών συστημάτων της εποχής μας, έχει οδηγήσει σε μια αυξημένη ανάγκη για υπερβολικά χαμηλή κατανάλωση ισχύος, μικρή επιφάνεια και υψηλή απόδοση. Ο προσεγγιστικός υπολογισμός είναι ένα αναδυόμενο υπολογιστικό πρότυπο που μας επιτρέπει να επιτυγχάνουμε αυτά τα θετικά κάνοντας συμβιβασμούς στην αριθμητική ακρίβεια. Πολλά συστήματα σε τομείς όπως τα πολυμέσα, τα νευρωνικά δίκτυα και η ανάλυση βιγ δατα παρουσιάζουν μια έμφυτη ανοχή σε ένα συγκεκριμένο επίπεδο ανακρίβειας στους υπολογισμούς και μπορούν να ωφεληθούν από τους προσεγγιστικούς υπολογισμούς. Οι υπολογιστικές και αποθηκευτικές απαιτήσεις των μοντέρνων συστημάτων έχουν ξεπεράσει κατά πολύ τους διαθέσιμους πόρους. Προβλέπεται ότι στην επερχόμενη δεκαετία, ο όγκος της πληροφορίας την οποία διαχειρίζονται τα παγκόσμια κέντρα δεδομένων θα αυξηθεί κατά πενήντα φορές, καθώς ο αριθμός των διαθέσιμων επεξεργαστών θα αυξηθεί μόνο κατά δέκα

φορές. Στην πραγματικότητα, η κατανάλωση ηλεκτρικής ενέργειας μόνο των κέντρων δεδομένων της Αμερικής αναμένεται να αυξηθεί από 61 δισεκατομμύρια κιλοβατώρες το 2006 και 91 δισεκατομμύρια κιλοβατώρες το 2013 σε 140 δισεκατομμύρα κιλοβατώρες το 2020. Είναι προφανές ότι η απαίτηση για αυξανόμενη απόδοση σύντομα θα ξεπεράσει την ανάπτυξη στους διαθέσιμους πόρους. Οπότε η αποκλειστική παροχή πόρων δεν θα λύσει τον γρίφο της βιομηχανίας στο κοντινό μέλλον. Η προσεγγιστική λειτουργία στο υλικό, κυρίως ασχολείται με την σχεδίαση προσεγγιστικών αριθμητικών μονάδων, όπως είναι οι αθροιστές και οι πολλαπλασιαστές, σε διαφορετικά επίπεδα αφαίρεσης όπως είναι τα τρανζίστορ, το κυκλωματικό, το επίπεδο πυλών και της εφαρμογής. Μερικοί αξιοσημείωτοι προσεγγιστικοί αθροιστές περιέχουν υποθετικούς αθροιστές, κατακερματισμένους αθροιστές και προσεγγιστικούς πλήρεις αθροιστές. Επίσης, στον τομέα των προσεγγιστικών πολλαπλασιαστών, οι οποίοι είναι το πιο απαιτητικό σε θέμα πόρων στοιχείο του υλικού, έχει γίνει σημαντική έρευνα.

Οι Προσεγγιστικοί Υπολογισμοί και η αποθήκευση πλεονεκτούν στην παρουσία ανεκτικών στα σφάλματα περιοχών κώδικα σε εφαρμογές και προφανείς περιορισμούς των χρηστών να διαχειριστούν έξυπνα την υλοποίηση, την αποθήκευση και την ακρίβεια του αποτελέσματος για πλεονεκτήματα στην απόδοση ή την ενέργεια. Στην πραγματικότητα ο προσεγγιστικός υπολογισμός εκμεταλλεύεται το κενό μεταξύ του επιπέδου ακρίβειας που απαιτείται από την εφαρμογή ή τον χρήστη και αυτού που παρέχεται από το υπολογιστικό σύστημα για να κάνει τις κατάλληλες βελτιστοποιήσεις.

Προσεγγιστικά Αριθμητικά Κυκλώματα

Προσεγγιστικοί Αθροιστές

Σε προσεγγιστικές υλοποίησεις, οι αθροιστές πολλών βιτ χωρίζονται σε δύο διαφορετικά μέρη : το ακριβές πάνω μέρος των πιο σημαντικών βιτ και το προσεγγιστικό κάτω μέρος των λιγότερο σημαντικών βιτ. Για κάθε χαμηλό βιτ , ένας προσεγγιστικός αθροιστής του ενός βιτ πραγματοποιεί μια τροποποιημένη, επομένως ανακριβή διαδικασία της πρόσθεσης. Αυτό συνήθως επιτυγχάνεται με την απλοποίηση ενός πλήρους αθροιστή σε κυκλωματικό επίπεδο, αντίστοιχα με μια διαδικασία η οποία αλλάζει κάποιες εισόδους στον πίνακα αληθείας ενός πλήρους αθροιστή σε λειτουργικό επίπεδο.

Προσεγγιστικοί Πολλαπλασιαστές

Σε αντίθεση με τον σχεδιασμό προσεγγιστικών αθροιστών, η σχεδίαση προσεγγιστικών πολλαπλασιαστών δεν έχει ερευνηθεί στο έπακρο. Προσεγγιστικοί πολλαπλασιαστές οι οποίοι χρησιμοποιούν τους υποθετικούς αθροιστές για να υπολογίσουν το άθροισμα των μερικών γινομένων έχουν σχεδιαστεί. Παρόλα αυτά, η άμεση εφαρμογή των προσεγγιστικών αθροιστών σε έναν πολλαπλασιαστή ίσως είναι μη αποδοτική όσον αφορά την ανταλλαγή ακρίβειας για εξοικονόμηση σε ενέργεια και επιφάνεια. Ένα σημαντικό στοιχείο στην σχεδίαση ενός προσεγγιστικού πολλαπλασιαστή, είναι η μείωση του κρίσιμου μονοπατιού στην άθροιση των μερικών γινομένων. Ο πολλαπλασιασμός συνήθως υλοποιείται από έναν συνδεδεμένο πίνακα

αθροιστών. Στο [4] μερικά λιγότερα σημαντικά βιτς στα μερικά αθροιστές μπορούν να αφαιρεθούν από τον πίνακα οδηγώντας σε γρηγορότερη λειτουργία. Στο [5], ένας μεγάλος πολλαπλασιαστής κατασκευάζεται από 2 × 2 απλοποιημένους πολλαπλασιαστές για να μειώσει την αριθμητική και υπολογιστική πολυπλοκότητα. Ένας αποδοτικός συνδυασμός που χρησιμοποιεί προ-επεξεργασία και επιπρόσθετη αντιστάθμιση σφάλματος προτείνεται στο [6] για να μειώσει την καθυστέρηση του κρίσιμου μονοπατιού. Συνδυασμοί της παραγωγής μερικών γινομένων και προσεγγίσεων εφαρμόζονται συνδυαστικά για περαιτέρω μείωση της κατανάλωσης ισχύος [7],[8],[9]. Ο κύριος στόχος στην σημερινή έρευνα όσον αφορά τους προσεγγιστικούς πολλαπλασιαστές είναι να μειωθεί ο αριθμός των μερικών γινομένων χρησιμοποιώντας υβριδικές κωδικοποιήσεις [10] για να εφαρμόσουμε προσεγγίσεις στην παραγωγή μερικών γινομένων.

Προσεγγιστικά Συνελικτικά Επίπεδα του Tensorflow

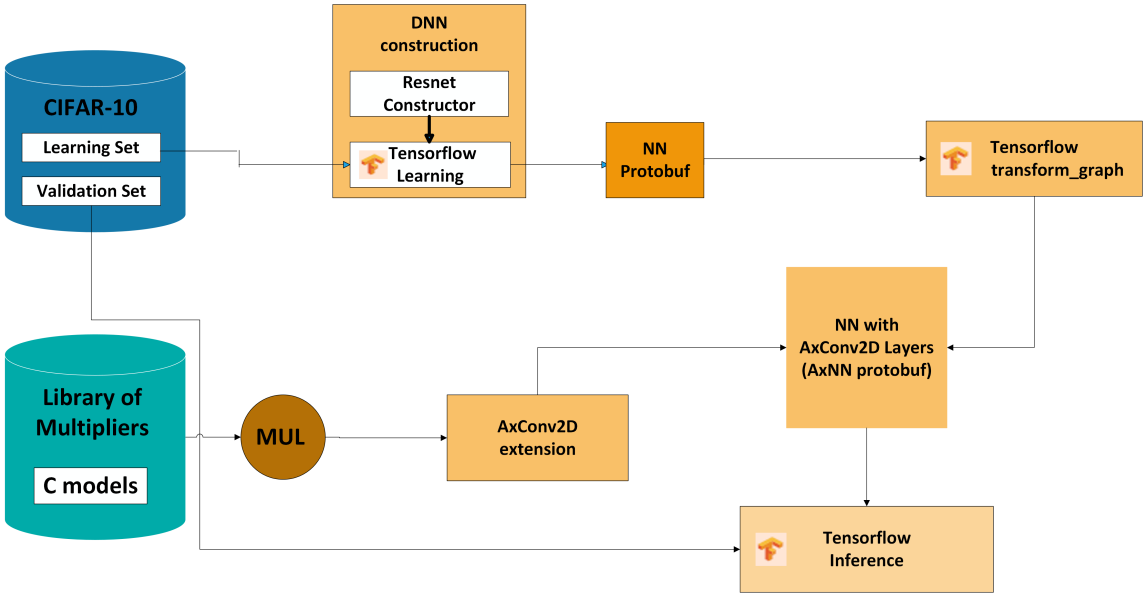
Tensorflow

Το TensorFlow είναι μια βιβλιοθήκη ανοιχτού κώδικα για την κατασκευή μοντέλων μηχανικής μάθησης μεγάλης κλίμακας. 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 sessionσεσσιον).

Το TensorFlow επιτρέπει στον χρήστη να δημιουργήσει διαγράμματα ροής δεδομένων και δομές που ορίζουν πως τα δεδομένα κινούνται μέσα απο το γράφημα. Πάιρνει ως είσοδο πίνακες πολλαπλών διαστάσεων που ονομάζονται Tensor.

Προσεγγιστικά Συνελικτικά Επίπεδα

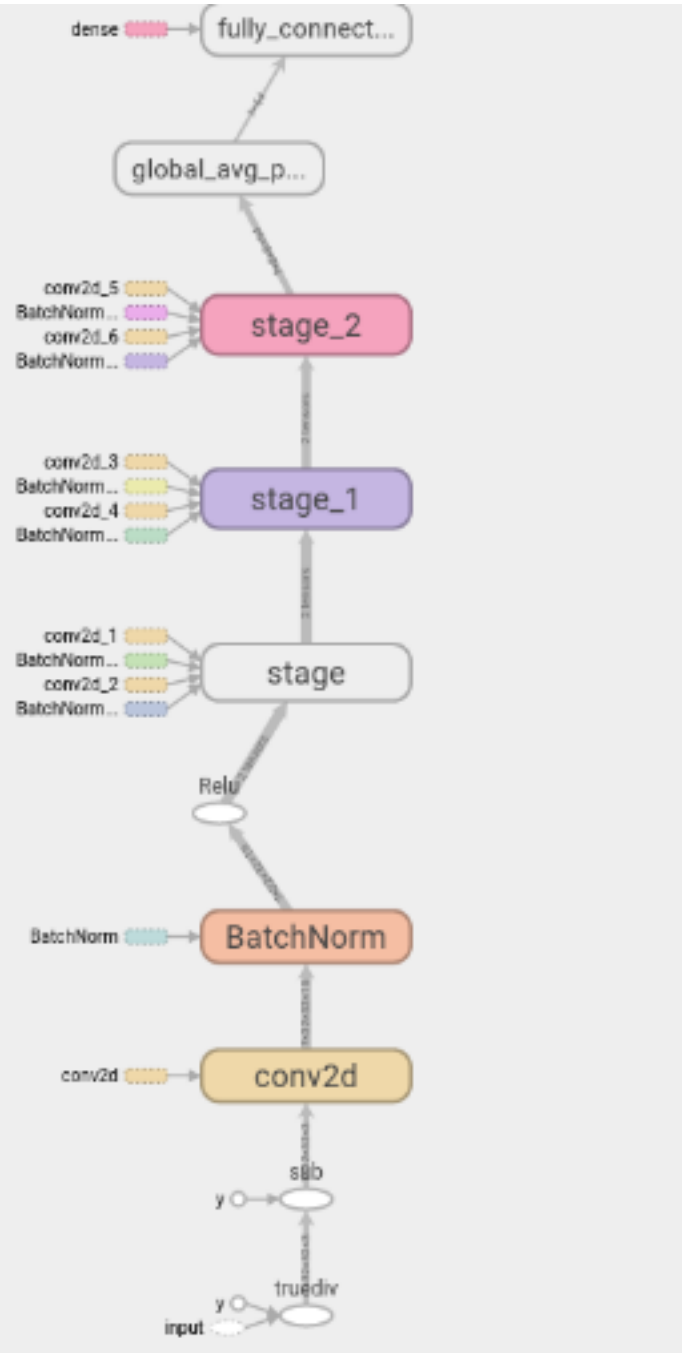
Η βιβλιοθήκη [1] την οποία θέλουμε να επεκτείνουμε σε αυτή η διπλωματική, ειναι μια βιβλιοθήκη ανοιχτού κώδικα που περιλαμβάνει προσεγγιστικά συνελικτικά στρώματα. Η βιβλιοθήκη αυτή επεκτείνει την βιβλιοθήκη του Tensorflow παρέχοντας προσεγγιστικά συνελικτικά στρώματα, τα οποία ειναι στρώματα τα μειωμένης ακρίβεια, συνώθως 8-bit τα οποία υλοποιούνται χρησιμοποιώντας προσεγγιστικούς πολλαπλασιαστές. Πιο συγκεκριμένα η βιβλιοθήκη αυτή επεκτείνει το Tensorflow προσθέτοντας το νεο AxConv2D στρώμα το οποίο υλοποιεί το QuantizedConv2D στρώμα με προσεγγιστικούς πολλαπλασιαστές. Το στρώμα αυτό επιτρέπει μέσω παραμέτρου στον χρήστη να ορίσει ποιον προσεγγιστικό πολλαπλασιαστή θέλει να χρησιμοποιήσει. Οι πολλαπλασιαστές είναι υπολοποιήμενοι σε C. Η βιβλιοθήκη αυτή χρησιμοποιείται μόνο για το inference.



Σχήμα 3: Tensorflow Approximate Layers Library

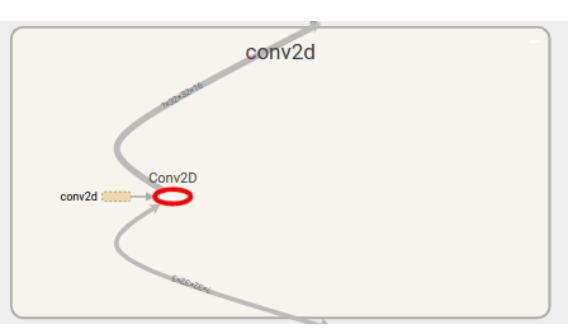
Το νευρωνικό δίκτυο Resnet-8 επιλέγεται και μετα εκπαιδεύται ώστε να αναγνωρίζει εικόνες από το Cifar-10 dataset. Το νευρωνικό δίκτυο που προκύπτει παγώνεται, δηλαδή δημιουργείται το protobuf αρχείο με κατάληξη .pb, χβαντοποιείται και τα συνελικτικά του στρώματα αντικαθιστούνται από τα προσεγγιστικά συνελτικά δικτύα AxConv2D χρησιμοποιώντας προσεγγιστικούς πολλαπλασιαστές, μέσω του εργαλείου του Tensorflow, το trasform graph.

Μπορούμε να οπτικοποιήσουμε τον παγωμένο γράφο , δηλαδή το protobuf αρχείο ενός άλλου εργαλείου του Tensorflow, το Tensorboard. Το παρακάτω σχήμα απεικονίζει το γράφημα του Resnet-8 μέσω του Tensorboard.

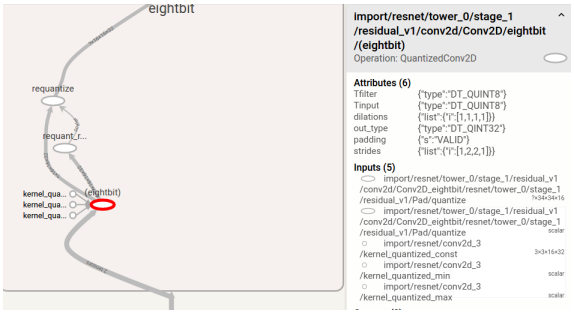


Σχήμα 4: Γράφος του Resnet-8

Οι παρακάτω εικόνες δείχνουν την πράξη της συνέλιξης Conv2D πριν την κβάντιση (αριστερά), και μετά την κβάντιση (δεξιά).



(α′) Συνελτική πράξη μετά την κβάντιση Conv2D

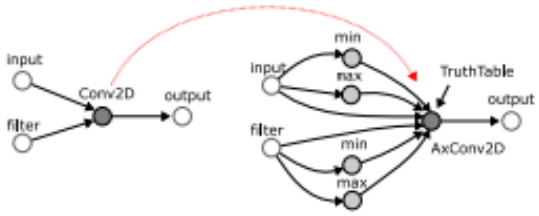


(β′) Συνελτική πράξη μετά την κβάντιση Quantized-Conv2D

Σχήμα 5: Πριν την κβάντιση vs Μετά την κβάντιση

Προκειμένου να μπορούμε να υποστηρίξουμε την χρήση προσεγγιστικών πολλαπλασιαστών στο κβαντισμένο πλέον συνελικτικό στρώμα χρησιμοποιούμε μία εντολή του εργαλείου transform graph, την rename op, η οποία παίρνει σαν είσοδο όνομα της QuantizedConv2D και το μετατρέπει σε AxConv2D.

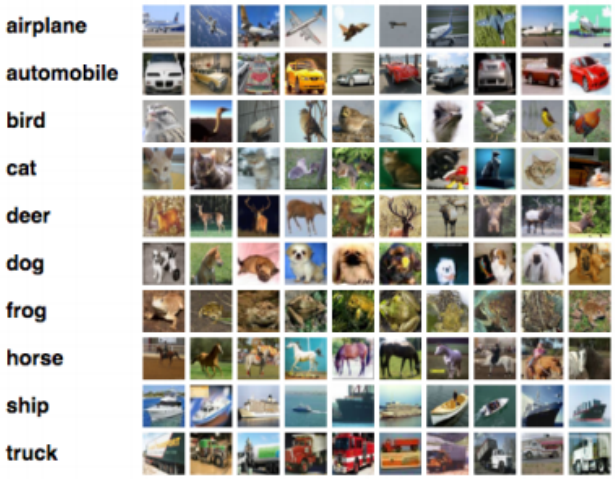
Η πράξη AxConv2D δεν ειναι καταχωρημένη στον πυρήνα του Tensorflow. Στο [1], υλοποείται η πράξη αυτή σε C++ και μετά καταχωρείται στον πυρήνα του Tensorflow. Αυτή η νέα πράξη υποστηρίζει κβάντισμένες(8-bit) των συνελικτικών πράξεων. Η σημαντική διαφορά ανάμεσα στη AxConv2D και τη QuantizedConv2D είναι ότι η AxConv2D υποστηρίζει την χρήση προσεγγιστικών πολλαπλασιαστών στις συνελικτικές πράξεις. Αυτό σημαίνει ότι οι χρήστες μπορούν να δώσουν σαν είσοδο οποιοδήποτε προσεγγιστικό πολλαπλασιαστή και οι πράξεις τις συνέλιξεις να εκτελεστούν με αυτόν τον πολλαπλασιαστή αντί για τον ακριβή.



Σχήμα 6: Εισάγοντας το προσεγγιστικο συνελικτικό στρώμα AxConv2D στον παγωμένο γράφο [2]

CIFAR-10

Το Cifar-10 είναι ενα dataset που αποτελείται απο 60000 32×32 χρωματιστές εικόνες σε 10 διαφορετικές κατηγορίες, με 6000 εικόνες σε κάθε κατηγορία. Υπάρχουν 50000 εικόνες για εκπαίδευση του νευρωνικού δικτύου και 10000 εικόνες για την επαλήθευση του. Το dataset χωρίζεται σε 5 training batches και σε 1 test batch με 10000 εικόνες το καθένα. Το test batch περιλαμβάνει ακριβώς 1000 τυχαίες εικόνες απο κάθε κατηγορία.



Σχήμα 7: Παρουσιάση του CIFAR-10

Προσεγγιστικές Τεχνικές

Επεκτείνουμε την βιβλιοθήκη [1] αναπτύσοντας και δημιουργώντας νέες προσεγγιστικές τεχνικές δημιουργώντας έτσι την δική μας βιβλιοθήκη. Η βιβλιοθήκη μας προσφέρει τις εξής δύο καινοτομίες:

1. Υποστήριξη πολλαπλών προσεγγιστικών πολλαπλασιαστών ταυτόχρονα αντί μόνο ενός.

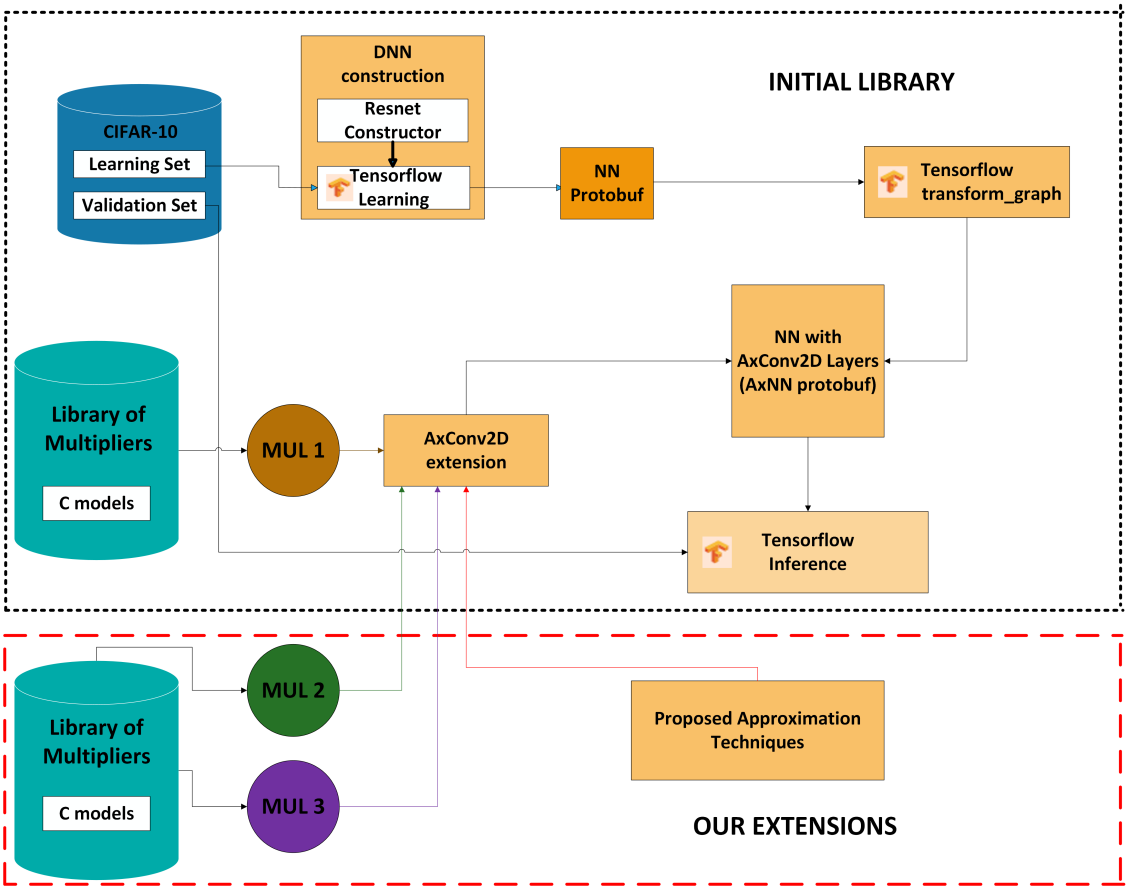
2. Προσεγγιστικές τεχνικές σε διαφορετικά επίπεδα του νευρωνικού δικτύου.

Οι προτεινόμενες προσεγγιστικές τεχνικές πραγματοποιούνται με ουσιαστικά δύο διαφορετικές μεθόδους:

1. Αντικατάσταση των πολλαπλασιασμών με διαφορετικούς προσεγγιστικούς πολλαπλασιαστές

2. Παράλειψη πολλαπλασιασμών

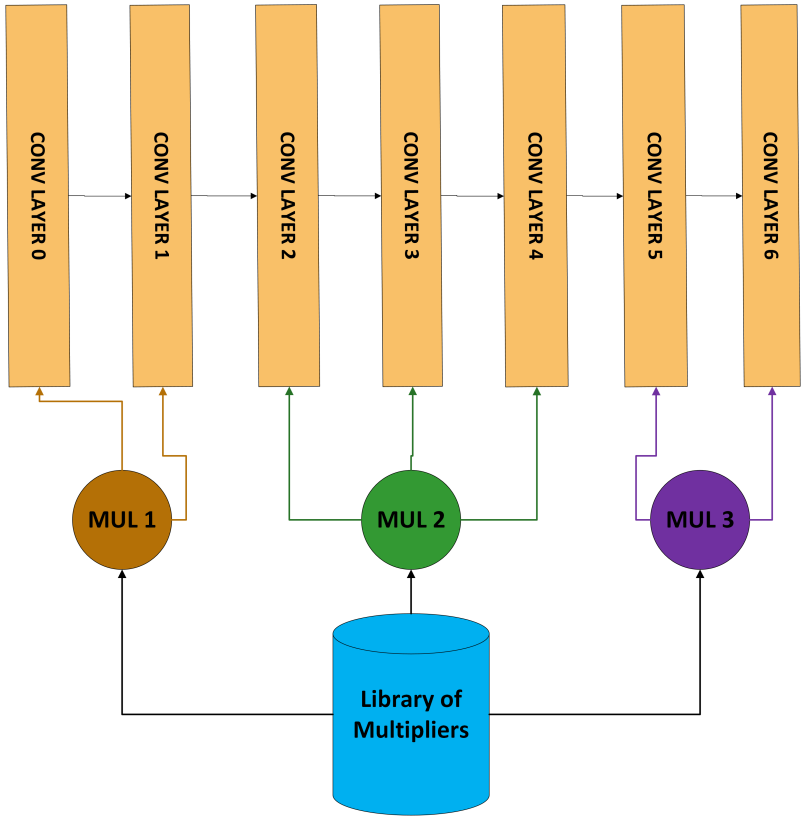
Αυτές οι δύο μέθοδοι χρησιμοποιήθηκαν προκειμένου να αναπτύξουμε και να δημιουργήσουμε τις τεχνικές στα διαφορετικά επίπεδα του νευρωνικού δικτύου και θα εξηγηθούν καλύτερα παρακάτω. Πρέπει να σημειωθεί ότι όλες οι τεχνικές αυτές πραγματοποιήθηκαν στο Resnet-8.



Σχήμα 8: Οι δικές μας επεκτάσεις

Πρώτη προσεγγιστική τεχνική: Ανάμεικτα προσεγγιστικα και ακριβή στρώματα (ετερογενής δομή ανά στρώμα)

Η τεχνική αυτή υλοποιήθηκε αντικαθιστώντας τους πολλαπλασμούς στα διάφορα στρώματα με διαφορεικούς προσεγγιστικούς πολλαπλασιαστές. Αυτό σημαίνει οτι δεν έχουν όλα τα στρώματα του δικτύου τον ίδιο βαθμό προσέγγισης. Κάποια στρώματα εκτελούν τους πολλαπλασιασμούς τους με τον ακριβή πολλαπλασιαστή ενώ άλλα στρώματα με τον προσεγγιστικό. Το Resnet-8 έχει 7 συνελικτικά στρώματα.



Σχήμα 9: Η πρώτη προσεγγιστική τεχνική

Πίνακας 1: Παράδειγμα με τα 4 πρώτα στρώματα να έχουν προσεγγιστικό πολλαπλασιαστή ενω τα τρία τελευταία ακριβή

Layers	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6	Inference Accuracy
Multiplier	$p = 1$	$p = 1$	$p = 1$	$p = 1$	$p = 0$	$p = 0$	$p = 0$	0.826

Στον παραπάνω πίνακα 4.1 ενα παράδειγμα αυτής της προσεγγιστικής τεχνικής δίνεται. Τα τέσσερα πρώτα στρώματα χρησιμοποιούν προσεγγιστικό πολλαπλασιαστή [11] με perforation $p = 1$. Η ακρίβεια που παίρνουμε χρησιμοποιώντας αυτή την τεχνική με τον συγκεκριμένο συνδυασμό πολλαπλασιαστών είναι νούμερο.

Δεύτερη προσεγγιστική τεχνική: Ανάμεικτα ακριβή-προσεγγιστικά φίλτρα ανά στρώμα

Αυτή η τεχνική υλοποιήθηκε αντικαθιστώντας τους πολλαπλασιασμούς με διαφορετικούς προσεγγιστικούς πολλαπλασιαστές. Πιο συγκεκριμένα κάθε ένα απο τα επτά συνελικτικά στρώματα στο Resnet-8 εχει διαφορετικό αριθμό φίλτρων όπως φαίνεται στον παρακάτω πίνακα 4.2:

Πίνακας 2: Αριθμός φίλτρων σε κάθε συνελτικό στρώμα

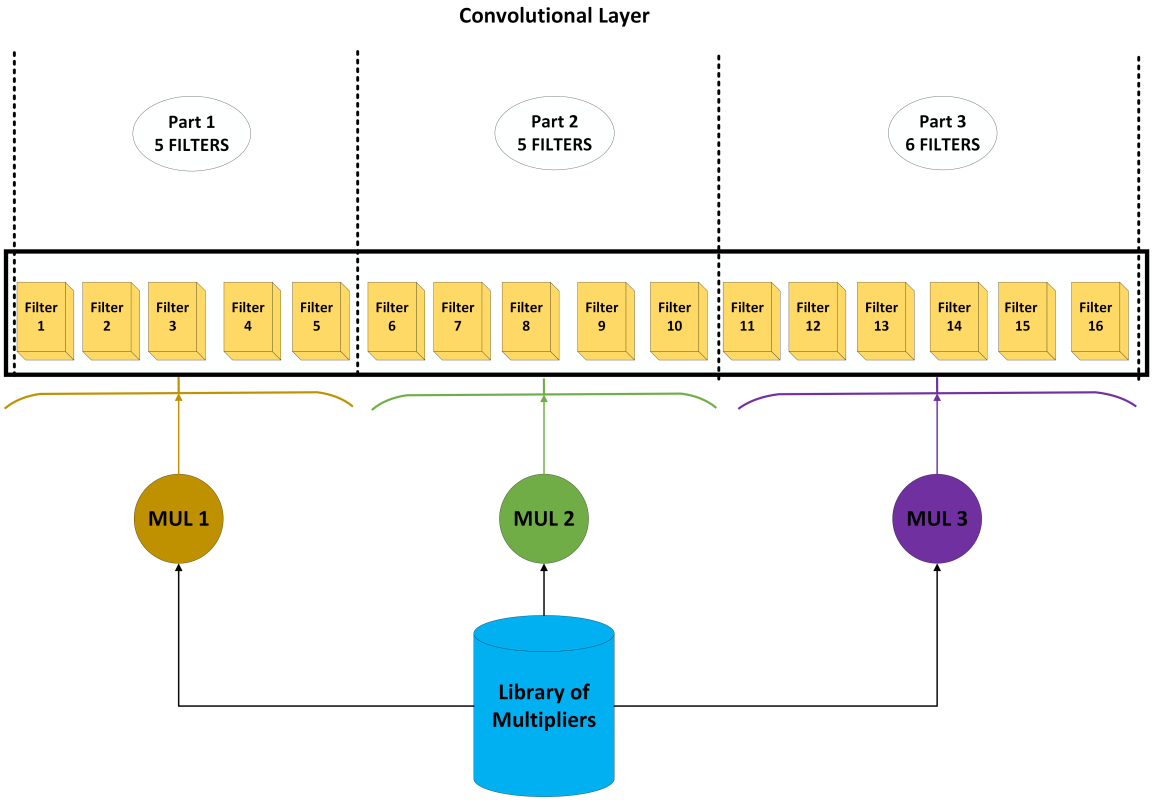
Layers	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6
Αριθμός φίλτρων	16	16	16	32	32	64	64

Σε αυτή την τεχνική σκεφτήκαμε να διαιρέσουμε τον αριθμό των φίλτρων σε κάθε στρώμα σε k ισοδύναμα κομμάτια, έτσι ώστε το άθροισμα αυτών των k κομματιών να ισούται με τον αριθμό των φίλτρων σε κάθε στρώμα 5.1 και η διαφορά ανάμεσα στο μεγαλύτερο και το μικρότερο αριθμό απο την ακολουθία να είναι η ελάχιστη.

Έστω N ο αριθμός των φίλτρων, αν $N \bmod k = 0$, τότε η ελάχιστη διαφορά θα είναι πάντα 0 και η ακολουθία θα αποτελείται απο ίσους αριθμούς που θα είναι ίσοι με $N \div k$. Αλλιώς η διαφορά θα είναι 1 και η ακολουθία θα είναι $N \div k$, $N \div k$, ...,$(N \div k) + 1$, $(N \div k)$ $+1$.

$$part_1 + part_2 + \ldots part_k = N \tag{1}$$

Καθένα απο αυτά τα κομμάτια περιέχει ενα συγκεκριμένο αριθμό φίλτρων. Εμείς αναθέτουμε σε καθένα απο αυτά τα κομμάτια ενα διαφορετικό προσεγγιστικό πολλαπλαστή με διαφορετικό perforation.



Σχήμα 10: Η δεύτερη προσεγγιστική τεχνική

. Για παράδειγμα ας πάρουμε το πρώτο στρώμα το οποίο έχει 16 φίλτρα και ας το διαι-

ρέσουμε σε $k = 3$ ισοδύναμα μέρη τον αριθμό των φίλτρων. Τ πρώτα δύο μέρη θα έχουν απο 5 φίλτρα ενώ το τρίτο θα έχει αναγκαστικά 6 φίλτρα. Σε καθένα απο αυτά τα μέρη αναθέτουμε εναν διαφορετικό προσεγγιστικό πολλαπλασιαστή.

Στον παρακάτω πίνακα δίνεται ενα παράδειγμα αυτής της τεχνικής

Πίνακας 3: Δεύτερη τεχνική χρησιμοποιώντας $k = 3$

$k = 3$	Multiplier1	Multiplier2	Multiplier3	Inference Accuracy
	$p = 1$	$p = 2$	$p = 2$	0.798

Τρίτη προσεγγιστική τεχνική

- **Προσεγγισμοί ανα φίλτρο με αντικατάσταση των πολλαπλασιασμών με διαφορετικούς προσεγγιστικούς πολλαπλασιαστές**

Αυτή η τεχνική υλοποιήθηκε αντικαθιστώντας τους πολλαπλασιασμούς μέσα στο φίλτρο με διαφορετικούς προσεγγιστικούς πολλαπλασιαστές. Κάθε εικόνα απο το Cifar-10 έχει τρεις διαστάσεις: ύψος,πλάτος, βάθος. Το ίδιο ισχύει και για τα φίλτρα. Σε αντίθεση με την προηγούμενη τεχνική , σε αυτήν την τεχνική βρισκόμαστε μέσα στο φίλτρο και θέλουμε να υλοποιήσουμε τους πολλαπλασιαμούς στην πράξη της συνέλιξης χρησιμοποιώντας προσεγγιστικούς πολλαπλασιαστές.

Οι προσεγγιστικοί πολλαπλασιασμοί συμβαίνουν είτε στο ύψος ή στο πλάτος ή στο βάθος. Στους παρακάτω τρεις πίνακες απεικόνιζεται ενα $3 \times 3 \times 3$ φίλτρο, το οποίο αποτυπωνεται ως 3 πίνακες διαστάσων 3×3.

Πίνακας 4: 3×3 φίλτρο για βάθος $= 0$

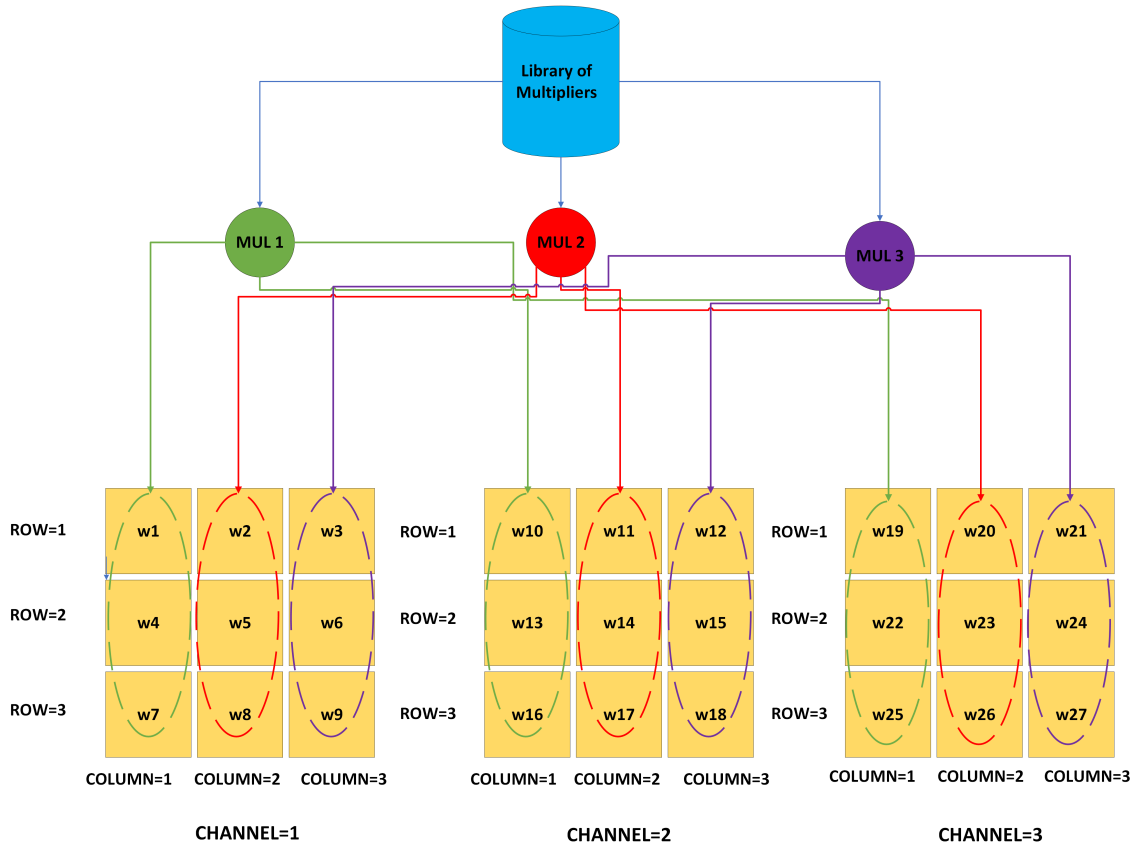
a_1	a_2	a_3
a_4	a_5	a_6
a_7	a_8	a_9

Πίνακας 5: 3×3 φίλτρο για βάθος $= 1$

a_{10}	a_{11}	a_{12}
a_{13}	a_{14}	a_{15}
a_{16}	a_{17}	a_{18}

Πίνακας 6: 3×3 φίλτρο για βάθος $= 2$

a_{19}	a_{20}	a_{21}
a_{22}	a_{23}	a_{24}
a_{25}	a_{26}	a_{27}



Σχήμα 11: Η τρίτη προσεγγιστική τεχνική: Α΄

Για να εξηγηθεί καλύτερα αυτή η τεχνική θα χρησιμοποιήσουμε 3 παραδείγματα:

Στο πρώτο παράδειγμα θα πάρουμε το πρώτο συνελικτικό στρώμα. Οι διαστάσεις των φίλτρων στο στρώμα αυτό ειναι $3 \times 3 \times 3$. Ας πούμε ότι οι πολλαπλασιασμοί για βάθος=0 θα εκτελεστούν χρησιμοποιώντας τον ακριβή πολλαπλασιαστή ενώ οι πολλαπλασιασμοί για βάθος=1 και βάθος=2 θα εκτελεστούν χρησιμοποιώντας τον προσεγγιστικό πολλαπλασιαστή με perforation $p = 2$. Σύμφωνα με τους τρεις παραπάνω πίνακες οι πολλαπλασιασμοί:

1. $a_1 \cdot i_1$, $a_2 \cdot i_2$, ... , $a_9 \cdot i_9$ θα εκτελεστούν με τον ακριβή πολλαπλασιαστή

2. $a_{10} \cdot i_{10}$, $a_{11} \cdot i_{11}$, ..., $a_{18} \cdot i_{18}$ θα εκτελεστούν με τον $p = 2$

3. $a_{19} \cdot i_{19}$, $a_{20} \cdot i_{21}$, ..., $a_{27} \cdot i_{27}$ θα εκτελεστούν με τον $p = 2$

,όπου $i_1, i_2, \ldots, i_{27}$ είναι οι τιμές της στην τρέχουσα τοποθεσία της εικόνας, δηλαδή τα pixel.

Στο τρίτο παράδειγμα θα πάρουμε ξανα το πρώτο συνελικτίκό στρώμα. Ας πούμε ότι οι πολλαπλασιασμοί για πλάτος=0 θα εκτελεστούν χρησιμοποιώντας τον ακριβή πολλαπλασιαστή ενώ οι πολλαπλασιασμοί για πλατος=1 και πλατος=2 θα εκτελεστούν χρησιμοποιώντας τον προσεγγιστικό πολλαπλασιαστή με perforation $p = 2$. Σύμφωνα με τους τρεις παραπάνω πίνακες οι πολλαπλασιασμοί:

1. $a_1 \cdot i_1$, $a_4 \cdot i_4$, $a_7 \cdot i_7$, $a_{10} \cdot i_{10}$, $a_{13} \cdot i_{13}$, $a_{16} \cdot i_{16}$, $a_{19} \cdot i_{19}$, $a_{22} \cdot i_{22}$, $a_{25} \cdot i_{25}$ θα εκτελεστούν με τον ακριβή πολλαπλασιαστή

2. $a_2 \cdot i_2$, $a_5 \cdot i_5$, $a_8 \cdot i_8$, $a_{11} \cdot i_{11}$, $a_{14} \cdot i_{14}$, $a_{17} \cdot i_{17}$, $a_{20} \cdot i_{20}$, $a_{23} \cdot i_{23}$, $a_{26} \cdot i_{26}$ θα εκτελεστούν με τον $p = 2$

3. $a_3 \cdot i_3$, $a_6 \cdot i_6$, $a_9 \cdot i_9$, $a_{12} \cdot i_{12}$, $a_{15} \cdot i_{15}$, $a_{18} \cdot i_{18}$, $a_{21} \cdot i_{21}$, $a_{24} \cdot i_{24}$, $a_{27} \cdot i_{27}$ θα εκτελεστούν με τον $p = 2$

,όπου $i_1, i_2, \ldots, i_{27}$ είναι οι τιμές της στην τρέχουσα τοποθεσία της εικόνας, δηλαδή τα pixel.

Στο τρίτο παράδειγμα θα πάρουμε ξανα το πρώτο συνελικτίκό στρώμα. Ας πούμε ότι οι πολλαπλασιασμοί για ύψος=0 θα εκτελεστούν χρησιμοποιώντας τον ακριβή πολλαπλασιαστή ενώ οι πολλαπλασιασμοί για ύψος=1 και ύψος=2 θα εκτελεστούν χρησιμοποιώντας τον προσεγγιστικό πολλαπλασιαστή με perforation $p = 2$. Σύμφωνα με τους τρεις παραπάνω πίνακες οι πολλαπλασιασμοί:

1. $a_1 \cdot i_1$, $a_2 \cdot i_2$, $a_3 \cdot i_3$, $a_{10} \cdot i_{10}$, $a_{11} \cdot i_{11}$, $a_{12} \cdot i_{12}$, $a_{19} \cdot i_{19}$, $a_{20} \cdot i_{20}$, $a_{21} \cdot i_{21}$ θα εκτελεστούν με τον ακριβή πολλαπλασιαστή

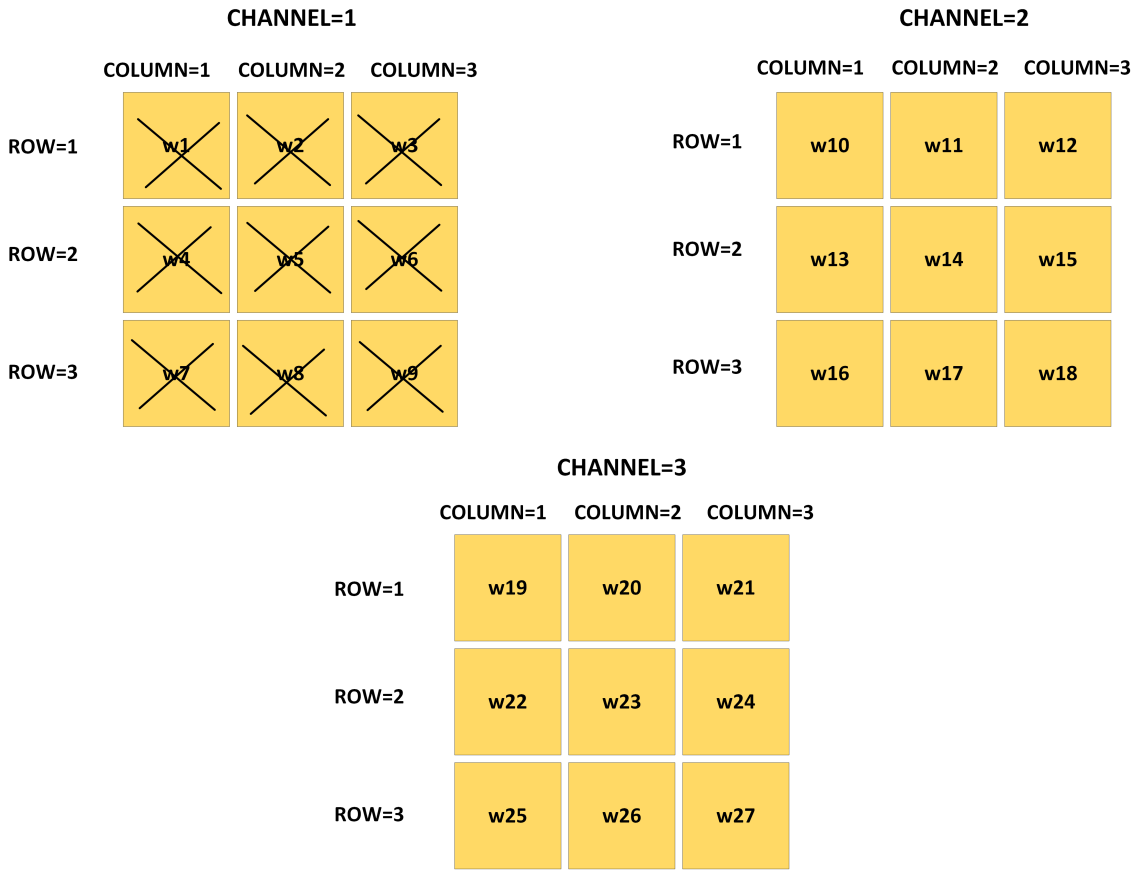
2. $a_4 \cdot i_4$, $a_5 \cdot i_5$, $a_6 \cdot i_6$, $a_{13} \cdot i_{13}$, $a_{14} \cdot i_{14}$, $a_{15} \cdot i_{15}$, $a_{22} \cdot i_{22}$, $a_{23} \cdot i_{23}$, $a_{24} \cdot i_{24}$ θα εκτελεστούν με τον $p = 2$

3. $a_7 \cdot i_7$, $a_8 \cdot i_8$, $a_9 \cdot i_9$, $a_{16} \cdot i_{16}$, $a_{17} \cdot i_{17}$, $a_{18} \cdot i_{18}$, $a_{25} \cdot i_{25}$, $a_{26} \cdot i_{26}$, $a_{27} \cdot i_{27}$ θα εκτελεστούν με τον $p = 2$

,όπου $i_1, i_2, \ldots, i_{27}$ είναι οι τιμές της στην τρέχουσα τοποθεσία της εικόνας, δηλαδή τα pixel.

- **Προσεγγισμοί ανα φίλτρο μέσω της παράλειψης πράξεων**

Η τεχνική αυτή υλοποιήθηκε παραλείποντας κάποιους πολλαπλασιασμούς στην πράξη της συνέλιξης, δηλαδή δεν εκτελέστηκαν καθόλου αυτοί οι πολλαπλασιασμοί. Ουσιαστικά στην τεχνική αυτή διαγράφουμε μερικά γινομένα δηλαδή δεν τα εκτελούμε καθόλου. Τα μερικά αυτα γινομένα έχουν αναφερθεί παραπάνω.



Σχήμα 12: Η τρίτη προσεγγιστική τεχνική: Β′

Για παράδεγιμα ας παρούμε την διάσταση του βάθους. Επιλέγουμε να εκτελέσουμε μόνο τους πολλαπλασιασμούς με βάθος=0. Αυτό σημαίνει ότι εκτελούμε μόνο τους πολλαπλασιασμούς $a_1 \cdot i_1$, $a_2 \cdot i_2$, ... , $a_9 \cdot i_9$, ενώ οι υπόλοιποι πολλαπλασιασμοί $a_{10}, a_{11}, \ldots, a27$ παραλείπονται, δηλαδή δεν εκτελούνται καθόλου.

Το ίδιο συμβαίνει και στις άλλες δύο διαστάσεις. Για παράδειγμα ας πάρουμε την διασταση του πλάτους. Επιλέγουμε να εκτελέσουμε μόνος τους πολλαπλασιασμούς με πλάτος=1. Αυτό σημαίνει ότι εκτελούμε μόνο τους πολλαπλασιασμούς $a_2 \cdot i_2$, $a_5 \cdot i_5$, $a_8 \cdot i_8$, $a_{11} \cdot i_{11}$, $a_{14} \cdot i_{14}$, $a_{17} \cdot i_{17}$, $a_{20} \cdot i_{20}$, $a_{23} \cdot i_{23}$, $a_{26} \cdot i_{26}$, ενώ οι υπόλοιποι πολλαπλασιασμοί παραλείπονται, δηλαδή δεν εκτελούνται καθόλου.

Τέταρτη Προσεγγιστική Τεχνική: Προσεγγίσεις ανα φίλτρο μέσω παράλειψης πράξεων βασισμένη στη κατανομή των τιμών του φίλτρου

Κάθε στρώμα έχει ενα συγκεκριμένο αριθμό φίλτρων. Κάθε $3 \times 3 \times 3$ φίλτρο περιέχει 27 τιμές που ονομάζονται βάρη. Αφου τα βάρη έχουν κβαντιστεί και χρησιμοποιούμε απροσημους πολλαπλασιαστές τα βάρη του κάθε φίλτρου βρίσκονται στο διάστημα $[0, 255]$. Αφου κάθε στρώμα έχει τον δικό του αριθμό φίλτρων ειναι προφανές ότι η κατανομή των βαρών των φίλτρων ειναι διαφορετική ανα στρώμα.

Σε αυτή την προσεγγιστική τεχνική, τυπώσαμε τις τιμές των βαρών όλων των φίλτρων για

κάθε στρωμα και καταλήξαμε στο συμπέρασμα ότι τα βάρη των φίλτρων ακολουθούν κανονική κατανομή ανα στρώμα. Καθε στρώμα έχει την δικιά του μέση τιμή και τυπική απόκλιση των βαρών των φίλτρων.

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n} \tag{2}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}} \tag{3}$$

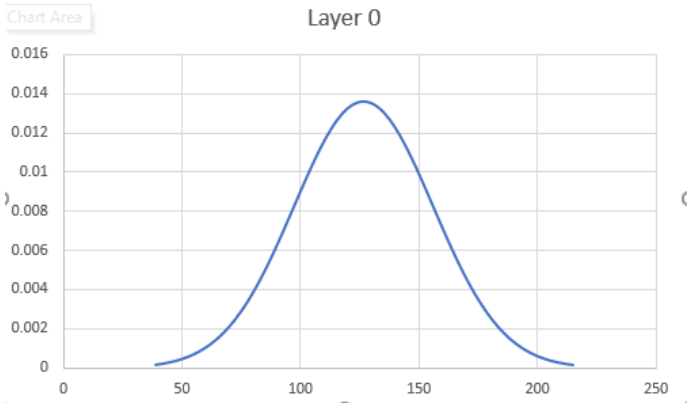
, όπου n ειναι ο αριθμός των όρων και x_i η τιμή του κάθε όρου.

Ο παρακάτω πίνακας 4.7 παρουσιάζει την μέση τιμή και την τυπική απόκλιση των βαρών των φίλτρων για καθε στρώμα ξεχωριστά:

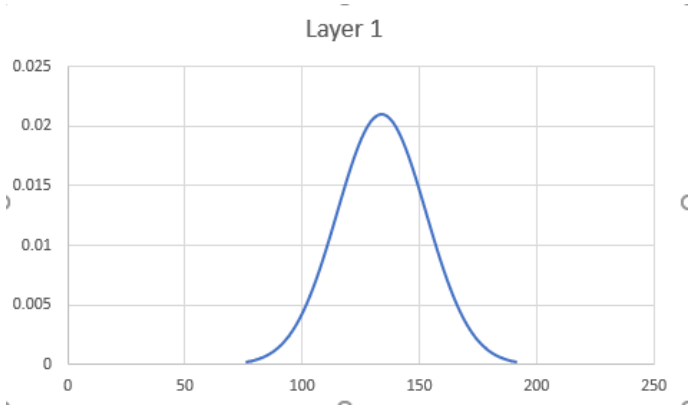
Πίνακας 7: Μέσος όρος και τυπική απόκλιση των φίλτρων ανα στρώμα

	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6
μ	126.7	133.7	147	154.4	133.1	134.3	115.8
σ	29.2	19.05	25	20.6	27.5	29	23.7

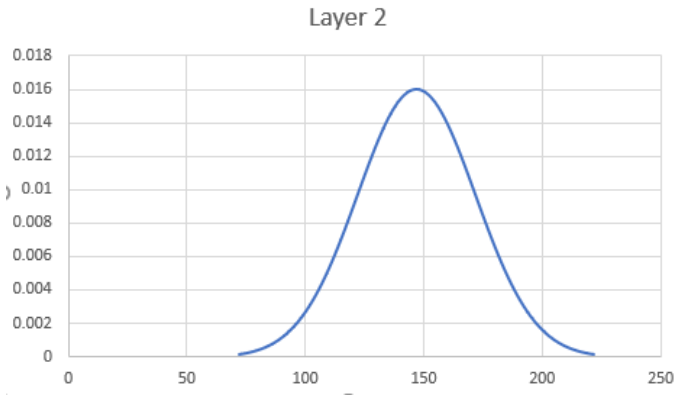
Τα παρακάτω σχήματα απεικονίζουν την κανονική κατανομή που ακολουθούν τα βάρη των φιλτρων ανα στρώμα:



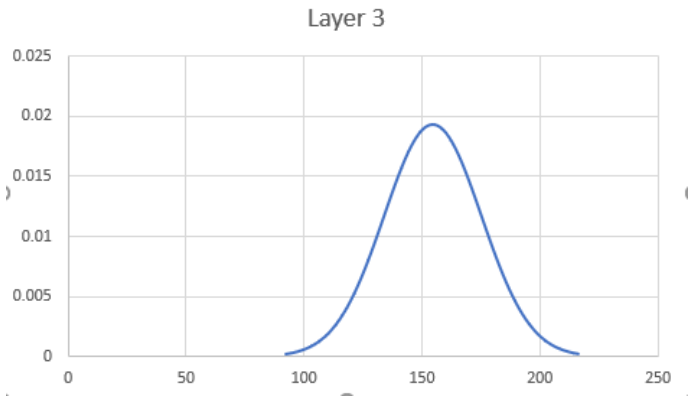
Σχήμα 13: Κατανομή των βαρών των φίλτρων στο πρώτο στρώμα



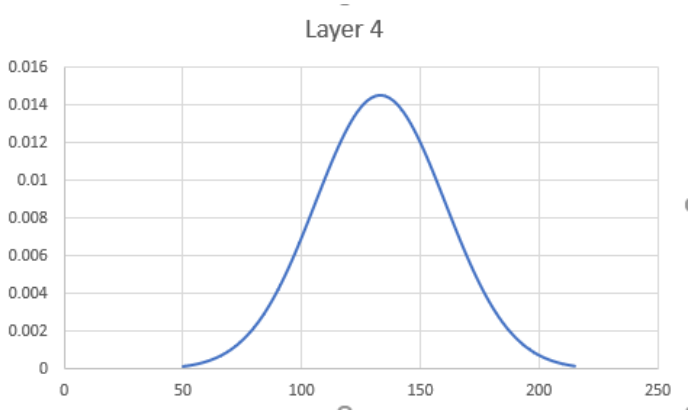
Σχήμα 14: Κατανομή των βαρών των φίλτρων στο δεύτερο στρώμα



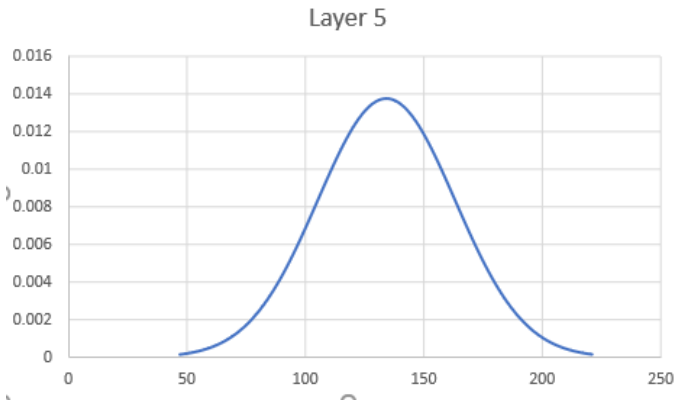
Σχήμα 15: Κατανομή των βαρών των φίλτρων στο τρίτο στρώμα



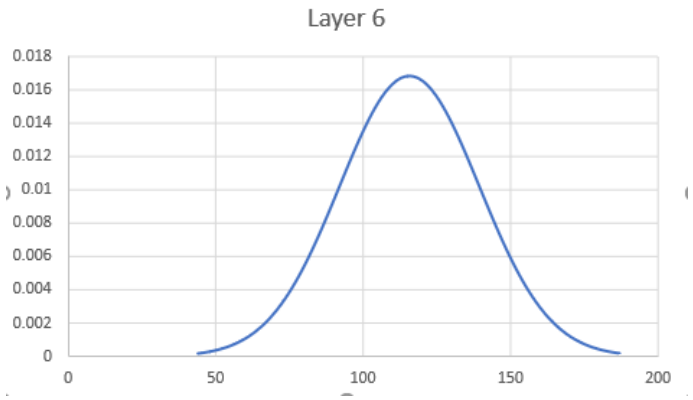
Σχήμα 16: Κατανομή των βαρών των φίλτρων στο τέταρτο στρώμα



Σχήμα 17: Κατανομή των βαρών των φίλτρων στο πέμπτο στρώμα

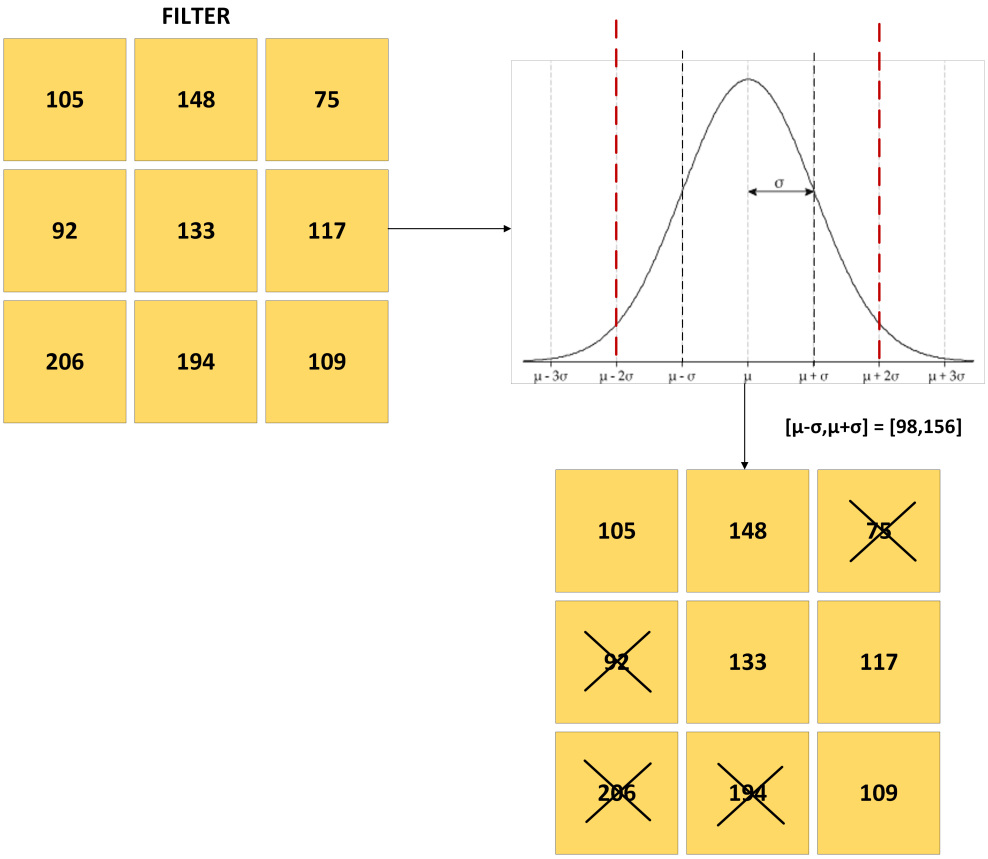


Σχήμα 18: Κατανομή των βαρών των φίλτρων στο έκτο στρώμα



Σχήμα 19: Κατανομή των βαρών των φίλτρων στο έβδομο στρώμα

Ουσιαστικά σε αυτήν την τεχνική προτείνουμε να εκτελούμε μόνο τους πολλαπλασιασμούς με τιμές των φίλτρων που βρίσκονται είτε στο διάστημα $[\mu-\sigma, \mu+\sigma]$ ή στο δίαστμα $[\mu-2\sigma, \mu+2\sigma]$. Αυτό σημαίνει ότι όλοι οι άλλοι πολλαπλασιασμοί με βάρη του φίλτρου που δεν ανήκουν σε ένα από τα δύο παραπάνω διαστήματα δεν θα εκτελούνται, δηλαδή θα παραλείπονται.



Σχήμα 20: Η τέταρτη προσεγγιστική τεχνική

Πειραματική Αξιολόγηση

Ολα τα πειράματα εκτέλεστηκαν στο Tensorflow1.14. Η βιβλιοθήκη ανοιχτού κώδικα στο [1] χρησιμοποιήθηκε προκειμένου να μετρήσουμε την ακρίβεια των πειραμάτων μας. Επιπλέον η ενέργεια των προσεγγιστικών πολλαπλασιαστών μας μετρήθηκε χρησιμοποιώντας το SynopsysDC σε τεχνολογια των 45nm [11]. Οι παράμετροι που δίνονται στην είσοδο είναι:

- Το γράφημα του Resnet-8
- Το Evaluation dataset του Cifar-10 που περιλαμβάνει 10000 εικόνες
- Τέσσερις διαφορετικούς πολλαπλασιαστές απο το [11] με διαφορετικό perforation
- Το filter parameter (k) μόνο για την δεύτερη προσεγγιστική τεχνική

Οι παράμετροι αξιολόγησης είναι:

- Η ακρίβεια πρόβλεψης του μοντέλου που προκύπτει απο κάθε προσεγγιστική τεχνική χρησιμοποιώντας διαφορετικούς συνδυασμούς πολλπλασιαστών με διαφορετικό perforation , καθώς και το Σφάλμα που υπολογίζεται ως Σφάλμα= 1- Ακρίβεια

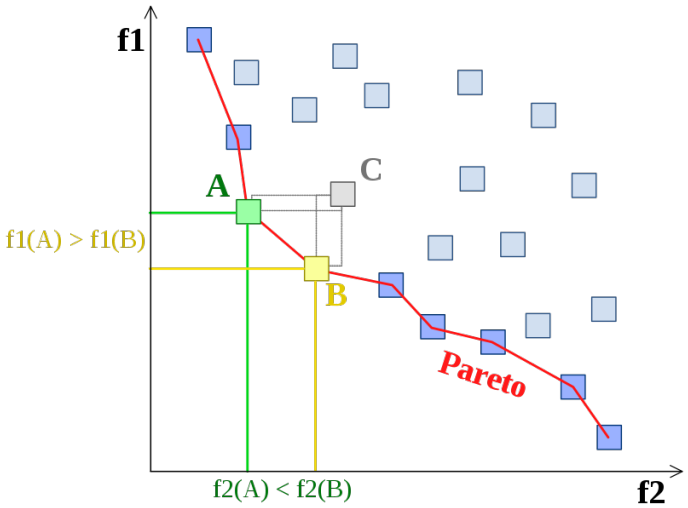
- Ενέργεια για το inference μιας εικόνας , που υπολογίζεται ως το γινόμενο την ενέργειας του ενός προσεγγιστικού πολλαπλασιαστή επί τον αριθμό των πολλαπλασιασμών που συμβαίνουν με αυτόν τον πολλαπλασιαστή κατα τη διάρκεια του inference.

- Throughput δηλαδη τα frames per second.

Για κάθε προσεγγιστική τεχνική δίνεται το Pareto Frontier καθώς και ενα βέλτιστο υποσύνολο των αποτελεσμάτων που ανήκουν στο Pareto Frontier

Ένα σημείο ανήκει στον Pareto Frontier αν δεν γίνεται strictly dominated από κάποιο άλλο σημείο.

Για παράδειγμα στο παρακάτω σχήμα τα κουτάκια αποτελούν κάποια σημεία και προτιμάμε τις μικρότερες τιμές απο τις μεγαλύτερες. Δηλαδή θέλουμε να έχει και μικρότερη τετμημένη (χ) αλλα και μικρότερη τεταγμένη(ψ). Το σημείο C δεν ανήκει στο Pareto Frontier διότι γίνεται strictly dominated και απο το σημείο A και από το σημείο B. Τα σημεία A και B δεν γίνονται strictly dominated απο κάποιο άλλο σημείο και για αυτό το λόγο ανήκουν στο Pareto Frontier.



Σχήμα 21: Παράδειγμα του Pareto Frontier

Θεμελιώδεις Μετρήσεις

Οι προσεγγιστικοί πολλαπλασιαστές που χρησιμοποιούμε στα πειράματα μας είναι απο το [11]. Ειναι πολλαπλασιαστές με διαφορετικό perforation ($p = 0$,$p = 1$,$p = 2$,$p = 3$)

Το perforationμε $p = k$ διαγράφει k συνεχόμενα μερικά γινόμενα ξεκινώντας απο τα αυτα με τη μικρότερη αξία. Δηλαδή ο προσεγγιστικός πολλαπλασιαστής με $p = 1$ θα έχει το τελευταίο μερικό γινόμενο του διεγραμμένο.

Ο παρακάτω πίνακας παρουσιάζει την ακρίβεια όταν καθένας απο τους παραπάνω προσεγγιστικούς πολλαπλασιαστές χρησιμοποείται μόνος αυτός στο δίκτυο:

Πίνακας 8: Ακρίβεια των πολλαπλασιαστών $p = 0, p = 1, p = 2, p = 3$

Multiplier	Inference Accuracy
$p = 0$	0.833
$p = 1$	0.815
$p = 2$	0.78
$p = 3$	0.193

Ο παρακάτω πίνακας παρουσιάζει την ενέργεια των πολλαπλασιαστών αυτών ως component σε τεχνολογία 65nm και 45nm

Πίνακας 9: Ενέργεια του κάθε πολλαπλασιαστή ($\mu W \cdot ns$)

	65nm, 16 bit-width	45nm, 16 bit-width
$p = 0$	3748.5	385.725
$p = 1$	2880	296.355
$p = 2$	2472.48	254.421
$p = 3$	2341.68	240.961

Τώρα θέλουμε να μετρήσουμε την συνολική ενέργεια που απαιτείται για το inference μια εικόνας σύμφωνα με τα όσα έχουμε πει παραπάνω. Ο συνολικός αριθμός πολλαπλασιασμών που εκτελούνται για το inference μιας εικόνας είναι **12238848**.

Πίνακας 10: Συνολική ενέργεια που απαιτείται για μια εικόνα (nJ)

	Energy
$p = 0$	4720.7
$p = 1$	3626.9
$p = 2$	3113.8
$p = 3$	2949.1

Πειραματική αξιολόγηση της πρώτης τεχνικής

Οι παρακάτω πίνακες απεικονίζουν την ακρίβεια πρόβλεψης, την ενέργεια για το inference μίας εικόνας καθώς και το throughput σε τρία διαφορετικά σενάρια:

- Για όλους τους πιθανούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 1$
- Για όλους τους πιθανούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 2$

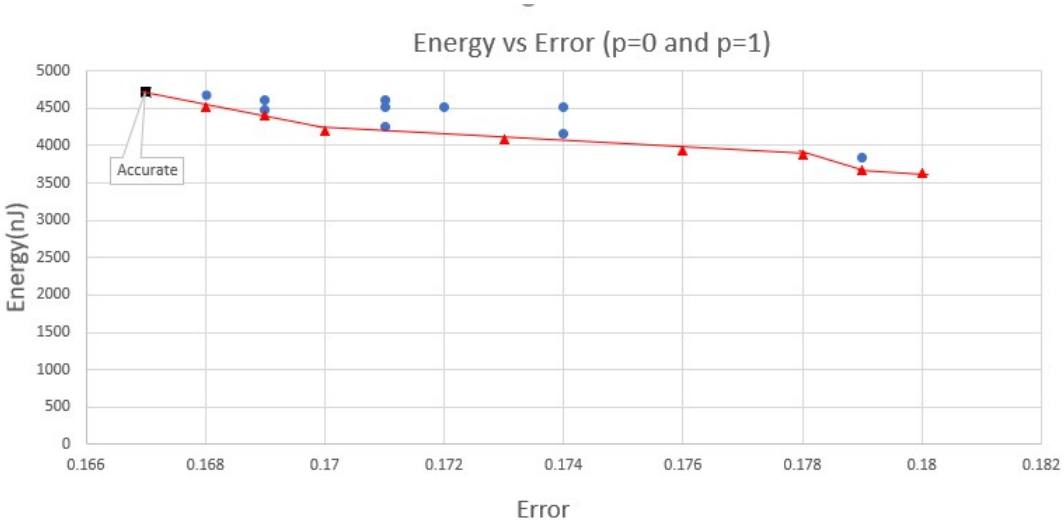
- Για όλους τους πιθανούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 3$

Πίνακας 11: $P = 0$ και $P = 1$

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Inference Accuracy	Energy(nJ)	Throughput(FPS)	Error
$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	0.833	4720.7	18.34	0.167
$p=1$	$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	0.832	4681.2	18.55	0.168
$p=0$	$p=1$	$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	0.826	4509.9	19.04	0.174
$p=0$	$p=0$	$p=1$	$p=0$	$p=0$	$p=0$	$p=0$	0.829	4509.9	19.08	0.171
$p=0$	$p=0$	$p=0$	$p=1$	$p=0$	$p=0$	$p=0$	0.831	4615.3	19.12	0.169
$p=0$	$p=0$	$p=0$	$p=0$	$p=1$	$p=0$	$p=0$	0.828	4509.9	19.04	0.172
$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	$p=1$	$p=0$	0.829	4615.3	19.08	0.171
$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	$p=1$	0.832	4509.9	19.12	0.168
$p=1$	$p=1$	$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	0.831	4470.3	19.19	0.169
$p=1$	$p=1$	$p=1$	$p=0$	$p=0$	$p=0$	$p=0$	0.829	4259.5	19.45	0.171
$p=1$	$p=1$	$p=1$	$p=1$	$p=0$	$p=0$	$p=0$	0.826	4154.1	19.76	0.174
$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	$p=0$	$p=0$	0.824	3943.2	20	0.176
$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	$p=0$	0.821	3837.8	20.16	0.179
$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	0.82	3626.9	20.32	0.18
$p=0$	$p=0$	$p=0$	$p=0$	$p=0$	$p=1$	$p=1$	0.831	4404.4	19.26	0.169
$p=0$	$p=0$	$p=0$	$p=0$	$p=1$	$p=1$	$p=1$	0.83	4193.6	19.49	0.17
$p=0$	$p=0$	$p=0$	$p=1$	$p=1$	$p=1$	$p=1$	0.827	4088.2	19.72	0.173
$p=0$	$p=0$	$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	0.822	3877.3	20	0.178
$p=0$	$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	$p=1$	0.821	3666.5	20.24	0.179

Όπως βλέπουμε απο τον παραπάνω πίνακα, δεν υπάρχουν σημαντικές διαφορές στην ακρίβεια πρόβλεψης για αυτούς τους συνδυασμούς οταν συγκρίνουμε με το **0.833** που ειναι η ακρίβεια όταν χρησιμοποιείται ο ακριβής πολλαπλασιαστής. Η πτώση στην ακρίβεια κυμαίνεται μεταξυ $0,1\% - 1,3\%$. Επιπλέον δεν έχουμε κάποιο σημαντικό κέρδος ως προς την κατανάλωση ενέργειας καθώς και ως προς το throughput .

Το παρακάτω scatter plot απεικονίζει την σχέση μεταξύ ενέργειας και σφάλματος:



Σχήμα 22: Ενέργεια-Σφάλμα όταν έχω $p = 0$ και $p = 1$

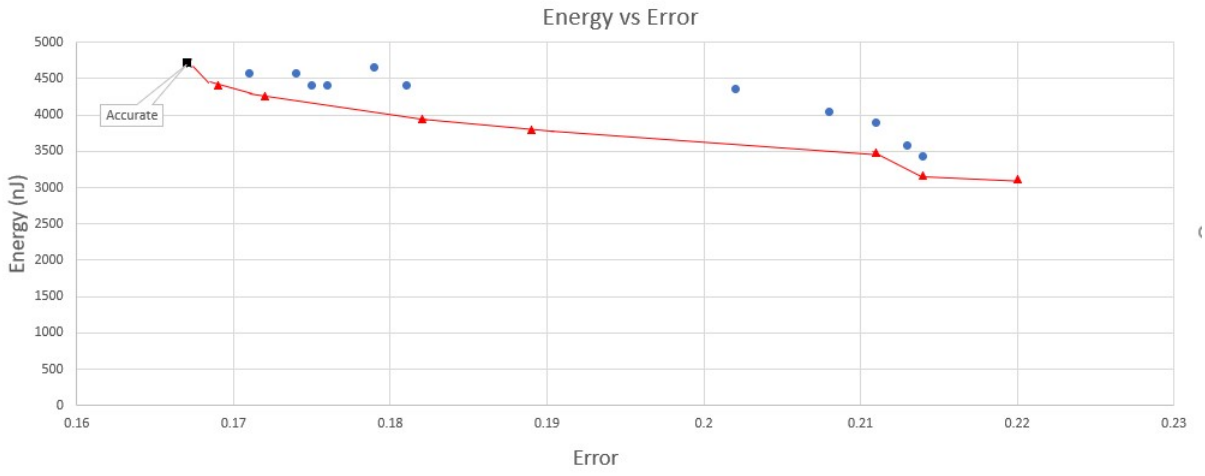
Η κόκκινη γραμμή είναι το **Pareto Frontier** και όλα τα σημεία που ανήκουν σε αυτή τη γραμμή αποτελούν τα βέλτιστα σημεία. Αυτά τα σημεία τονίζονται με χρώμα στον παραπάνω πίνακα. Παρατηρούμε ότι τα περισσότερα σημεία που ανήκουν στο Pareto Frontier είναι όταν προσεγγίζουμε τα τελευταία στρώματα. Αυτό σημαίνει ότι ειναι προτιμότερο να χρησιμοποιούμε τον προσεγγιστικό πολλαπλασιαστή $p = 1$ στα τελευταία στρώματα , δηλαδή ξεκινώντας απο το στρώμα 6 και πηγαίνοντας προς τα πίσω προς το στρώμα 0. Όπως φαίνεται από το παραπάνω σχήμα όταν χρησιμοποιούμε μόνο τον ακριβή πολλαπλασιαστή $p = 0$ το σημείο που παίρνουμε έχει $Error = 0.167, Energy = 4720.7nJ$. Όταν συγκρίνουμε τα υπόλοιπα σημεία που ανήκουν στο **Pareto Frontier** με αυτά τα αποτελέσματα βλέπουμε ότι έχουμε μικρή πτώση στην ακρίβεια αλλα δεν έχουμε κάποιο σημαντικό κέρδος στη ενέργεια.

Πίνακας 12: $P = 0$ και $P = 2$

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Inference Accuracy	Energy(nJ)	Throughput(FPS)	Error
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.833	4720.768451	18.34	0.167
$p = 2$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.821	4662.685532	18.79	0.179
$p = 0$	$p = 2$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.819	4410.992886	19.23	0.181
$p = 0$	$p = 0$	$p = 2$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.824	4410.992886	19.19	0.176
$p = 0$	$p = 0$	$p = 0$	$p = 2$	$p = 0$	$p = 0$	$p = 0$	0.826	4565.880668	19.15	0.174
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 2$	$p = 0$	$p = 0$	0.825	4410.992886	19.26	0.175
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 2$	$p = 0$	0.829	4565.880668	19.19	0.171
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 2$	0.831	4410.9	19.30	0.169
$p = 2$	$p = 2$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.798	4352.9	19.34	0.202
$p = 2$	$p = 2$	$p = 2$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.792	4043.1	19.64	0.208
$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 0$	$p = 0$	$p = 0$	0.789	3888.2	20	0.211
$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 0$	$p = 0$	0.787	3578.4	20.28	0.213
$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 0$	0.786	3423.5	20.45	0.214
$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	0.78	3113.8	20.66	0.22
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 2$	$p = 2$	0.828	4256.1	19.41	0.172
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 2$	$p = 2$	$p = 2$	0.818	3946.3	19.72	0.182
$p = 0$	$p = 0$	$p = 0$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	0.811	3791.4	19.96	0.189
$p = 0$	$p = 0$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	0.789	3481.6	20.36	0.211
$p = 0$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	$p = 2$	0.786	3171.8	20.53	0.214

Όπως βλέπουμε απο τον παραπάνω πίνακα, δεν υπάρχουν σημαντικές διαφορές στην ακρίβεια πρόβλεψης για αυτούς τους συνδυασμούς οταν συγκρίνουμε με το **0.833** που ειναι η ακρίβεια όταν χρησιμοποιείται ο ακριβής πολλαπλασιαστής. Η πτώση στην ακρίβεια κυμαίνεται μεταξυ $0,2\% - 5,3\%$, που σημαίνει ότι έχουμε μεγαλύτερη πτώση στην ακρίβεια τώρα από ότι πριν.

Το παρακάτω scatter plot απεικονίζει την σχέση μεταξύ ενέργειας και σφάλματος:



Σχήμα 23: Ενέργεια-Σφάλμα όταν έχω $p = 0$ και $p = 2$

Η κόκκινη γραμμή είναι το **Pareto Frontier** και όλα τα σημεία που ανήκουν σε αυτή

τη γραμμή αποτελούν τα βέλτιστα σημεία. Αυτά τα σημεία τονίζονται με χρώμα στον παραπάνω πίνακα. Παρατηρούμε ότι τα περισσότερα σημεία που ανήκουν στο Pareto Frontier είναι όταν προσεγγίζουμε τα τελευταία στρώματα. Αυτό σημαίνει ότι ειναι προτιμότερο να χρησιμοποιούμε τον προσεγγιστικό πολλαπλασιαστή $p = 2$ στα τελευταία στρώματα , δηλαδή ξεκινώντας απο το στρώμα 6 και πηγαίνοντας προς τα πίσω προς το στρώμα 0. Όπως φαίνεται από το παραπάνω σχήμα όταν χρησιμοποιούμε μόνο τον ακριβή πολλαπλασιαστή $p = 0$ το σημείο που παίρνουμε έχει $Error = 0.167, Energy = 4720.7nJ$. Όταν συγκρίνουμε τα υπόλοιπα σημεία που ανήκουν στο **Pareto Frontier** με αυτά τα αποτελέσματα βλέπουμε ότι έχουμε πτώση στην ακρίβεια που είναι μεγαλύτερη από πριν. Παρόλλα αυτά παρατηρούμε ότι έχουμε πιο σημαντικά κέρδη στην κατανάλωση ενέργειας τώρα. Για παράδειγμα ο συνδυασμός $p = 0, p = 0, p = 0, p = 2, p = 2, p = 2, p = 2$ σε κάθε στρώμα αντίστοιχα δίνει ακρίβεια **0.811** και συνολική ενέργεια **3791.4** nJ που αν το συγκρίνουμε με τα αποτελέσματα που παίρνουμε απο τον ακριβή πολλαπλασιαστή μας δίνει πτώση στην ακρίβεια κατα $2, 2\%$ και έχουμε μείωση στην κατανάλωση ενέργειας κατα 19.7% .

Πίνακας 13: $P = 0$ και $P = 3$

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Inference Accuracy	Energy(nJ)	Throughput(FPS)	Error
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.833	4720.7	18.34	0.167
$p = 3$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.619	4656.7	18.93	0.381
$p = 0$	$p = 3$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.599	4379.2	19.34	0.401
$p = 0$	$p = 0$	$p = 3$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.741	4379.2	19.34	0.259
$p = 0$	$p = 0$	$p = 0$	$p = 3$	$p = 0$	$p = 0$	$p = 0$	0.756	4550.1	19.30	0.244
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 3$	$p = 0$	$p = 0$	0.756	4379.2	19.41	0.244
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 3$	$p = 0$	0.7759	4550.1	19.37	0.2241
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 3$	0.76	4379.2	19.45	0.24
$p = 3$	$p = 3$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.31	4315.1	19.49	0.69
$p = 3$	$p = 3$	$p = 3$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	0.255	3973.6	19.92	0.745
$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 0$	$p = 0$	$p = 0$	0.231	3802.9	20.24	0.769
$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 0$	$p = 0$	0.204	3461.3	20.61	0.796
$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 0$	0.196	3290.6	20.79	0.804
$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	0.193	2949.0	21.05	0.807
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 3$	$p = 3$	0.68	4208.4	19.56	0.32
$p = 0$	$p = 0$	$p = 0$	$p = 0$	$p = 3$	$p = 3$	$p = 3$	0.585	3866.9	20	0.415
$p = 0$	$p = 0$	$p = 0$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	0.503	3696.1	20.32	0.497
$p = 0$	$p = 0$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	0.435	3354.6	20.61	0.565
$p = 0$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	$p = 3$	0.306	3013.1	20.83	0.694

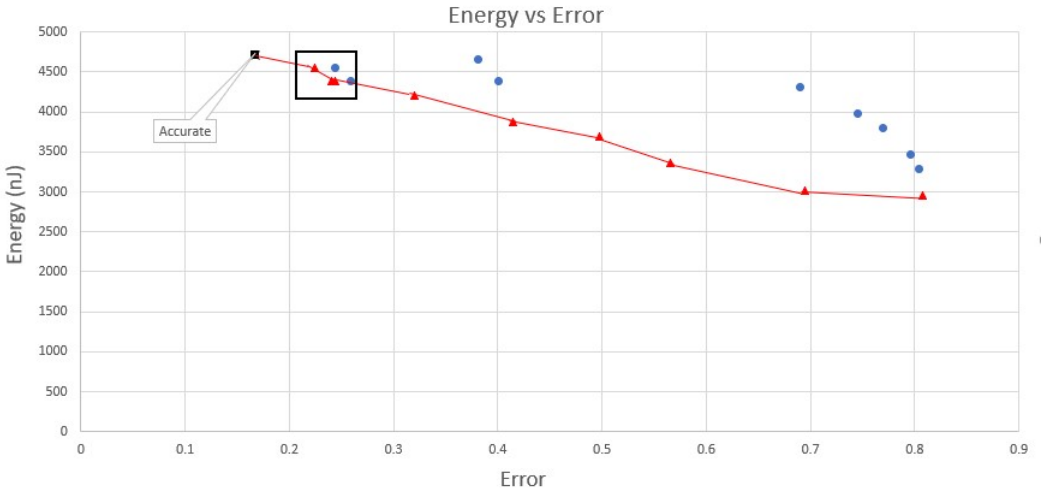
Όπως βλέπουμε απο τον παραπάνω πίνακα έχουμε πλέον σημαντικές διαφορές στην ακρίβεια συγκριτικά με το 0.833 του ακριβή πολλαπλασιαστή. Η πτώση στην ακρίβεια τώρα κυμαίνεται μεταξύ $5, 7\% - 64\%$. Αυτό ειναι αναμενόμενο καθώς ο $p = 3$ έχει πολυ μεγάλο σφάλμα.

Αυτό που όμως ειναι πολύ σημαντικό είναι το γεγονός ότι χρησιμοποιώντας τον $p = 3$ στα τελευταία στρώματα παίρνουμε πολύ καλύτερη ακρίβεια από ότι οταν τον χρησιμοποιούμε

στα πρώτα στρώματα. Για παράδειγμα όταν τον χρησιμοποιούμε στο τελευταία δύο στρώματα η ακρίβεια είναι 0.68% ενώ οταν τον χρησιμοποιούμε στα δύο πρώτα η ακρίβεια είναι 0.31%.

Καταλήγουμε άρα στο συμπέρασμα ότι γενικά είναι προτιμότερο όταν χρησιμοποιούμε αυτή την τεχνική να χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές στα τελευταία στρώματα αντί για τα πρώτα.

Το παρακάτω scatter plot απεικονίζει την σχέση μεταξύ ενέργειας και σφάλματος:



Σχήμα 24: Ενέργεια-Σφάλμα όταν έχω $p = 0$ και $p = 3$

Συγκρίνοντας τα σημεία που ανήκουν το **Pareto Frontier** με το σημείο που προκύπτει απο τον ακριβή πολλαπλασιαστή βλέπουμε ότι η πτώση στην ακρίβεια ειναι τεράστια (έως και 64%) Μπορεί να υπάρχουν σημαντικά κέρδη στην ενέργεια αλλα η πτώση στην ακρίβεια ειναι τοσο μεγάλη που δεν μπορεί να γίνει ανεκτή.

Πειραματική αξιολόγηση δεύτερης τεχνικής

- Για $k = 3$ **parts**

Το scatter plot παρακάτω παρουσιάζει την σχέση ενέργειας και σφάλματος όταν χωρίζουμε τον αριθμό των φίλτρων σε κάθε στρώμα σε 3 ισοδύναμα κομμάτια και για 6 διαφορετικά σενάρια:

1. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 1$

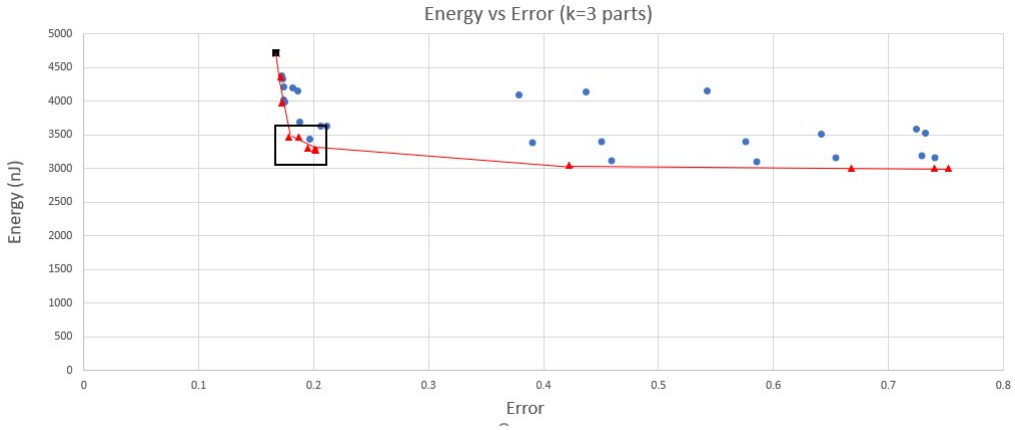
2. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 2$

3. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 3$

4. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 1$ και $p = 2$

5. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 1$ και $p = 3$

6. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 2$ και $p = 3$



Σχήμα 25: Δεύτερη προσεγγιστική τεχνική: Ενέργεια-Σφάλμα για $k = 3$

Η κόκκινη γραμμή είναι το **Pareto Frontier** και όλα τα σημεία που ανήκουν σε αυτό ειναι τα βέλτιστα κατά **Pareto** σημεία. Οι λύσεις αυτές παρουσιάζονται στον παρακάτω πίνακα. Το μάυρο τετράγωνο περιλαμβάνει ενα υποσύνολο των βέλτιστων αυτών σημείων που εμείς θεωρούμε ως τα πιο βέλτιστα σε αυτή την τεχνική με βάση δύο περιορισμούς. Θέλουμε το σφάλμα να είναι μικρότερο απο 22%, και η συνολική ενέργεια να είναι τουλάχιστον κατα 1000. nJ μικρότερη από αυτή που καταναλώνεται όταν χρησιμοποιούμε τον ακριβή πολλαπλασιαστή.

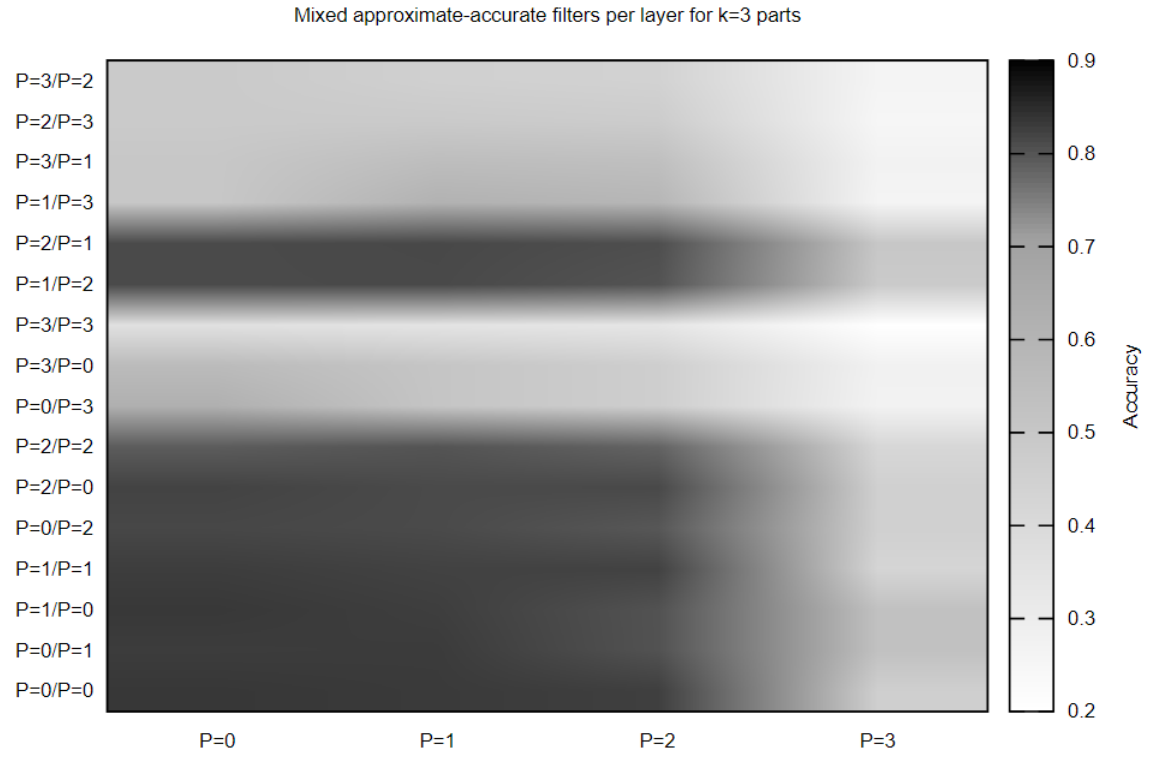
Τα σημεία αυτά είναι χρωματισμένα στον παρακάτω πίνακα:

Πίνακας 14: Παρέτο σημεία για $k = 3$

1^{st} *part*	2^{nd} *part*	3^{rd} *part*	Energy(nJ)	Throughput(FPS)	Error
$p = 0$	$p = 0$	$p = 0$	4720.7	18.34	0,167
$p = 0$	$p = 1$	$p = 0$	4364.1	19.49	0.171
$p = 0$	$p = 1$	$p = 1$	3973.7	19.76	0.172
$p = 2$	$p = 1$	$p = 1$	3464.2	20.61	0.178
$p = 1$	$p = 2$	$p = 1$	3459.6	20.40	0.187
$p = 2$	$p = 2$	$p = 1$	3296.9	21.09	0.195
$p = 2$	$p = 1$	$p = 2$	3281.1	21.05	0.201
$p = 2$	$p = 2$	$p = 3$	3055.0	21.59	0.422
$p = 2$	$p = 3$	$p = 3$	3001.2	21.36	0.668
$p = 3$	$p = 3$	$p = 2$	3007.8	21.83	0.74
$p = 3$	$p = 2$	$p = 3$	3002.7	21.69	0.752

Παρατηρούμε ότι το υποσόνολο αυτό των βέλτιστων σημείων προκύπτει όταν χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές $p = 1$ ανδ $p = 2$. Είναι πολυ καλές λύσεις , καθώς συγκρίνοντας τις λύσεις αυτές με τον $p = 0$, η μείωση στην ακρίβεια ειναι πολυ μικρή καθώς κυμαίνεται μεταξύ $1,1\% - 3,4\%$ ενώ έχουμε μείωση στην κατανάλωση ενέργειας έως και 30.5%, αποτέλεσμα πολύ καλό δεδομένου ότι η μείωση στην ακρίβεια είναι τόσο μικρή.

Το heatmap παρακάτω απεικονίζει πως αλλάζει η ακρίβεια χρησιμοποιώντας σε αυτή την τεχνική διαφορετικούς προσεγγιστικους πολλαπλασιαστές.



Σχήμα 26: Αλλαγές στην ακρίβεια για $k = 3$

Από το παραπάνω heatmap παρατηρούμε ότι η ακρίβεια αρχίζει να πέφτει όταν αρχίζουμε να χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές. Συγκεκριμένα παρατηρούμε ότι οι συνδυασμοί με τους πολλαπλασιαστές $p = 0$, $p = 1$ ανδ $p = 2$ δεν έχει μεγάλο αντίκτυπο στην ακρίβεια καθώς οι περιοχές αυτές παραμένουν σκουρόχρωμες. Παρόλλα αυτά όταν αρχίζουμε να χρησιμοποιούμε τον προσεγγιστικό πολλαπλασιαστή $p = 3$ βλέπουμε ξεκάθαρα ότι η ακρίβεια αρχίζει να πέφτει σημαντικά και αυτό φαίνεται απο τις πιο άσπρες περιοχές.

Πειραματική αξιολόγηση τρίτης τεχνικής

Προσεγγίσεις ανά φίλτρο χρησιμοποιώντας διαφορετικούς προσεγγιστικους πολλαπλασιαστές

- **Επίπεδο βάθους**

Το παρακάτω scatter plot δείχνει την σχέση μεταξύ ενέργειας και σφάλματος για έξι διαφορετικά σενάρια στο επίπεδο του βάθος:

1. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 1$

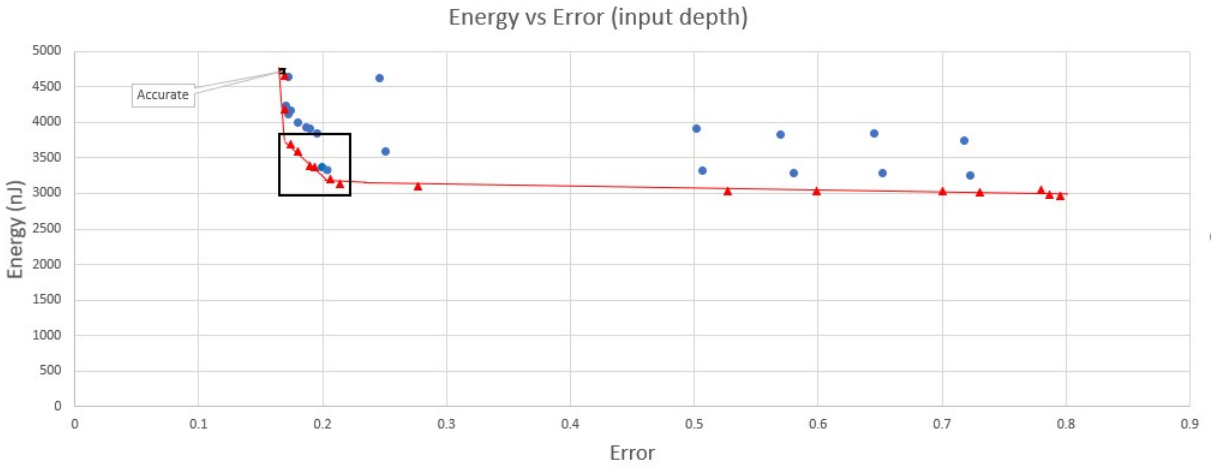
2. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 2$

3. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 0$ και $p = 3$

4. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 1$ και $p = 2$

5. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 1$ και $p = 3$

6. Για όλους τους διαφορετικούς συνδυασμούς προσεγγιστικών πολλαπλασιαστών $p = 2$ και $p = 3$



Σχήμα 27: Τρίτη προσεγγιστική τεχνικής: Ενέργεια-Σφάλμα στο επίπεδο βάθους

Η κόκκινη γραμμή είναι το **Pareto Frontier** και όλα τα σημεία που ανήκουν σε αυτό ειναι τα βέλτιστα κατά **Pareto** σημεία. Οι λύσεις αυτές παρουσιάζονται στον παρακάτω πίνακα. Το μάυρο τετράγωνο περιλαμβάνει ενα υποσύνολο των βέλτιστων αυτών σημείων που εμείς θεωρούμε ως τα πιο βέλτιστα σε αυτή την τεχνική με βάση δύο περιορισμούς. Θέλουμε το σφάλμα να είναι μικρότερο απο 22%, και η συνολική ενέργεια να είναι τουλάχιστον κατα 1000. nJ μικρότερη από αυτή που καταναλώνεται όταν χρησιμοποιούμε τον ακριβή πολλαπλασιαστή.

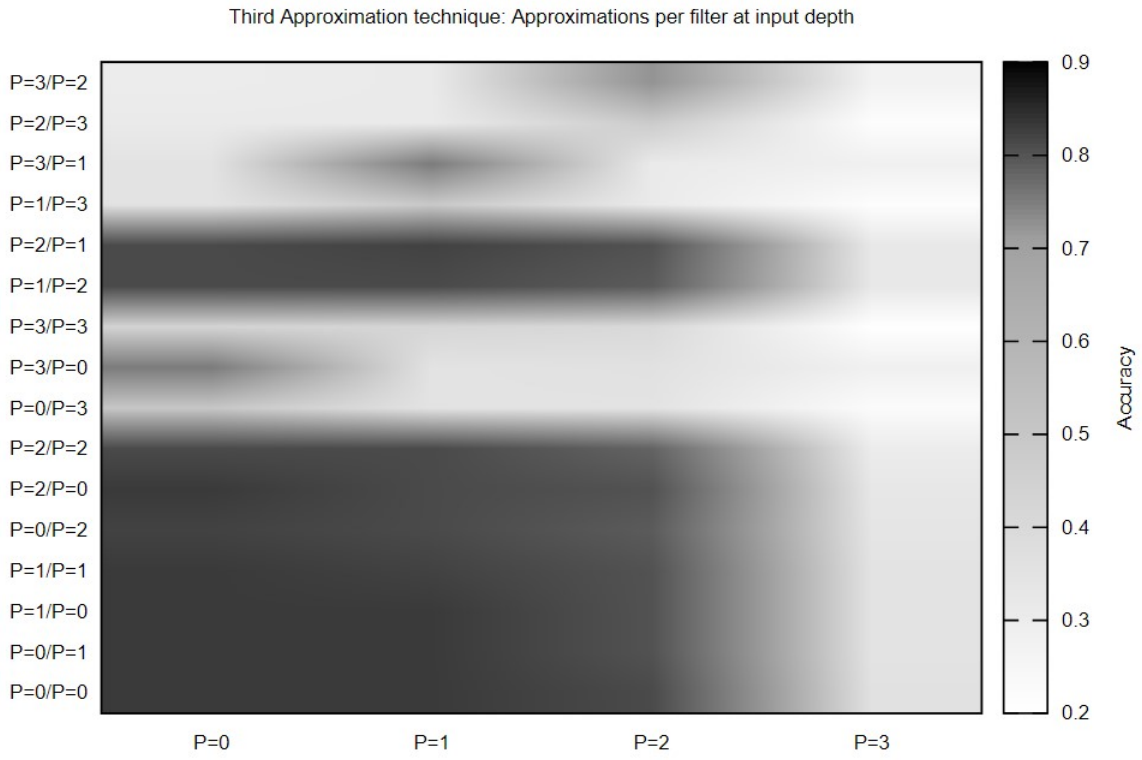
Τα σημεία αυτά είναι χρωματισμένα στον παρακάτω πίνακα:

Πίνακας 15: Παρέτο σημεία και το βέλτιστο υποσύνολο τους στο επίπεδου βάθους

$0-1$	$1-2$	$2-3$	Energy(nJ)	Throughput(FPS)	Error
$p=0$	$p=0$	$p=0$	4720.7	18.34	0,167
$p=0$	$p=1$	$p=0$	4661.4	19.76	0.168
$p=1$	$p=0$	$p=0$	4180.4	19.88	0.169
$p=1$	$p=0$	$p=1$	3686.2	19.96	0.174
$p=1$	$p=2$	$p=1$	3599.1	20.24	0.18
$p=1$	$p=1$	$p=2$	3395.1	20.24	0.19
$p=2$	$p=1$	$p=1$	3373.4	20.16	0.199
$p=2$	$p=0$	$p=2$	3200.9	20.20	0.206
$p=2$	$p=1$	$p=2$	3141.6	20.36	0.214
$p=2$	$p=3$	$p=2$	3104.8	20.79	0.277
$p=2$	$p=2$	$p=3$	3039.3	20.74	0.527
$p=2$	$p=3$	$p=3$	3030.4	21.05	0.599
$p=3$	$p=2$	$p=2$	3032.4	20.70	0.7
$p=3$	$p=3$	$p=2$	3023.5	21.05	0.73
$p=3$	$p=0$	$p=3$	3045.1	20.70	0.78
$p=3$	$p=1$	$p=3$	2985.8	20.83	0.787
$p=3$	$p=2$	$p=3$	2958.0	21.14	0.795

Παρατηρούμε ότι το υποσόνολο αυτό των βέλτιστων σημείων προκύπτει όταν χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές $p=1$ ανδ $p=2$. Είναι πολυ καλές λύσεις , καθώς συγκρίνοντας τις λύσεις αυτές με τον $p=0$, η μείωση στην ακρίβεια ειναι πολυ μικρή καθώς κυμαίνεται μεταξύ $0.7\% - 4.7\%$ ενώ έχουμε μείωση στην κατανάλωση ενέργειας έως και 33.5%, αποτέλεσμα πολύ καλό δεδομένου ότι η μείωση στην ακρίβεια είναι τόσο μικρή.

Το heatmap παρακάτω απεικονίζει πως αλλάζει η ακρίβεια χρησιμοποιώντας σε αυτή την τεχνική διαφορετικούς προσεγγιστικους πολλαπλασιαστές.

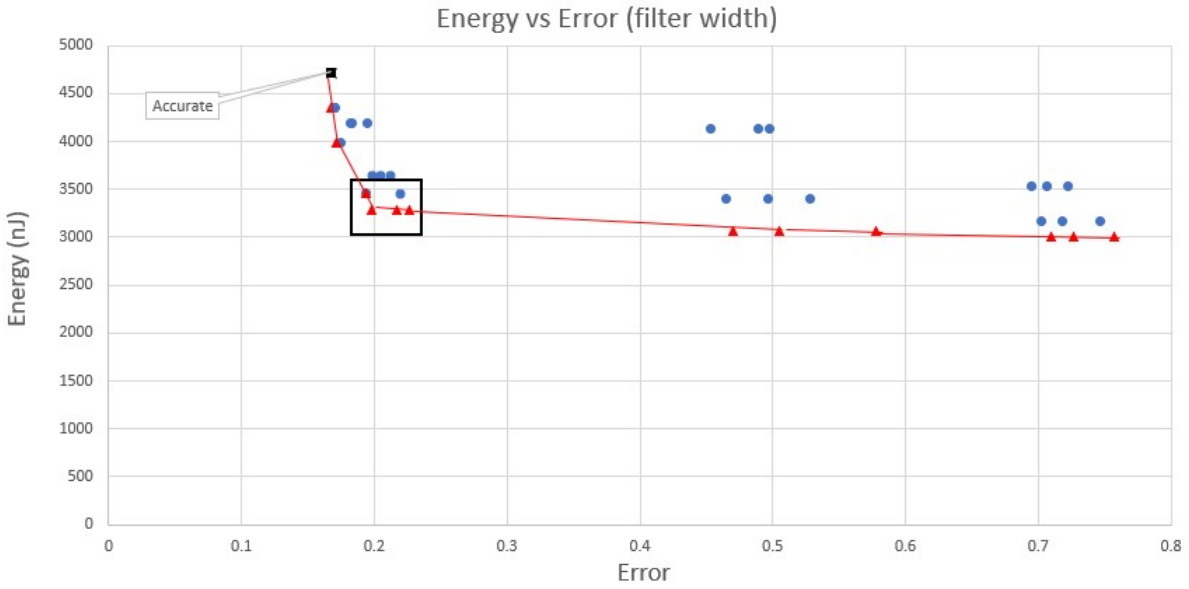


Σχήμα 28: Τρίτη προσεγγιστική Τεχνικής: Αλλαγές στην ακρίβεια στο επίπεδου βάθους

Από το παραπάνω heatmap παρατηρούμε ότι η ακρίβεια αρχίζει να πέφτει όταν αρχίζουμε να χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές. Συγκεκριμένα παρατηρούμε ότι οι συνδυασμοί με τους πολλαπλασιαστές $p = 0$, $p = 1$ ανδ $p = 2$ δεν έχει μεγάλο αντίκτυπο στην ακρίβεια καθώς οι περιοχές αυτές παραμένουν σκουρόχρωμες. Παρόλλα αυτά όταν αρχίζουμε να χρησιμοποιούμε τον προσεγγιστικό πολλαπλασιαστή $p = 3$ βλέπουμε ξεκάθαρα ότι η ακρίβεια αρχίζει να πέφτει σημαντικά και αυτό φαίνεται απο τις πιο άσπρες περιοχές.

- **Επίπεδο πλάτους**

Το παρακάτω scatter plot δείχνει την σχέση μεταξύ ενέργειας και σφάλματος για έξι διαφορετικά σενάρια στο επίπεδο του πλάτους:



Σχήμα 29: Τρίτη προσεγγιστκή τεχνικής: Ενέργεια-Σφάλμα στο επίπεδο πλάτους

Η κόκκινη γραμμή είναι το **Pareto Frontier** και όλα τα σημεία που ανήκουν σε αυτό ειναι τα βέλτιστα κατά **Pareto** σημεία. Οι λύσεις αυτές παρουσιάζονται στον παρακάτω πίνακα. Το μάυρο τετράγωνο περιλαμβάνει ενα υποσύνολο των βέλτιστων αυτών σημείων που εμείς θεωρούμε ως τα πιο βέλτιστα σε αυτή την τεχνική με βάση δύο περιορισμούς. Θέλουμε το σφάλμα να είναι μικρότερο απο 23%, και η συνολική ενέργεια να είναι τουλάχιστον κατα 1000 nJ μικρότερη από αυτή που καταναλώνεται όταν χρησιμοποιούμε τον ακριβή πολλαπλασιαστή.

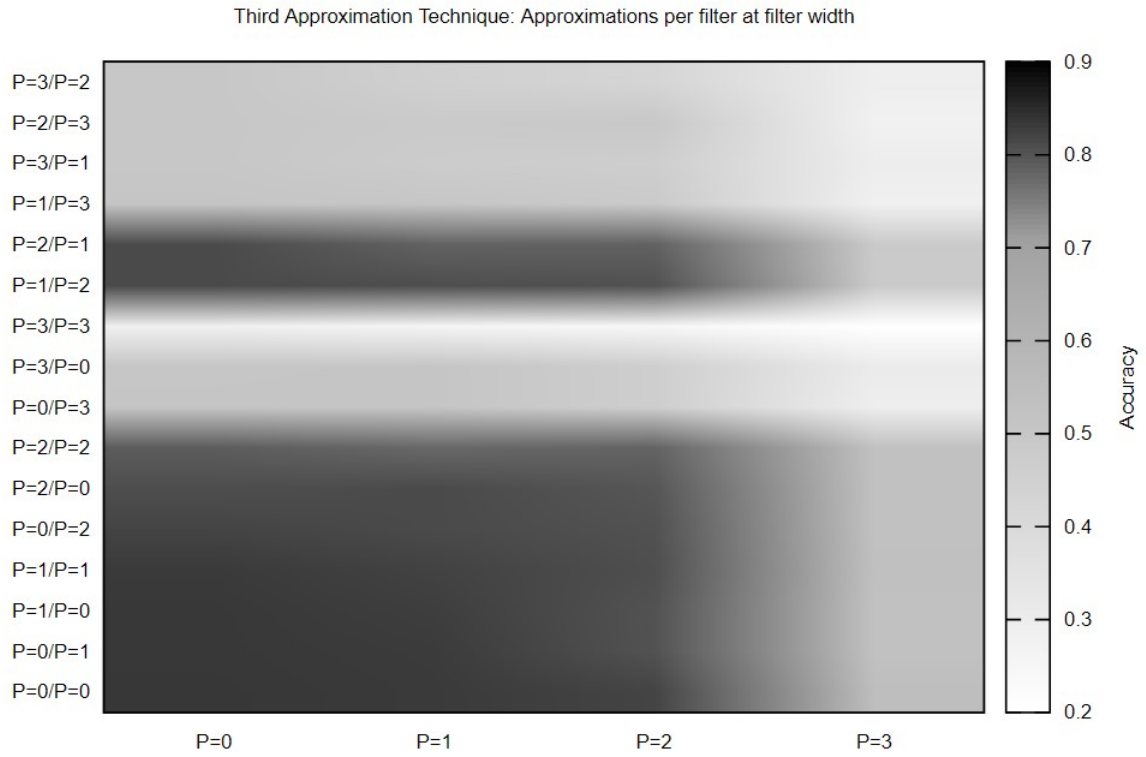
Τα σημεία αυτά είναι χρωματισμένα στον παρακάτω πίνακα:

Πίνακας 16: Παρέτο σημεία και το βέλτιστο υποσύνολο τους στο επίπεδου πλάτους

$0-1$	$1-2$	$2-3$	Energy(nJ)	Throughput(FPS)	Error
$p=0$	$p=0$	$p=0$	4720.768451	18.34	0,167
$p=0$	$p=1$	$p=0$	4356.1	19.8019802	0.168
$p=1$	$p=0$	$p=1$	3991.5	19.96007984	0.172
$p=1$	$p=1$	$p=2$	3455.9	20.28397566	0.194
$p=2$	$p=1$	$p=2$	3284.8	20.32520325	0.198
$p=2$	$p=2$	$p=1$	3284.8	20.40816327	0.217
$p=1$	$p=2$	$p=2$	3284.8	20.36659878	0.226
$p=3$	$p=2$	$p=2$	3058.8	20.70393375	0.47
$p=2$	$p=2$	$p=3$	3058.8	20.83333333	0.505
$p=2$	$p=3$	$p=2$	3058.8	20.74688797	0.578
$p=3$	$p=3$	$p=2$	3003.9	20.96436059	0.71
$p=3$	$p=2$	$p=3$	3003.9	21.09704641	0.726
$p=2$	$p=3$	$p=3$	3003.9	20.96436059	0.757

Παρατηρούμε ότι το υποσόνολο αυτό των βέλτιστων σημείων προκύπτει όταν χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές $p=1$ ανδ $p=2$. Είναι πολυ καλές λύσεις , καθώς συγκρίνοντας τις λύσεις αυτές με τον $p=0$, η μείωση στην ακρίβεια ειναι πολυ μικρή καθώς κυμαίνεται μεταξύ $2.7\% - 5.9\%$ ενώ έχουμε μείωση στην κατανάλωση ενέργειας έως και 30.4%, αποτέλεσμα πολύ καλό δεδομένου ότι η μείωση στην ακρίβεια είναι τόσο μικρή.

Το heatmap παρακάτω απεικονίζει πως αλλάζει η ακρίβεια χρησιμοποιώντας σε αυτή την τεχνική διαφορετικούς προσεγγιστικους πολλαπλασιαστές.

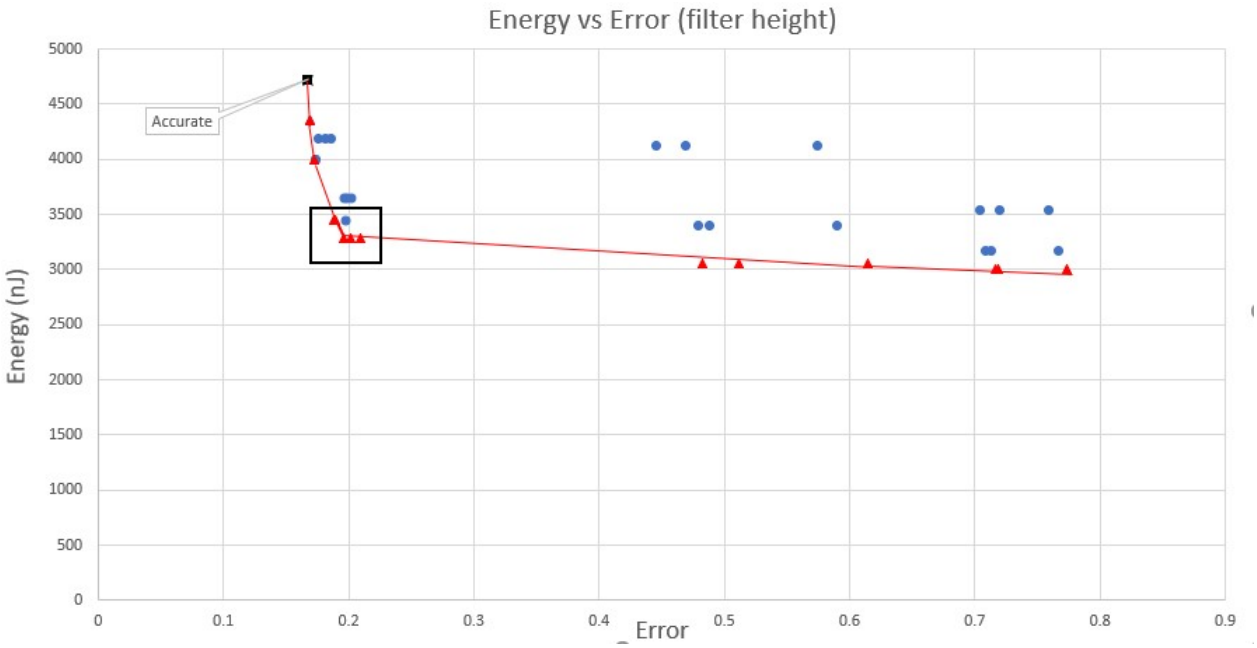


Σχήμα 30: Τρίτη προσεγγιστική Τεχνική: Αλλαγές στην ακρίβεια στο επίπεδου πλάτους

Από το παραπάνω heatmap παρατηρούμε ότι η ακρίβεια αρχίζει να πέφτει όταν αρχίζουμε να χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές. Συγκεκριμένα παρατηρούμε ότι οι συνδυασμοί με τους πολλαπλασιαστές $p = 0$, $p = 1$ ανδ $p = 2$ δεν έχει μεγάλο αντίκτυπο στην ακρίβεια καθώς οι περιοχές αυτές παραμένουν σκουρόχρωμες. Παρόλλα αυτά όταν αρχίζουμε να χρησιμοποιούμε τον προσεγγιστικό πολλαπλασιαστή $p = 3$ βλέπουμε ξεκάθαρα ότι η ακρίβεια αρχίζει να πέφτει σημαντικά και αυτό φαίνεται απο τις πιο άσπρες περιοχές.

- **Επίπεδο ύψους**

Το παρακάτω scatter plot δείχνει την σχέση μεταξύ ενέργειας και σφάλματος για έξι διαφορετικά σενάρια στο επίπεδο του ύψους:



Σχήμα 31: Τρίτη προσεγγιστκή τεχνικής: Ενέργεια-Σφάλμα στο επίπεδο ύψους

Η κόκκινη γραμμή είναι το **Pareto Frontier** και όλα τα σημεία που ανήκουν σε αυτό ειναι τα βέλτιστα κατά **Pareto** σημεία. Οι λύσεις αυτές παρουσιάζονται στον παρακάτω πίνακα. Το μάυρο τετράγωνο περιλαμβάνει ενα υποσύνολο των βέλτιστων αυτών σημείων που εμείς θεωρούμε ως τα πιο βέλτιστα σε αυτή την τεχνική με βάση δύο περιορισμούς. Θέλουμε το σφάλμα να είναι μικρότερο απο 22%, και η συνολική ενέργεια να είναι τουλάχιστον κατα 1000 nJ μικρότερη από αυτή που καταναλώνεται όταν χρησιμοποιούμε τον ακριβή πολλαπλασιαστή.

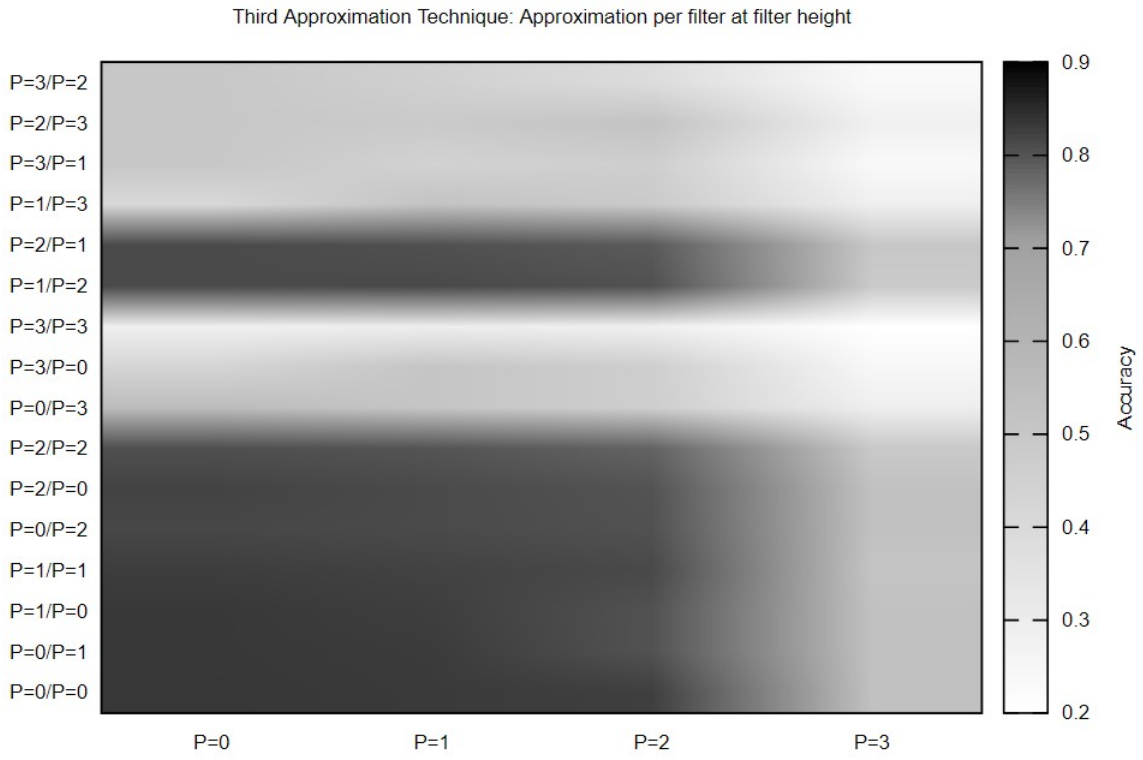
Τα σημεία αυτά είναι χρωματισμένα στον παρακάτω πίνακα:

Πίνακας 17: Τρίτη προσεγγιστκή τεχνικής: Ενέργεια-Σφάλμα στο επίπεδο ύψους

$0-1$	$1-2$	$2-3$	Energy(nJ)	Throughput(FPS)	Error
$p=0$	$p=0$	$p=0$	4720.7	18.34	0,167
$p=1$	$p=0$	$p=0$	4356.1	19.80	0.169
$p=1$	$p=0$	$p=1$	3991.5	19.92	0.173
$p=2$	$p=1$	$p=1$	3455.9	20.16	0.189
$p=2$	$p=1$	$p=2$	3284.8	20.40	0.196
$p=1$	$p=2$	$p=2$	3284.8	20.32	0.202
$p=2$	$p=2$	$p=1$	3284.8	20.36	0.209
$p=2$	$p=2$	$p=3$	3058.8	20.70	0.482
$p=3$	$p=2$	$p=2$	3058.8	20.70	0.512
$p=2$	$p=3$	$p=2$	3058.8	20.74	0.615
$p=2$	$p=3$	$p=3$	3003.9	21.09	0.716
$p=3$	$p=2$	$p=3$	3003.9	21.09	0.719
$p=3$	$p=3$	$p=2$	3003.9	21.18	0.773

Παρατηρούμε ότι το υποσόνολο αυτό των βέλτιστων σημείων προκύπτει όταν χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές $p=1$ ανδ $p=2$. Είναι πολυ καλές λύσεις , καθώς συγκρίνοντας τις λύσεις αυτές με τον $p=0$, η μείωση στην ακρίβεια ειναι πολυ μικρή καθώς κυμαίνεται μεταξύ $2.2\%-4.2\%$ ενώ έχουμε μείωση στην κατανάλωση ενέργειας έως και 30.4%, αποτέλεσμα πολύ καλό δεδομένου ότι η μείωση στην ακρίβεια είναι τόσο μικρή.

Το heatmap παρακάτω απεικονίζει πως αλλάζει η ακρίβεια χρησιμοποιώντας σε αυτή την τεχνική διαφορετικούς προσεγγιστικους πολλαπλασιαστές.



Σχήμα 32: Τρίτη προσεγγιστική Τεχνικής: Αλλαγές στην ακρίβεια στο επίπεδου ύψους

Από το παραπάνω heatmap παρατηρούμε ότι η ακρίβεια αρχίζει να πέφτει όταν αρχίζουμε να χρησιμοποιούμε τους προσεγγιστικούς πολλαπλασιαστές. Συγκεκριμένα παρατηρούμε ότι οι συνδυασμοί με τους πολλαπλασιαστές $p = 0$, $p = 1$ ανδ $p = 2$ δεν έχει μεγάλο αντίκτυπο στην ακρίβεια καθώς οι περιοχές αυτές παραμένουν σκουρόχρωμες. Παρόλλα αυτά όταν αρχίζουμε να χρησιμοποιούμε τον προσεγγιστικό πολλαπλασιαστή $p = 3$ βλέπουμε ξεκάθαρα ότι η ακρίβεια αρχίζει να πέφτει σημαντικά και αυτό φαίνεται απο τις πιο άσπρες περιοχές.

Προσεγγίσεις ανά φίλτρο παραλείποντας πράξεις

- **Επίπεδο βάθους**

Η ακρίβεια αυτή προκύπτει όταν εκτελούμε μόνο τις πράξεις που ικανοποιούν την συνθήκη της πρώτης στήλης του παρακάτω πίνακα:

Πίνακας 18: Ακρίβεια όταν παραλείπουμε πράξεις στο επίπεδο βάθους

	Number of multiplications	Inference Accuracy	Energy(nJ)	FPS(Throughput)
Input Depth = 0	6045696	0.115	2331.9	52.63
Input Depth = 1	663552	0.105	255.9	62.5
Input Depth = 2	5529600	0.104	2132.8	54.05
Input Depth = 0 *and* 1	6709248	0.12	2587.8	50
Input Depth = 1 *and* 2	6193152	0.138	2388.8	52.63
Input Depth = 0 *and* 2	11575296	0.507	4464.8	22.72

- **Επίπεδο Πλάτους**

Η ακρίβεια αυτή προκύπτει όταν εκτελούμε μόνο τις πράξεις που ικανοποιούν την συνθήκη της πρώτης στήλης του παρακάτω πίνακα:

Πίνακας 19: Ακρίβεια όταν παραλείπουμε πράξεις στο επίπεδο πλάτους

	Number of multiplications	Inference Accuracy	Energy(nJ)	FPS(Throughput)
Filter Width = 0	4079616	0.11	1573.5	74.62
Filter Width = 1	4079616	0.151	1573.5	81.30
Filter Width = 2	4079616	0.082	1573.5	73.52
Filter Width = 0 *and* 1	8159232	0.227	3147.1	42.55
Filter Width = 1 *and* 2	8159232	0.163	3147.1	41.15
Filter Width = 0 *and* 2	8159232	0.205	3147.1	39.84

- **Επίπεδο Ύψους**

Η ακρίβεια αυτή προκύπτει όταν εκτελούμε μόνο τις πράξεις που ικανοποιούν την συνθήκη της πρώτης στήλης του παρακάτω πίνακα:

Πίνακας 20: Ακρίβεια όταν παραλείπουμε πράξεις στο επίπεδο ύψους

	Number of multiplications	Inference Accuracy	Energy(nJ)	FPS(Throughput)
Filter Height = 0	4079616	0.092	1573.5	74.07
Filter Height = 1	4079616	0.162	1573.5	54.64
Filter Height = 2	4079616	0.098	1573.5	56.81
Filter Height = 0 *and* 1	8159232	0.219	3147.1	33.89
Filter Height = 1 *and* 2	8159232	0.178	3147.1	32.46
Filter Height = 0 *and* 2	8159232	0.127	3147.1	33.00

Από τους τρεις παραπάνω πίνακες ότι η πτώση στην ακρίβεια συγκριτικά με την ακρίβεια

που παίρνουμε όταν χρησιμοποιούμε μονο τον ακριβή πολλαπλαστή χωρις να παραλείπουμε καμία πράξη, είναι τεράστια ενω όπως ειναι αναμεενόμενο έχουμε μεγάλη πτώση στην κατανάλωση ενέργειας.

Παρόλλα αυτά η τεράστια πτώση στην ακρίβεια καθιστά την τεχνική αυτή μη αποτελεσματική

Πειραματική αξιολόγηση της τέταρτης τεχνικής

Ο παρακάτω πίνακας απεικονίζει τα διαστήματα$[\mu - \sigma, \mu + \sigma]$, $[\mu - 2\sigma, \mu + 2\sigma]$ για κάθε στρώμα του νευρωνικού δικτύου ξεχωριστά:

Πίνακας 21: Διαστήματα για κάθε στρώμα του νευρωνικού δικτύου

	Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
$[\mu - \sigma, \mu + \sigma]$	[98,156]	[114,153]	[122,172]	[133,175]	[105,161]	[105,163]	[92,140]
$[\mu - 2\sigma, \mu + 2\sigma]$	[69,185]	[95,172]	[97,197]	[112,196]	[77,189]	[76,192]	[68,164]

Πίνακας 22: Αριθμός πολλαπλασιασμών σε κάθε στρώμα για αυτά τα διαστήματα

	Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	**Άθροισμα**
$[\mu - \sigma, \mu + \sigma]$	339968	1264640	1703936	829952	1702400	792320	1617472	**8250688**
$[\mu - 2\sigma, \mu + 2\sigma]$	418816	1783808	2238464	1118720	2241792	1107584	2228928	**11138112**

1. Εκτελώντας μόνο τις πράξεις με τιμές φίλτρου που ανήκουν στο διάστημα $[\mu - \sigma, \mu + \sigma]$ του κάθε στρώματος και παραλείποντας δηλαδή τις υπόλοιπες έχεις ως αποτέλεσμα να παίρνουμε ακρίβεια ίση με $\mathbf{0.553}$ και συνολική ενέργεια για μια εικόνα ίση με $\mathbf{3182.4}$ nJ, που σημαίνει ότι εχουμε μείωση στην ενέργεια κατά 32.6% συγκριτικά με τον ακριβή πολλαπλασιαστή. Παρόλλα αυτά η πτώση στην ακρίβεια είναι πολύ μεγάλη και δεν μπορεί να γίνει δεκτή.

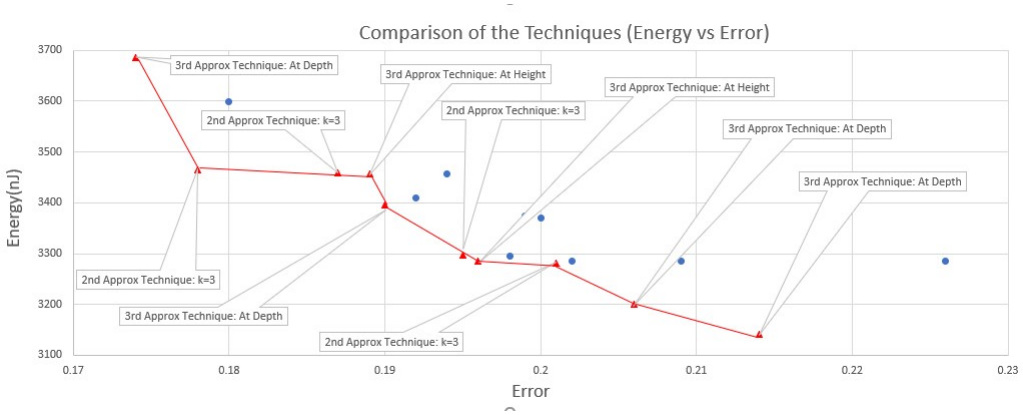
2. Εκτελώντας μόνο τις πράξεις με τιμές φίλτρου που ανήκουν στο διάστημα $[\mu - 2\sigma, \mu + 2\sigma]$ του κάθε στρώματος και παραλείποντας δηλαδή τις υπόλοιπες έχεις ως αποτέλεσμα να παίρνουμε ακρίβεια ίση με $\mathbf{0.686}$ και συνολική ενέργεια για μια εικόνα ίση με $\mathbf{4296.1}nJ$

Σύγκριση των προσεγγιστικών τεχνικών

Στόχος μας εδώ ειναι να βρούμε ποιες απο τις παραπάνω τεχνικές είναι οι πιο αποτελεσματικές δηλαδή δίνουν τα βέλτιστα αποτελέσματα ως προς την ενέργεια και την ακρίβεια.

Για τον σκοπό αυτό θα συγκρίνουμε το υποσύνολο των βέλτιστων λύσεων που προκύπτουν από κάθε τεχνική.

Το παρακάτω σχήμα απεικονίζει όλα τα βέλιστα υποσύνολα σημείων που προέκυψαν από κάθε τεχνική:



Σχήμα 33: Σύγκριση προσεγγιστικών τεχνικών

Η κόκκινη γραμμή είναι το Pareto Frontier και όλα τα σημεία που ανήκουν σε αυτή ειναι τα βέλιστα κατά Pareto σημεία. Αυτά τα βέλτιστα σημεία παρουσιάζονται στον παρακάτω πίνακα:

Πίνακας 23: Παρέτο βέλτιστα σημεία από τη σύγκριση των προσεγγιστικών τεχνικών

Τεχνική	$0-1$	$1-2$	$2-3$	Energy(nJ)	Error	Energy Saving %
3^{rd}: *Depth*	$p=1$	$p=0$	$p=1$	3686.2	0.174	21.9%
2^{nd}: $k=3$	$p=2$	$p=1$	$p=1$	3464.2	0.178	26.62%
2^{nd}: $k=3$	$p=1$	$p=2$	$p=1$	3459.6	0.187	26.71%
3^{rd}: *Height*	$p=2$	$p=1$	$p=1$	3455.9	0.189	26.79%
3^{rd}: *Depth*	$p=1$	$p=1$	$p=2$	3395.1	0.19	28.08%
2^{nd}:$k=3$	$p=2$	$p=2$	$p=1$	3296.9	0.195	30.16%
3^{rd}: *Height*	$p=2$	$p=1$	$p=2$	3284.8	0.196	30.41%
2^{nd}: $k=3$	$p=2$	$p=1$	$p=2$	3281.1	0.201	30.5%
3^{rd}: *Depth*	$p=2$	$p=0$	$p=2$	3200.9	0.206	32.2%
3^{rd}: *Depth*	$p=2$	$p=1$	$p=2$	3141.6	0.214	33.45%

Μπορούμε εύκολα να παρατηρήσουμε ότι τα περισσότερα βέλτιστα κατά Pareto σημεία προκύπτουν όταν εφαρμόζουμε την τρίτη προσεγγιστική τεχνική, αντικαθιστώντας τους πολλαπλασιασμούς με διαφορετικούς προσεγγιστικούς πολλαπλασιαστές και κυρίως όταν εφαρμόζουμε αυτή την τεχνική στο επίπεδο του βάθους. Τα υπόλοιπα βέλτιστα σημεία προκύπτουν από την δεύτερη προσεγγιστική τεχνική όταν χωριζουμε τον αριθμό των φίλτρων σε κάθε στρώμα σε 3 κομμάτια .

Συνεπώς οι δύο αυτές τεχνικές είναι οι καλύτερες αφού ειναι οι καλύτερες προσεγγιστικές τεχνικές αφού παρέχουν τα καλύτερα trade-off μεταξύ σφάλματος και ενέργειας.

Σύγκριση με State-of-the-art

Θα συγκρίνουμε την ακρίβεια που προκύπτει όταν κάποιοι απο τους καλύτερους πολλαπλαστές , ως προς το σφάλμα τους απο την EvoApproxLib [3] χρησιμοποιούνται στο νευρωνικό χωρίς καμία απο τις προσεγγιστικές μας τεχνικές με την ακριβεια που προκύπτει όταν οι ίδιοι πολλαπλασιαστές εφαρμόζονται στις καλύτερες προσεγγιστικές τεχνικές που περιγράψαμε παραπάνω.

Επιπλέον θα συγκρίνουμε την ενέργεια που απαιτείται για μία εικόνα όταν οι ίδιοι πολλαπλασιαστές από την EvoApproxLib χρησιμοποιούνται χωρίς καμία τεχνική με την ενέργεια που προκύπτει όταν εφαρμοζονται οι καλύτερες μας τεχνικές χρησιμοποιώντας τους πολλαπλασιαστές απο [11] , δηλαδή πολλαπλασιαστές με διαφορετικό perforation.

Στον παρακάτω πίνακα απεικονίζεται η ακρίβεια, η ενέργεια που απαιτείται για μία εικόνα και το σφάλμα όταν μερικοί από τους καλύτερους πολλαπλασιαστές απο την EvoApproxLib χρησιμοποιούνται στο Resnet-8 χωρίς να εφαρμοζεται καμία απο τις πρσεγγιστικές μας τεχνικές.

Πίνακας 24: Ακρίβεια, Ενέργεια και σφάλμα για μερικούς πολλαπλασιαστές απο την[3]

Multiplier	Inference Accuracy	Energy(nJ)	Error
mul8u 2AC	0.798	5290.7	0.202
mul8u 2HH	0.767	5322.4	0.233
mul8u 2P7	0.829	6708.3	0.171
mul8u 14VP	0.828	6147.8	0.172
mul8u 150Q	0.83	6124.3	0.17
mul8u GS2	0.826	6012.7	0.174
mul8u NGR	0.77	4627.7	0.23
mul8u ZFB	0.767	4204.2	0.233

Οι δικές μας καλύτερες λύσεις που προέκυψαν από την προηγούμενη σύγκριση είναι πολύ καλύτερες συγκριτικά με τις λύσεις που προσφέρουν οι προσεγγιστικοί πολλαπλασιαστές από την EvoApproxLib τόσο ως προς το σφάλμα όσο και ως προς την ενέργεια

Θα συγκρίνουμε την ακρίβεια και την ενέργεια που προκύπτει όταν χρησιμοποιούμε τους πολλαπλαστίες απο την EvoApproxLib χωρίς την εφαρμογή των τεχνικών μας με την ακρίβεια και την ενέργεια που παίρνουμε απο τις βέλτιστες μας λύσεις μέσω των καλύτερων τεχνικών μας.

Για παράδειγμα ο πολλαπλαστής mul8u GS2 όταν χρησιμοποιείται μόνος του χωρίς καμία τεχνική μας δίνει σφάλμα στην ακρίβεια 17.4% και ενέργεια για μία εικόνα ίση με **6012.7** nJ , ενώ όταν χρησιμοποιούμε την τρίτη μας τεχνική στο επίπεδου του βάθους που έχουμε συμπεράνει πιο πάνω ότι ειναι μία απο τις καλύτερες, χρησιμοποιώντας τον εξής συνδυασμό προσεγγιστικών πολλαπλασιαστών $p = 1$, $p = 0$, $p = 1$ έχουμε το ίδιο σφάλμα 17.4% αλλα με ενέργεια ίση με **3686.2** nJ , που ειναι κατά 38.7% λιγότερη.

Επίσης, όταν χρησιμοποιείται ο πολλαπλασιαστής mul8u 2AC όνος του χωρίς καμία τεχνική μας δίνει σφάλμα στην ακρίβεια 20.2% και ενέργεια για μία εικόνα ίση με **5290.7** nJ , ενώ όταν χρησιμοποιούμε την δεύτερη χωρίζοντας τον αριθμό των φίλτρων σε κάθε στρωμα σε $k = 3$ κομμάτια, που και αυτή ειναι μία απο τις καλύτερες τεχνικές μας, χρησιμοποιώντας τον εξής συνδυασμό προσεγγιστικών πολλαπλασιαστών $p = 2$, $p = 1$, $p = 2$ εχόυμε σφάλμα ίσο με 20.1% και ενέργεια ίση με **3281.1** nJ , που είναι κατά 38% λιγότερη. Αυτό σημαίνει ότι όχι μόνο έχουμε μικρότερο σφάλμα που ειναι επιθυμητό αλλά και η κατανάλωση ειναι πολύ λιγότερη.

Αυτά τα σημαντικά κέρδη στην κατανάλωση ενέργειας πετυχαίνοντας περίπου το ίδιο σφάλμα είναι αποτέλεσμα της χρήσης των συγκεκριμένων αυτών πολλαπλασιαστών [11] που ο καθένας έχει διαφορετικό perforation. Προφανώς βέβαια οφείλονται και στην χρήση των δικών μας προσεγγιστικών τεχνικών που επιτρέπουν την ταυτόχρονη χρήση αυτών των πολλαπλασιαστών.

Chapter 1

Introduction

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

Many computationally intensive applications (such as image recognition, video processing and data mining) feature an intrinsic error-resilience property [12]. Since they often process noisy or redundant data and their users are willing to accept certain errors in many cases, the principles of **approximate computing** can be employed in the design of their energy efficient implementations [13]. At the circuit level, approximations (i.e. circuit simplifications) are intentionally introduced to find a good trade-off between power consumption, performance and error. A distinguished class of applications among all these error resilient applications are hardware accelerators of deep neural networks (DNNs) [14]. Due to the increased computation requirements of modern DNN workloads, inference engines are implemented either in FPGA or ASIC technology, which provide increased throughput and/or low power consumption. In this context, several FPGA-based [15], [16] and ASIC-based [17], [18] DNN accelerators have been proposed in the literature. In the case of DNNs, approximate implementations have been proposed at the level of DNN architecture, data representation, arithmetic operations, memory access and memory cells [14], [19], [20]. The approximations can be introduced to the circuit in

various steps of the standard circuit design flow.

Approximate Computing (AC) [13] is an alternative design approach that exploits the inherent error tolerance of algorithms and applications from domains such as machine learning (ML), DSP, numerical analysis, etc, and relaxes the accuracy in the calculations to provide significant gains in power and/or energy consumption, area, latency, etc. It can be applied at different layers of the design abstractions, i.e., starting from the application level and moving to the hardware and VLSI level [21].

1.1 Motivation and Thesis Objectives

This dissertation contributes to the areas of deep neural networks and approximate computing, focusing on the development new approximation techniques. Our motivation is to design and create new approximation techniques using approximate multipliers in Deep Neural Networks that achieve a good trade-off between inference accuracy and energy. More specifically, we desire to extend via multiple new approximation techniques the open-source library [1] that already extends TensorFlow library providing Approximate Convolutional (ApproxConv) layers. Specifically, this library provides the user the chance to approximate one Deep Neural network uniformly (i.e the same approximation everywhere of the neural network) using only one approximate multiplier. Contrary to this library [1] , in our proposed extended library of approximation techniques we offer two new main innovations:

1. Support of **multiple** approximate multipliers at the same time instead of only 1 as proposed in [1].

2. Approximations at different levels , which will be analyzed in chapter 4.

We want to study the impact of using multiple approximate multipliers simultaneously in our design on the inference accuracy- energy trade-off, as well as design and create new approximation techniques using this approximate multipliers at different levels inside the Deep Neural Network and test their efficiency by counting the inference accuracy and the enengy required for the inference of one input image that each approximation technique has as a result using various combinations of approximate multipliers.

What we want to achieve is to give the opportunity to any user to test his/her arbitrary multipliers on the proposed different approximation techniques. Thus with these proposed approximation techniques we offer a new extended library of Tensorflow. Our proposed extended library of approximation techniques is better than the library in [1] , because while this library restricts the users to use only one approximate multiplier, our proposed library provides the users the opportunity to use more than 1 approximate multiplier simultaneously. Furthermore, while the library in [1] provides only one way of approximation (i.e uniform approximation using one approximate multiplier all around the Deep Neural Network), our proposed extended library provides users the opportunity

to test their approximate multipliers at different approximation levels using our proposed various approximation techniques. Overall, with our proposed library users can experiment using various approximate multipliers at different approximation levels and adjust the level of approximation he wants based on the accuracy-energy trade-off he wants to achieve.

To summarize, the current thesis aims at:

- Studying and understanding how the library in [1] works. This is the library on which this thesis is based.
- Extending this library, so it can support multiple approximate multipliers at the same time.
- Extending this library by designing and creating new approximation techniques at different levels inside the Deep Neural Network.
- Analyzing the experimental results so we can test the efficiency of our proposed approximation techniques by measuring the inference accuracy and the energy required for the inference of one input image using various combinations of approximate multipliers.
- Comparing our proposed approximation techniques with each other as well as with the accurate implementation and coming up with the best approximation techniques.
- Comparing our proposed approximation techniques with approximate multipliers from the EvoApproxLib [3] and drawing conclusions about our techniques.

1.2 Thesis Outline

In chapter 2 some theoretical mandatory background will be given in order to better understand the broader meaning of deep neural networks and approximate computing. More specifically, we will dive further into the world of convolutional neural networks and the basic concepts of what approximate computing is.

In chapter 3 the basic library basic library/tool called TensorFlow Approximate Layers [1] that was used in this theses will be described. More specifically, we will describe how this open source library works and we will provide simple examples for its usage.

Following, in chapter 4 we will present and describe the different approximation techniques we propose in this thesis. These proposed approximation techniques are extensions of the open source library already described in the chapter 3. More specifically, a detailed explanation of how each approximation technique works will be given.

Finally, in chapter 5 the experimental results of each approximation technique will be displayed and some comparisons between approximate techniques will be made as well as with some approximate multipliers from EvoApproxlib [3]. More specifically, we will study how each approximation technique influences the classification accuracy as well as

the energy consumed for the inference of one input image. Furthermore, we will provides some scatter plots illustrating the relation between the energy and the error with their Pareto frontier as well and some heatmaps depicting how the inference accuracy changes when different approximate multipliers are employed.

Chapter 6 shows some conclusion drawn from the previous results, as well as some suggestions for future work.

Chapter 2

Theoretical Background and Related Work

This chapter's goal is to present the two main technological advancements that this Thesis is based on: Convolutional Neural Networks (CNNs) and Approximate Computing.

2.1 Convolutional Neural Networks

2.1.1 The bigger picture of Machine Learning

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

Machine Learning is a computational sub-field of Artificial Intelligence. Artificial Intelligence itself poses two main questions: "What is intelligence and how does it work?" and "Can we build intelligent machines?". Correlated with the latter one and as its name suggests, in Machine Learning we try to train computers, in a way that they can learn to solve problems without being explicitly programmed. Using a more formal definition for "learning" in this context : "A computer program is said to learn from experience E, with respect to some task T and some performance measure P, if it's performance on T as measured by P is improved with experience E". At the core of Machine Learning lies the assumption that knowledge can be derived from data. Based on this assumption, the majority of ML algorithms so far are data-driven in contrast to other AI approaches which may be symbolic,knowledge based etc. Machine Learning takes steps towards "intelligent"

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

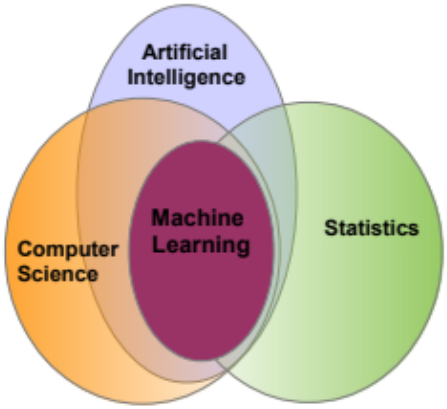


Figure 2.1: Machine Learning's relation to other fields

Types of Machine Learning

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

- ***Supervised Learning***: In Supervised Learning we have the input variables (X) and the output variables and the goal is for the algorithm to learn an approximation of the mapping unction from the input to the output, $Y = F(X)$. It is called "supervised" because there is a form of "teacher", giving feedback to the algorithm, based on the already known correct answers. The most common problems that fall under supervised learning are classification and regression, depending on whether the output is discrete (i.e classes) or continuous, respectively. Both classification and regression problems may have one or more input variables and input variables may be any data type, such as numerical or categorical. Some machine learning algorithms are described as "supervised" machine learning algorithms as they are designed for supervised machine learning problems. Popular examples include: decision trees, support vector machines, and many more.

- ***Unsupervised Learning***: In Unsupervised Learning we are only given the input data (X) and no corresponding output variables. The goal is to model the underlying structure in the data, if of course there is one. As such, unsupervised learning does not have a teacher correcting the model, as in the case of supervised learning. The most common problems in unsupervised learning are those of clustering: We are interested in grouping a set of objects in such a way that objects in the same group, the "cluster", are more similar to each other than to those in other clusters. An example of a clustering algorithm is k-Means where k refers to the number of clusters to discover in the data.

The problems we dealt with in this thesis are classification problems, so supervised learning will be further discussed.

2.1.2 Introduction to Artificial Neural Networks

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

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

A biological as well as a mathematical model of a neuron can be seen in figure 2.2 and 2.3 respectively.

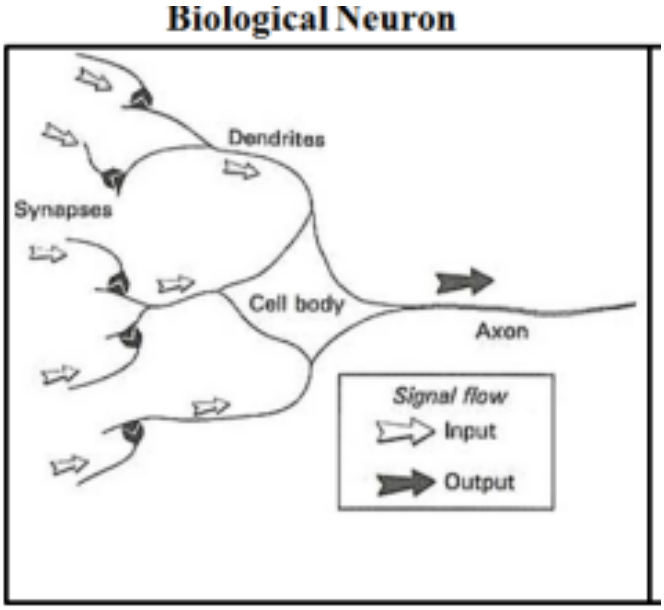


Figure 2.2: Biological Neuron

A single neuron also called a perceptron is the basic building block in Artificial Neural Networks. Neurons are the basic computational units and are consisted by three main parts. The input data expressed in numerical form, the activation function and the output data. A neuron receives a number of input signals x_i multiplied by weights w_i. These

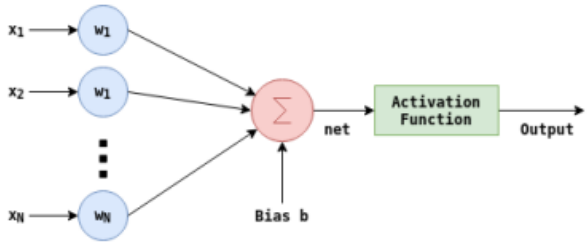


Figure 2.3: Single Neuron Model

weighted input data are summed biased with a fixed value b_i and are fed into the activation function Φ that produce the final output.

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

The equations in figure 2.3 are described by 5.1 and 5.4

$$Net = \sum_{i=1}^{N} x_i * w_i + b \tag{2.1}$$

$$Output = \Phi(Net) = \Phi(\sum_{i=1}^{N} x_i * w_i + b) \tag{2.2}$$

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

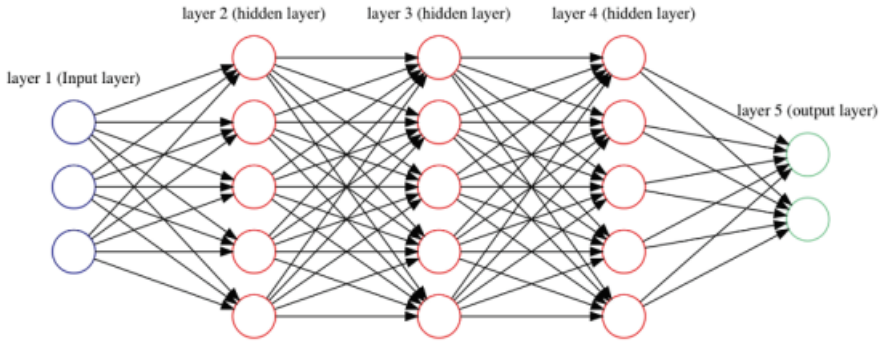


Figure 2.4: Example of a Feed-Forward Neural Network with tree hidden layers, three inputs and two outputs. The weights of the network are illustrated with the arrows.

As it can be clearly seen in figure 2.4, the input is a [1x3] vector and the output a [1x2] vector.All the weights of a layer can be stored in a single matrix. For example, the weights that connect the input to the second layer in figure 3.8 are a matrix of size [3x5]. Then, in the forward pass phase, the values at the second layer can be calculated through multiplication of the input vector by the weight matrix, resulting to a [1x5] vector. By applying the activation function F to each one of the vector's elements, the values at the second layer are calculated. In a similar way the output of each of the next layers is calculated. The full forward pass of this 4-layer neural network is then simply four matrix multiplications, interwoven with the application of the activation function.

The first step in order to construct a neural network is to train the model using the training data. In order to do so, weights are initialized randomly. Considering training, we should take care of the loss function. Loss function shows how good a neural network is on a specific task. The intuitive way to implement it is to take each training example, pass it though the network, get the output value and subtract it from the actual value that was expected in the output.

$$J(y, \hat{y}) = \sum_{i=1}^{N} x_i * w_i + b \tag{2.3}$$

Where i is the index of training example, y stands for the output number we expect from the network and $\hat{y}$ for the number we actually got by passing our example through the network. We want as small number as possible regarding Loss Function. If the Loss Function has a big value, that means our network does not perform very well.

Since initial weights are randomly initialized, we expect a bad performance of the network. In the process of training, we want to start with a bad performing neural network and wind up with a network with high accuracy. In terms of loss function, we should get the minimum value in the end of training. In each iteration, weights are adjusted in order to achieve higher accuracy. The problem of training is equivalent to the problem

2.1.3 Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special kind of neural networks for processing data that has a known grid-like topology. Data can be thought as 1D time-series data and image data that can be thought as a 2D grid of pixels. As the name indicates these networks employ the mathematical operation called convolution. CNNs are very similar to ordinary Neural Networks. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. CNNs are biologically inspired models by research of D.H. Hubel and T.N. Wiesel. They proposed and explanation for the way in which mammals visually perceive the world around then using a layered architecture of neurons in the brain and this in turn inspired engineers to attempt to develop similar pattern recognition mechanisms in computer vision. As an example, let's consider a car. How

does a human recognize that is a car? Humans search for the characteristics that are unique to a car like wheels, head-lights, doors, rear trunk, glass windows, hood and other features that differ it from other models of transport. Similarly when recognizing a wheel, we look for circular-shaped objects, comparatively dark colored with a rough texture positioned below the main structure of the car. All these little details are taken into account to form some basic information. These little information together bunch up to form a particular characteristic that is unique to an object that we are recognizing. "A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activation to another through a differentiable function." What it actually means is that, each layer is associated with converting the information from the values, available in the previous layers, into some more complex information and pass on to the next layers for further generalization.

The CNN is a combination of two basic building blocks:

- **The Convolution Block**:This block consists of the Convolution Layer and theP-ooling Layer. This layer forms the essential component of feature extraction.
- **The Fully Connected Block**: Consists of a fully connected neural network architecture. This layer performs the task of classification based on the input from the convolutional block.

Below two different examples of convolutional neural networks are illustrated. 2.5 illustrates how a convolutional neural network works in general while 2.6 illustrates a detailed convolutional neural network, the VGG-16 network

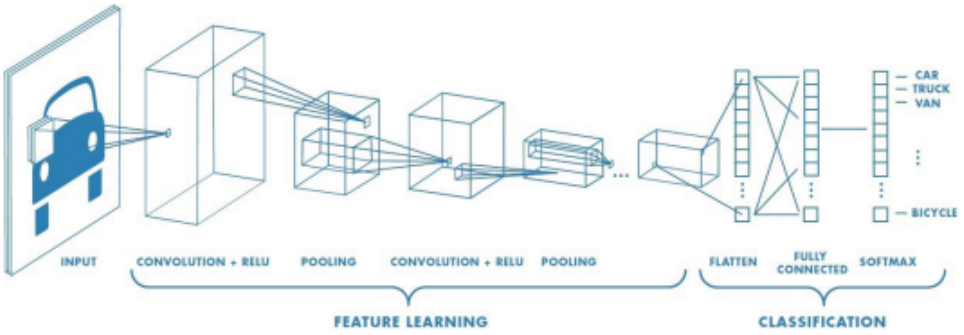


Figure 2.5: Example of a Convolutional Neural Network that categorizes means of transport from an input image

Filters or Kernels or Convolutional Matrices are also an image that depict a particular feature for example a curve or a dot. Convolution is a special operation applied on a particular matrix (usually the Image matrix), using another matrix (usually the Filter matrix). Kernel is simply a small matrix of weights. This kernel slides over the 2D input data, performing an element wise multiplication with the part of the input it is currently on and then summing up the results into a single output pixel.

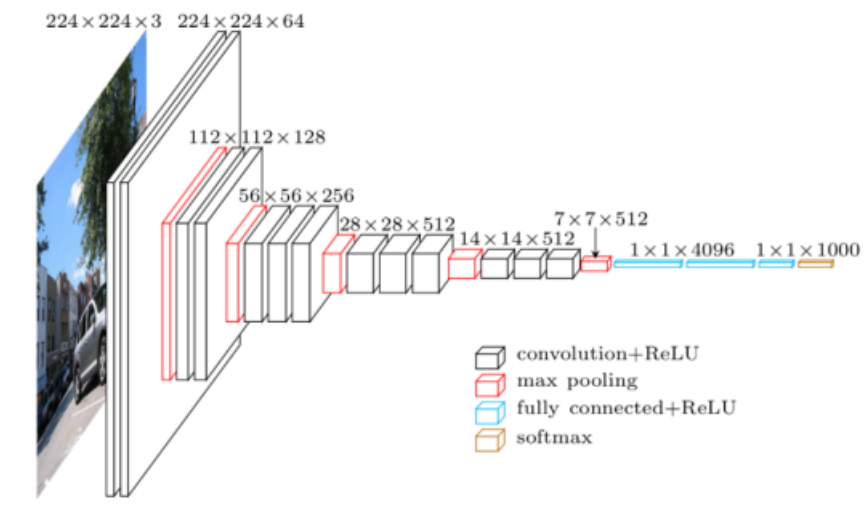


Figure 2.6: Example of a Convolutional Neural Network,Network "VGG-16

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

Figure 2.7: Convolution of an image with known kernels, from the traditional Computer Vision field: Blur, sharpen, edge detection.

Data arrangement in a CNN

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

2.1.4 2D Convolution Operation

When we apply 2D convolution, one tricky issue is that we tend to lose pixels on the perimeter of our image. Since we typically use small kernels compared to image size, we might only lose a few pixels but this may sum up when we apply many convolutional layers connected in a row. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the

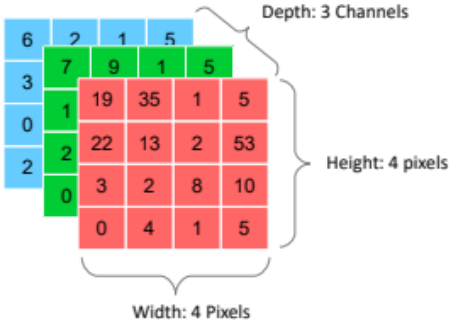


Figure 2.8: Example of an RGB image: A 3D matrix of size 4x4x3

image. Typically we set the extra pixels to 0 called Zero-Padding.

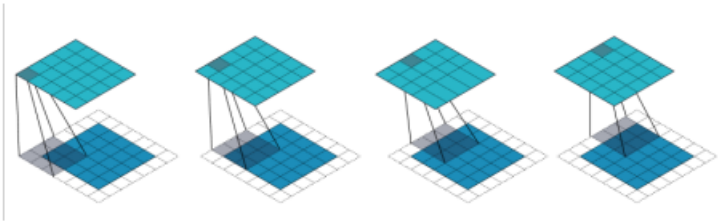


Figure 2.9: 2D convolution using "Same" Padding for 3x3 Kernel and 5x5 Image. Borders of Image are extended by zeros and we have the same size 5x5 output Image.

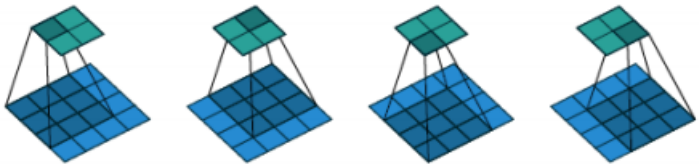


Figure 2.10: 2D Cross-correlation: a 3x3 kernel slides over a 4x4 input with unit stride and no padding (i.e $H_{in} = 4$ $K = 3$, $S = 1$, $P = 0$. The output Image has reduced size 2x2compared to the 4x4 input Image.

As mentioned before when running a convolution layer, we want an output with a lower size than the input. One way to accomplish this is by using a pooling layer. Yet another way to do it is to use a stride. The idea of stride is to skip some of the slide locations of the kernel. A stride of 1 means to pick slides a pixel apart, so basically every single slide, acting as a standard convolution. A strife of 2 means picking slides 2 pixels apart, skipping every other stride in the process, downsizing by roughly a factor of 2. A stride of 3 means skipping every 2 slides, downsizing by roughly a factor 3 and so on.

When a 2D input Image x is convolved with a kernel h of size $N \times M$ and the output

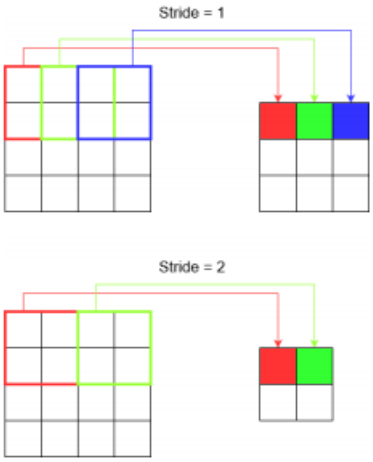


Figure 2.11: Comparing two different types of stride.

image is y, then to calculate output pixel (i, j) we use the following formula.

$$y(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[i+m, j+n] * h[n.m], i \in ImageWidth, j \in ImageHeight \quad (2.4)$$

2.1.5 Different type of layers used in a CNN

Convolution Layer

The convolution layer is the basic building block of a Convolutional Neural Network, as its name suggests. These are also the layers that occupy the most of the computation time. · A convolution layer extracts N output feature maps, from M input feature maps, by convolving each one of the M input feature maps with N filters.

Each one of the input feature maps has size of $Image_{width} \times Image_{height} = Im_w \times Im_h$. Since the number of input feature maps is M , then the input of the layer is a 3D matrix with size $M \times Im_w \times Im_h$. Each filter is of size $K \times K \times M$. Each one of the $K \times K$ 2D matrices of a filter is called a kernel. Each one of the $N \cdot K \times K \times M$ values that compose the N filters of a convolution layer is called a weight. If each input feature map is square, of size $H_{in} \times H_{in}$ and each output feature map is of size $H_{out} \times H_{out}$, then the convolution layer consists of $H_{in} \times H_{in} \times N$ neurons (height x width x depth) and $N \cdot K \cdot K \cdot M$ weights. The receptive field of each neuron is of size $K \times K \times M$ of the input 3D volume. A total of $H_{out} \cdot H_{out} \cdot N \cdot K \cdot K \cdot M$ multiplications is required for every convolution layer. The forward pass in these layers is computed as:

$$F = [n,i,j] = b[n] + \sum_{m=1}^{M} \sum_{x=0}^{K-1} \sum_{y=0}^{K-1} \Phi[m, i+x, j+y] w[n,m,x,y] \quad (2.5)$$

Where

- F is a tensor of output feature maps

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

Although it is called convolution layer, this layer actually performs a cross-correlation operation, also known as a sliding dot product, between each of the input feature maps and the filters' kernels. The 3D operation can be performed by adding the results of multiple 2D operations. In the equation 5.6 above, the two inner sums perform the 2D cross-correlation over an input feature map, while the external sum realizes the 3D operation by adding the results of all the M input feature maps at each kernel location. In the 2D cross-correlation operation, each "pixel" of the input feature map is replaced with a linear combination of its neighbours Figure 2.12 illustrates an example of the 3D convolution operation, while figure 2.10 illustrates how a kernel slides upon the input feature map during the 2D cross-correlation operation. The size of the output feature maps is dependent to Padding and Stride as it was mentioned before. Thus the size of the output feature map can be calculated with the following equation 2.5

$$H_{out} = \frac{H_{in} - K + 2P}{S} + 1 \tag{2.6}$$

where S is the stride with which the 2D kernel slides upon the 2D input feature map and P is the amount of padding used at the border of the input feature map. The size of the output feature map is $H_{out} \times H_{out}$ This is also the number of times that the 2D kernel fits inside the 2D input feature map. Looking a bit ahead, the computations described in equation 5.6 exhibit a large amount of potential parallelism, since all the multiplications involved are independent of one another. That means that all the $K \times K$ multiplications within a kernel can be computed concurrently, while all the M input feature maps can be convolved concurrently, while all the N output feature maps can be generated concurrently. Moreover, all the $H_{out} \times H_{out}$ "pixels" of each of the N output feature maps can be calculated simultaneously.

Pooling Layer The pooling layer operates independently on every feature map and it performs a spatial sub-sampling of its input. It is common to insert a pooling layer between successive convolution layers in a CNN architecture. The operation of this layer is similar to that of the convolution layer in the sense that there is a kernel that slides upon the input feature map. This time however, the kernel does not have a set of tunable weights. Instead, at each position it performs a predefined function on the corresponding pixels" of the input feature map. Common functions are the **max** and the **average**. More specifically the Maximum Pooling calculates the maximum value of each patch of the feature map, while the Average Pooling calculates the average value of each patch on the feature map. The pooling layer accepts a volume of size $H_1 \times W_1 \times D_1$ and outputs a volume of size $H_2 \times W_2 \times D_2$, where $W_2 = (W_1 - K)/S + 1$, $H_2 = (H_1 - K)/S + 1$ and $D_2 = D_1$. S is the stride with which the $K \times K$ kernel slides upon the feature map. The

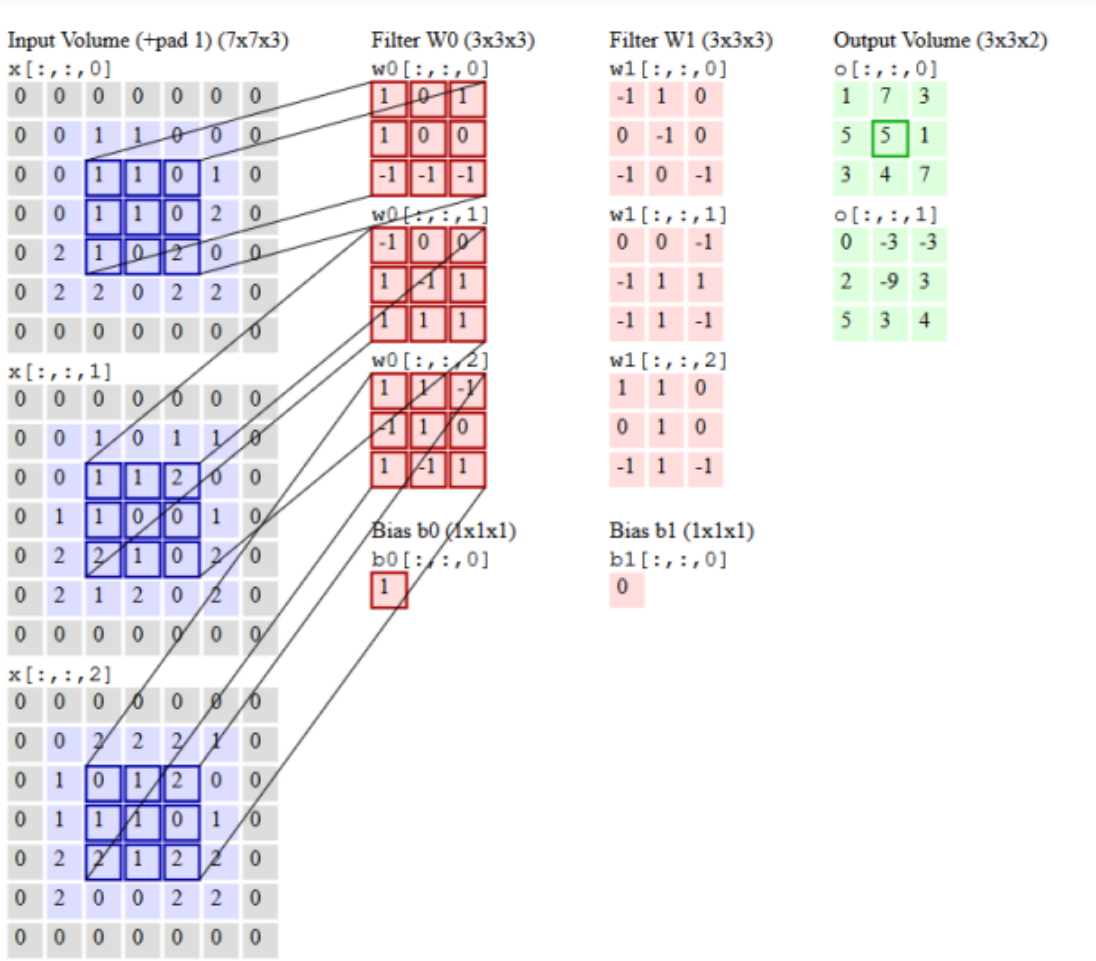


Figure 2.12: Example of 3D Convolution operation: Three input feature maps, gettingconvolved with two filters, generating two output feature maps. Padding=1, Stride=1. The marked pixel on the output feature map is a sum of all the dot products between the marked area of the input feature maps and the W0 filter's kernels

pooling layer leaves the depth dimension of its input unchanged. Figure 2.13 illustrates this procedure when the max pooling is utilized.

The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input. They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation. The results of max pooling are down sampled or pooled feature maps that highlight the most present feature in the patch.

Activation Layer/Function

The activation function is a node that is put at the end of or in between Neural Networks. They help to decide if the neuron would fire or not. If the input value of a neuron exceeds a threshold then the activation function fires this neuron otherwise it disables it.

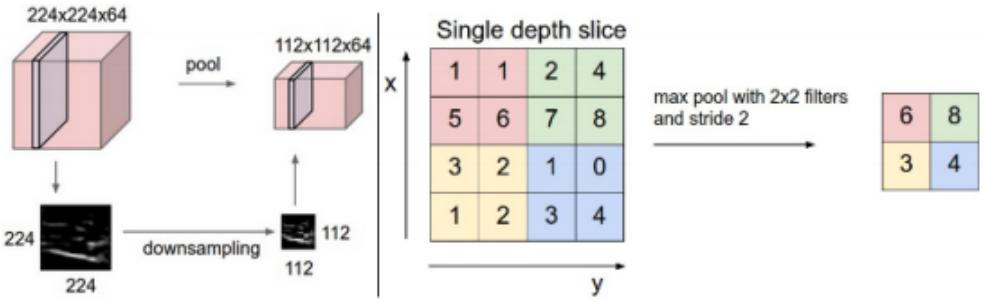


Figure 2.13: Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Right: Max pooling with a stride of 2. Here, each max is taken over 4 numbers and so a 2x2 square is produced

The activation function is an integral part of the artificial neuron's model. Without it, an Artificial Neural Network would just be performing a multiplication between its input and a weighted matrix, which means that the model of the classic ANN would degenerate to that of linear regression. Thus, an activation layer is being attached to every convolution layer and the element-wise operation is being performed at the output of every neuron. Figure 2.14 illustrates several of these functions:

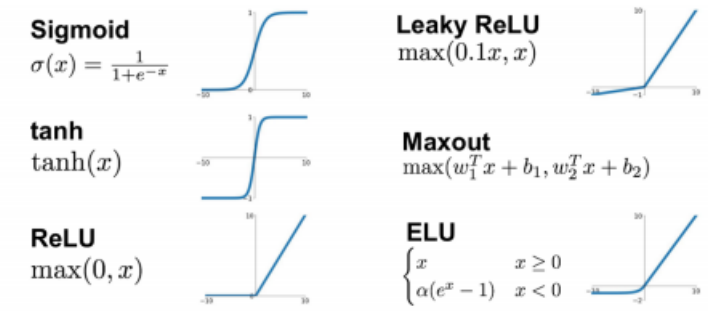


Figure 2.14: The most Popular Activation Functions in Neural Network

Among these, ReLU is the easiest one to implement and the less expensive one from a computational point of view, thus making it the most widely used activation function in neural networks nowadays. One of the greatest advantage ReLU has over other activation functions is that it does not activate all neurons at the same time. From the image for ReLU function above, we will notice that it converts all negative inputs to zero and the neuron does not get activated and this is why ReLU is very computational efficient as few neurons are activated per time.

Flattening and Fully-Connected Layers

Flattening and fully-connected layers are what we have at the last stage of CNN. Flattening is converting the data into a 1-dimensional array for inputting it to the next layer. We flatten the output of the convolutional layers to create a single long feature

vector. And it is connected to the final classification model, which is called a fully-connected layer, which will be described next. In other words, we put all the pixel data in one line and make connections with the final layer.

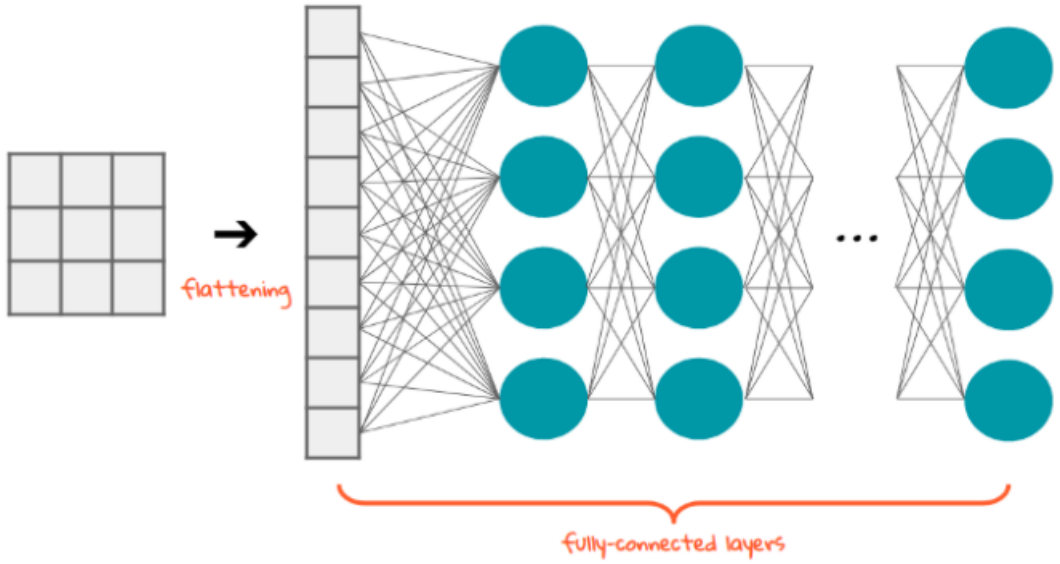


Figure 2.15: Once the pooled featured map is obtained, the next step is to flatten it. Flattening involves transforming the entire pooled feature map matrix into a single column which is then fed to the fully connected layers for processing.

The objective of a fully connected layer is to take the results of the convolution process and use them to classify the image into a label. As mentioned before the output of convolution and pooling process is flattened into a single vector of values. After flattening, the flattened feature map is passed through a neural network. This step is made up of the input layer, the fully connected layer, and the output layer. The fully connected layer is similar to the hidden layer in ANNs but in this case it's fully connected. This means that all neurons in these layer are connected to all neurons in the previous layer. The output layer is where we get the predicted classes. The information is passed through the network and the error of prediction is calculated. The fully connected part of the CNN network goes through its own backpropagation process to determine the most accurate weights. Each neuron receives weights that prioritize the most appropriate label. This means that the error is backpropagated through the system to improve the prediction.

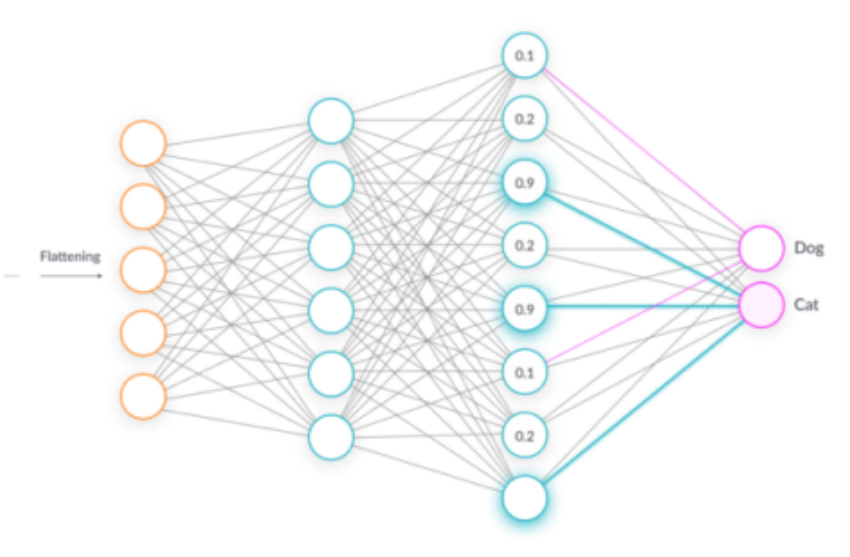


Figure 2.16: Example of the Fully Connected Part of a CNN that classifies images of cats and dogs.

Softmax Layer

The softmax layer is the most common classifier. A classifier layer is added after the last convolution or fully-connected layer in image classification CNNs, and normalizes the raw class scores Z_i produced from the rest of the network in the [0,1] range, to interpret them as probabilities P_i according to 2.7

$$P_i = \frac{e^{Z_i}}{\sum_{k=1}^{K} e^{Z_k}} \qquad i = 1, \ldots, K \tag{2.7}$$

2.1.6 In one shot

Figure 2.17 illustrates what has been described so far:

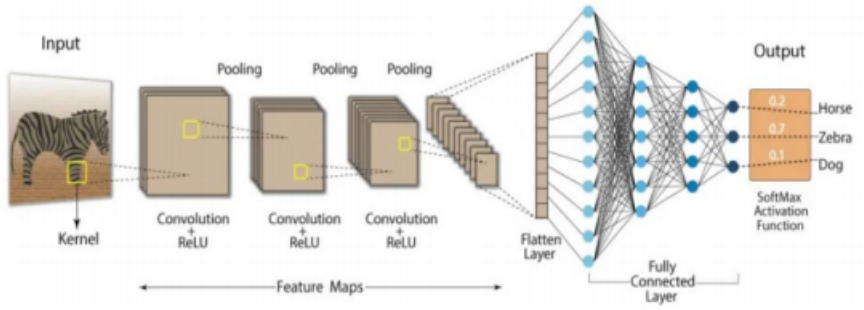


Figure 2.17: Example of how a CNN works

2.1.7 Resnet Architecture

Since all the experiments in this theses are being conducted on Resnet-8 , a brief description of its architecture will be given. Over the years, researchers tend to make deeper neural networks (adding more layers) to solve such complex tasks and to also improve the classification/recognition accuracy. But, it has been seen that as we go adding on more layers to the neural network, it becomes difficult to train them and the accuracy starts saturating and then degrades also. Here ResNet[22] comes into rescue and helps solve this problem. This problem of training very deep networks has been alleviated with the introduction of ResNet or residual networks and these Resnets are made up from Residual Blocks.

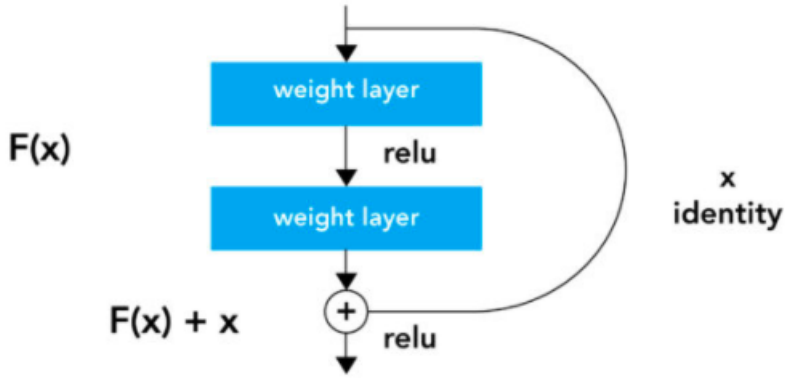


Figure 2. Residual learning: a building block.

Figure 2.18: Residual Learning: A Building Block

The very first thing we notice to be different is that there is a direct connection which skips some layers(may vary in different models) in between. This connection is called 'skip connection' and is the core of residual blocks. Due to this skip connection, the output of the layer is not the same now. Without using this skip connection, the input 'x' gets multiplied by the weights of the layer followed by adding a bias term.

Next, this term goes through the activation function, $f()$ and we get our output as $H(x)$:

$H(x) = F(wx + b)$ or $H(x) = F(x)$

Now with the introduction of skip connection, the output is changed to:

$$H(x) = F(x) + x \tag{2.8}$$

Figure 2.18 illustrates equation **??** . The intermediate building blocks consist mainly of convolutional networks that differ in number, size of features and sequence, depending on their architectures. What is really important about the ResNet architecture is that it

does not add additional parameters for training and therefore their use does not increase the complexity, unlike other architectures.

The smallest Resnet-8 network consists of three stages with $n = 1$ residual blocks in each stage(2.19)

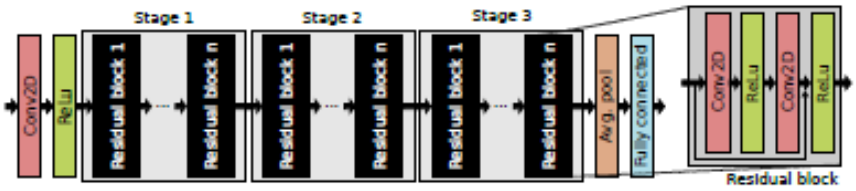


Figure 2.19: Architecture of ResNet convolutional neural network

2.2 Approximate Computing

2.2.1 The basics of approximate computing

The pervasive, portable, embedded and mobile nature of present age computing systems has led to an increasing demand for ultra low power consumption, small footprint and high performance. Approximate computing [21] is nascent computing paradigm that allow us to achieve these objectives by compromising the arithmetic accuracy. Many systems used in domains, like multimedia, neural networks and big data analysis, exhibit an inherent tolerance to a certain level of inaccuracies in computation and thus can benefit from approximate computing. The computational and storage demands of modern systems have far exceeded the available resources. It is expected that, in the coming decade, the amount of information managed by worldwide data centers will grow 50-fold, while the number of 5 processors will increase only tenfold. It is clear that rising performance demands will soon outpace the growth in resource budgets; hence, over provisioning of resources alone will not solve the conundrum that awaits the computing industry in the near future. Functional approximation[5], in hardware, mostly deals with the design of approximate arithmetic units, such as adders and multipliers, at different abstraction levels, i.e. transistor, gate, RTL and application. Some notable approximate adders include speculative adders [23], segmented adders[4] carry select adders[24] and approximate full adders[25]. Most of approximate multipliers have been designed at higher levels of abstraction, i.e., gate, RTL and application. Also, in the field of approximate multipliers, i.e., the most power-hungry component of hardware, accelerators, significant research has been conducted [11, 26, 27, 28]. A promising solution for this dilemma is approximate computing (AC), which is based on the intuitive observation that, while performing exact computation or maintaining peak-level service demand require a high amount of resources, allowing selective approximation or occasional violation of the specification can provide disproportionate gains in efficiency. For example, for a k-means clustering algorithm, up

to 50 times energy saving can be achieved by allowing a classification accuracy loss of 5% [29]. Similarly, a neural approximation approach can accelerate an inverse kinematics application by up to 26 times compared to the GPU execution, while incurring an error of less than 5% [30]. Approximate computing and storage approach leverages the presence of error-tolerant code regions in applications and perceptual limitations of users to intelligently trade off implementation, storage, and/or result accuracy for performance or energy gains. In brief, approximate computing exploits the gap between the level of accuracy required by the applications/users and that provided by the computing system, for achieving di-verse optimizations. Thus, approximate computing has the potential to benefit a wide range of applications/frameworks, for example, data analytics, scientific computing, multimedia and signal processing, machine learning and MapReduce, etc. However, although promising, approximate computing is not a panacea. Effective use of approximate computing requires judicious selection of approximable code/data portions and approximation strategy, since uniform approximation can produce unacceptable quality loss [31, 32, 33]. Even worse, approximation in control flow or memory access operations can lead to catastrophic results such as segmentation fault [34] Further, careful monitoring of output is required to ensure that quality specifications are met, since large loss makes the output unacceptable or necessitates repeated execution with precise parameters. Clearly, leveraging the full potential of approximate computing requires addressing several issues.

2.2.2 Approximate Arithmetic Circuits

There are two basic approximate arithmetic components: *approximate adders* and *approximate multipliers*.

Approximate Adders

In approximate implementations, multiple-bit adders are divided into two modules: the *accurate* upper part of more significant bits and the *approximate* lower part of less significant bits. For each lower bit, a single-bit approximate adder implements a modified, thus inexact function of the addition. This is often accomplished by simplifying a full adder design at the circuit level, equivalent to a process that alters some entries in the truth table of a full adder at the functional level.

Approximate Multipliers

Significant research has been conducted on the optimization of multiplication circuits [35], i.e., the key processing units of DSP accelerators, which inherently affect the performance of the entire application. In this context, several approximate multipliers have been proposed in the literature. Approximate multipliers that use the speculative adders to compute sum of partial products have been designed in [36],[37]. However, the straightforward application of approximate adders in a multiplier may be inefficient in terms of of trading off accuracy for savings in energy and area. For an approximate multiplier, the main key design aspect is reducing the critical path by adding the partial products. Since multiplication is usually implemented by a cascaded array of adders, some of the least

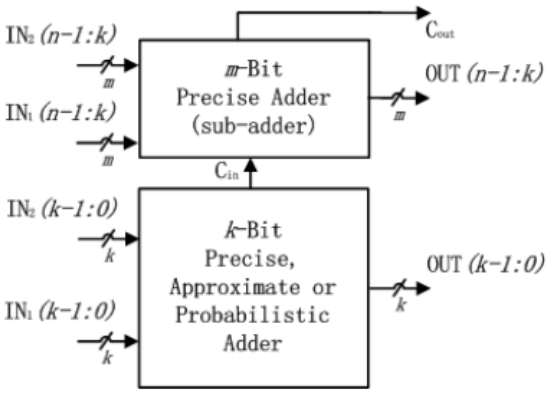


Figure 2.20: A general architecture for an approximate adder divided into two modules: the accurate MSBs and approximate LSBs

significant bits in the partial products are simply omitted (with some error compensation mechanisms) and also some adders can be removed in array for a faster operation.

The approximate multipliers are classified into four categories:

1. Approximation in Generating Partial Products
2. Using simpler structure to generate partial products
3. Approximation in the Partial Product Tree
4. Omitting some partial products

In [5], a large multiplier is constructed by 2×2 simplified multipliers to reduce arithmetic and computation complexity. An efficient design that uses input preprocessing and additional error compensation is proposed in [6] to reduce the critical path delay. Combinations of both partial product generation and approximations are applied in collaboration to further reduce power consumption [7], [8], [9]. The main goal in nowadays research considering approximate multipliers is to reduce the number of partial products using hybrid radix encoding [10] to apply approximations on the partial product generation. Finally, approximate floating-point multipliers have been proposed in the literature [38], [39].

2.3 Related Work

Approximations in Neural Networks

As neural networks are inherently error-resilient, various approaches have been proposed to approximate them. A straightforward approach for the automated construction of NNs with approximate CP is to optimize the bit precision for the data structures used in NN [40]. A recent research shows that in specific cases one bit can be sufficient to

represent the weights[41]. Let us suppose that the bit width is fixed to n bits due to architectural constraints. There are several ways how to improve the energy efficiency of the n-bit arithmetic operations. Venkataramani et al.[31] proposed a methodology of identifying error-resilient neurons based on the backpropagation gradients. For the error-resilient neurons, an approximation using precision modification and piecewise-linear approximation of activation function was applied to create an approximate neural network. Since training is by itself an error-healing process, after creating the approximate version, the NN is retrained. They also proposed a neuromorphic processing engine platform to determine the best tradeoff between the precision and energy. Zhang et al.[42] used a different approach for the critical neuron identification. A neuron is considered as critical, if small jitters on the neurons computation introduce a large output quality degradation; otherwise, the neuron is resilient.They presented a theoretical approach for finding the critical neurons. The least critical neurons are candidates for approximation. Due to the tight interconnection between the neurons, the ranking of candidate neurons is updated after approximation of each neuron. Hence, an iterative algorithm for the criticality ranking and approximation was developed. Three approximation strategies were used – precision scaling, memory access skipping and approximating the multiplier circuits. To increase the overall accuracy, the resulting neural networks were retrained. This approach was only evaluated on a MLP. In the case of CNNs, Mrazek et al. [43] introduced approximate multipliers to convolutional layers of the LeNet neural network. They showed that the back-propagation algorithm can adapt the weights of CNN to the used approximate multipliers and significant power saving can be achieved for a negligible loss in accuracy. Approximate multipliers based on the principles of multiplierless multiplication were introduced to complex CNNs in [44].The authors modified the learning algorithm in such a way that only those weights could be used for which an efficient implementation of approximate multiplication exists. The authors showed that the approximations can provide significant power savings in the computational path even for deep neural networks. However, the major limitation of this approach is that arbitrary approximate multiplier cannot be introduced to the NN.Although the aforementioned approximation methods decrease the accuracy of the NNs, the resulting NNs can be beneficial for other approaches, for example, in progressive chain classifiers (PCCs) [45]. In PCCs, there is a chain of classifier models that progressively grow in complexity and accuracy. After evaluating a stage it is checked whether its confidence is high; if so the remaining stages of the PCC are not evaluated.

Chapter 3

Tensorflow Approximate Layers

This chapter's goal is to present and describe the basic library/tool called TensorFlow Approximate Layers [1] that was used in this theses. This library was extended via various ways which will be described in the next chapter.

3.1 Tensorflow

First of all, an introduction to Tensorflow is considered necessary before we proceed any further.

3.1.1 What is TensorFlow?

TensorFlow is an open-source end-to-end platform for creating Machine Learning applications. It is a symbolic math library that uses dataflow and differentiable programming to perform various tasks focused on training and inference of deep neural networks. It allows developers to create machine learning applications using various tools, libraries, and community resources. Currently, the most famous deep learning library in the world is Google's TensorFlow. Google product uses machine learning in all of its products to improve the search engine, translation, image captioning or recommendations. TensorFlow is a library developed by the Google Brain Team to accelerate machine learning and deep neural network research. It was built to run on multiple CPUs or GPUs and even mobile operating systems, and it has several wrappers in several languages like Python, C++ or Java.

3.1.2 How TensorFlow Works

TensorFlow enables you to build dataflow graphs and structures to define how data moves through a graph by taking inputs as a multi-dimensional array called Tensor. It allows you to construct a flowchart of operations that can be performed on these inputs, which goes at one end and comes at the other end as output.

3.1.3 TensorFlow Architecture

Tensorflow architecture works in three parts:

1. Preprocessing the data
2. Build the model
3. Train and estimate the model

It is called Tensorflow because it takes input as a multi-dimensional array, also known as *tensors*. You can construct a sort of *flowchart* of operations (called a Graph) that you want to perform on that input. The input goes in at one end, and then it flows through this system of multiple operations and comes out the other end as output. This is why it is called TensorFlow because the tensor goes in it flows through a list of operations, and then it comes out the other side.

3.1.4 Basic components of Tensorflow

- **Tensor**

Tensorflow's name is directly derived from its core framework: **Tensor**. In Tensorflow, all the computations involve tensors. A tensor is a vector or matrix of n-dimensions that represents all types of data. All values in a tensor hold identical data type with a known (or partially known) shape. The shape of the data is the dimensionality of the matrix or array. A tensor can be originated from the input data or the result of a computation. In TensorFlow, all the operations are conducted inside a **graph**. The graph is a set of computation that takes place successively. Each operation is called an **op node** and are connected to each other. The graph outlines the ops and connections between the nodes. However, it does not display the values. The edge of the nodes is the tensor, i.e., a way to populate the operation with data.

- **Graphs**

TensorFlow makes use of a graph framework. The graph gathers and describes all the series computations done during the training. The graph has lots of advantages:

1. It was done to run on multiple CPUs or GPUs and even mobile operating system
2. The portability of the graph allows to preserve the computations for immediate or later use. The graph can be saved to be executed in the future.
3. All the computations in the graph are done by connecting tensors together. A tensor has a node and an edge. The node carries the mathematical operation and produces an endpoints outputs. The edges the edges explain the input/output relationships between nodes.

3.1.5 Why is Tensorflow popular?

TensorFlow is the best library of all because it is built to be accessible for everyone. Tensorflow library incorporates different API to built at scale deep learning architecture like CNN or RNN. TensorFlow is based on graph computation; it allows the developer to visualize the construction of the neural network with Tensorboard. This tool is helpful to debug the program. Finally, Tensorflow is built to be deployed at scale. It runs on CPU and GPU.

3.2 Approximate Convolutional Layers

3.2.1 Overview of the library

The library[1] on which this thesis is based, is an open-source library of approximate convolutional layers. This library extends TensorFlow library providing Approximate Convolutional (ApproxConv) layers, i.e. layers with reduced precision (typically 8 bits) implemented using approximate multipliers. More specifically this library extends TensorFlow library by **AxConv2D** layer that implements **QuantizedConv2D** layer with approximate multiplier. The proposed layer enables to specify via parameter which approximate multiplier should be used. The approximate multipliers are implemeted in C. The layers optionally allow to use weight tuning algorithm that tries to modify weights of the layer to minimize mean arithmetic error of the multipliers. In contrast with standard implementation, the proposed layer introduces additional two parameters: AxMult(str) and AxTune(bool). It should be noted that this library can be used for inference path only.

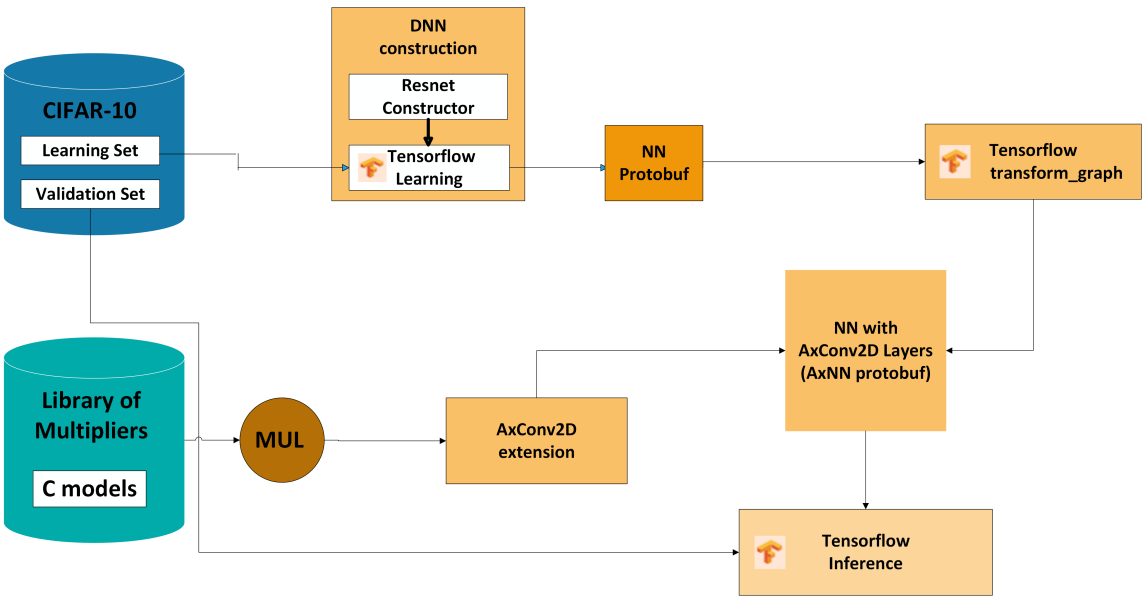


Figure 3.1: Tensorflow Approximate Layers Library

3.2.2 How does this library work?

ResNet networks [22] are chosen and trained to recognize images from CIFAR-10 dataset. The resulting neural networks are frozen, quantized and the convolutional layers are replaced by approximate multipliers by means of *transform graph* tool.

3.2.2.1 Freezing Tensorflow models

Neural networks are computationally very expensive. For example Alexnet[46] has more than 60 million parameters that are needed to calculate a prediction on one image. Apart from it, there is also a similar number of gradients that are calculated and used to perform backward propagation during training. Tensorflow models contain all of these variables. However someone don't need the gradients when he deploys his model on a webserver so why carry all this load. Freezing is the process to identify and save all of required things(graph, weights etc) in a single file that you can easily use.

A typical Tensorflow model contains 4 files:

1. **Model-ckpt.meta**: This contains the complete graph. It contains the graphDef that describes the data-flow, annotations for variables, input pipelines and other relevant information
2. **Model-ckpt.data-0000-of-00001**: This contains all the values of variables(weights, biases, placeholders,gradients, hyper-parameters etc).
3. **Model-ckpt.index**: Metadata.
4. **Checkpoint**: All checkpoint information

So, in summary, we want to get rid of unnecessary meta-data, gradients and unnecessary training variables and encapsulate it all in a single file . This single encapsulated file (.pb extension) is called *frozen graph def*. It's essentially a serialized graph def protocol buffer written to disk.

3.2.2.2 Quantization on Neural Networks

Quantization focuses on reducing the size of the model as well as the inference time of a CNN and at the same time have minimal accuracy losses. Despite the abundance of quantization methods, a fairly efficient technique in the programming environment of Tensorflow is the technique of 8-bit quantization. There are several reasons in order for us to us this efficient technique:

1. Arithmetic with lower bit-depth is faster, assuming the hardware supports it. Even though floating-point computation is no longer "slower" than integer on modern CPUs, operations with 32-bit floating point will almost always be slower than, say, 8-bit integers.

2. In moving from 32-bits to 8-bits, we get almost $4\times$ reduction in memory straightaway. Lighter deployment models mean they hog less storage space, and are easier to share over smaller bandwidths.

3. Lower bit-widths also mean we can squeeze more data into the same caches/registers. This means we can reduce how often we access things from RAM, which usually consumes a lot of time and power.

4. Floating point arithmetic is hard — which is why it may not always be supported on microcontrollers on some ultra low-power embedded devices, such as drones, watches, or IoT devices. Integer support, on the other hand, is readily available.

There has been an increasing amount of work in quantizing neural networks, and there are, broadly speaking, two reasons for this. First, DNNs are known to be quite robust to noise and other small perturbations once trained. This means even if we subtly round off numbers, we can still expect a reasonably accurate answer. Moreover, the weights and activations by a particular layer often tend to lie in a small range, which can be estimated beforehand. This means we don't need the ability to store 10^6 and 10^{-6} in the same data type — allowing us to concentrate our precious fewer bits within a smaller range, say -3 to $+3$. As you might imagine, it'll be crucial to accurately know this smaller range — a recurring theme you'll see below. So, if done right, quantization only causes a small loss of precision, which usually doesn't change the output significantly.

- **8-Bit Quantization uint**

First, a quick primer on floating/fixed-point representation. Floating point uses a mantissa and an exponent to represent real values and both can vary. The exponent allows for representing a wide range of numbers, and the mantissa gives the precision. The decimal point can *float* meaning it can appear anywhere relative to the digits.

If we replace the exponent by a fixed scaling factor, we can use integers to represent the value of a number relative to (i.e. an integer multiple of) this constant. The decimal point's position is now *fixed* by the scaling factor. Going back to the number line example, the value of the scaling factor determines the smallest distance between 2 ticks on the line, and the number of such ticks is decided by how many bits we use to represent the integer (for 8-bit fixed point, 256 or 28). We can use these to tradeoff between range and precision. Any value that is not an exact multiple of the constant will get rounded to the nearest point.

Unlike floating-point, there is no universal standard for fixed-point numbers, and is instead domain-specific. Our quantization scheme (mapping between real and quantized numbers) requires the following:

1. **It should be linear or affine**. If it isn't, then the result of fixed-point calculations won't directly map back to real numbers.

2. **It allows us to always represent 0.f accurately**. If we quantize and dequantize any real value, only 256 or generally, 2^B of them will return the exact the same number, while all others will suffer some precision loss. If we ensure that 0.f is one of these 256 values , it turns out that CNNs can be quantized more accurately. The authors claim that this improves accuracy because 0 has a special significance in CNNs (such as padding). Besides, having 0.f map to another value that's higher/lower than zero will introduce a bias in the quantization scheme.

So our quantization scheme will simply be a shifting and scaling of the real number line to a quantized number line. For a given set of real values, we want the minimum/maximum real values in this range [rmin,rmax] to map to the minimum/maximum integer values [0,2^{B-1}] respectively, with everything in between linearly distributed.

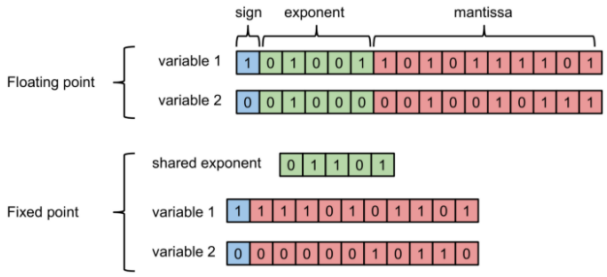


Figure 3.2: Floating point versus Fixed point

This gives us a pretty simple linear equation:

$$r = \frac{r_{max} - r_{min}}{2^{B-1}} \times (q - z) = S \times (q - z) \tag{3.1}$$

Here,

1. r is the real value (usually float32)
2. q is its quantized representation as a B-bit integer (uint8, uint32, etc)
3. S (float32) and z (uint) are the factors by which we scale and shift the number line. Here z is the quantized 'zero-point' which will always map back exactly to 0.f.

Therefore, thanks to *transform graph* tool the Resnet is implemented with quantized weights and constants of type 8-bit uint, ie positive integers with values from 0-255

3.2.2.3 Replacement of Convolutional Layer by Approximate Layers

After the neural network is frozen as it was mentioned earlier, it must be quantized and the convolutional layer replaced by approximate implementation. In order to quantize the frozen neural network and replace the convolutional layers by approximate ones, the *transform graph* tool is used. The Graph Transform framework offers a suite of tools

for modifying computational graphs, and a framework to make it easy to write your own modifications. The Graph Transform tool is designed to work on models that are saved as GraphDef files, usually in a binary protobuf format. This is the low-level definition of a TensorFlow computational graph, including a list of nodes and the input and output connections between them. The binary protobuf format usually has a ".pb" suffix.

The Graph Transform tool can be called like this:

```
$bazel build tensorflow/tools/graph_transforms:transform_graph
bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
        --in_graph=tensorflow_inception_graph.pb \
        --out_graph=optimized_inception_graph.pb \
        --inputs='Mul:0' \
        --outputs='softmax:0' \
        --transforms='
strip_unused_nodes(type=float, shape="1,299,299,3")
remove_nodes(op=Identity, op=CheckNumerics)
fold_old_batch_norms
'$
```

The arguments here are specifying where to read the graph from, where to write the transformed version to, what the input and output layers are, and what transforms to modify the graph with. The transforms are given as a list of names, and can each have arguments themselves. These transforms define the pipeline of modifications that are applied in order to produce the output. Sometimes you need some transforms to happen before others, and the ordering within the list lets you specify which happen first.

In our case, after the Resnet-8 is trained to recognized images from CIFAR-10 dataset it is frozen , so that the binary protobuff file can be produced. After that the neural network is quantized and and convolutional layers are replaced by approximate multipliers by means of *transform graph* tool using the following commands:

```
tensorflow/bazel-bin/tensorflow/tools/graph_transforms/
    transform_graph \
            --in_graph=resnet_8.pb \
            --out_graph=resnet8_quant_ax.pb \
            --outputs='resnet/tower_0/fully_connected/dense/
                BiasAdd:0' \
            --inputs='input' \
            --transforms='
            add_default_attributes
            strip_unused_nodes(type=float, shape="1,299,299,3")
            remove_nodes(op=Identity, op=CheckNumerics)
            fold_constants(ignore_errors=true)
            fold_batch_norms
```

```
    quantize_weights
    quantize_nodes
    strip_unused_nodes
    sort_by_execution_order
    rename_op(old_op_name=QuantizedConv2D,new_op_name=
        AxConv2D)'
```

We have to specify the input and output node in order to use this tool. This can be thanks to **Tensorboard**. TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking experiment metrics like loss and accuracy, visualizing the model graph, projecting embeddings to a lower dimensional space, and much more. Here we use Tensorboard in order to visualize the model graph and find the input and output node. Figure 3.3 illustrated the graph of Resnet-8 which is visualized with Tensorboard.

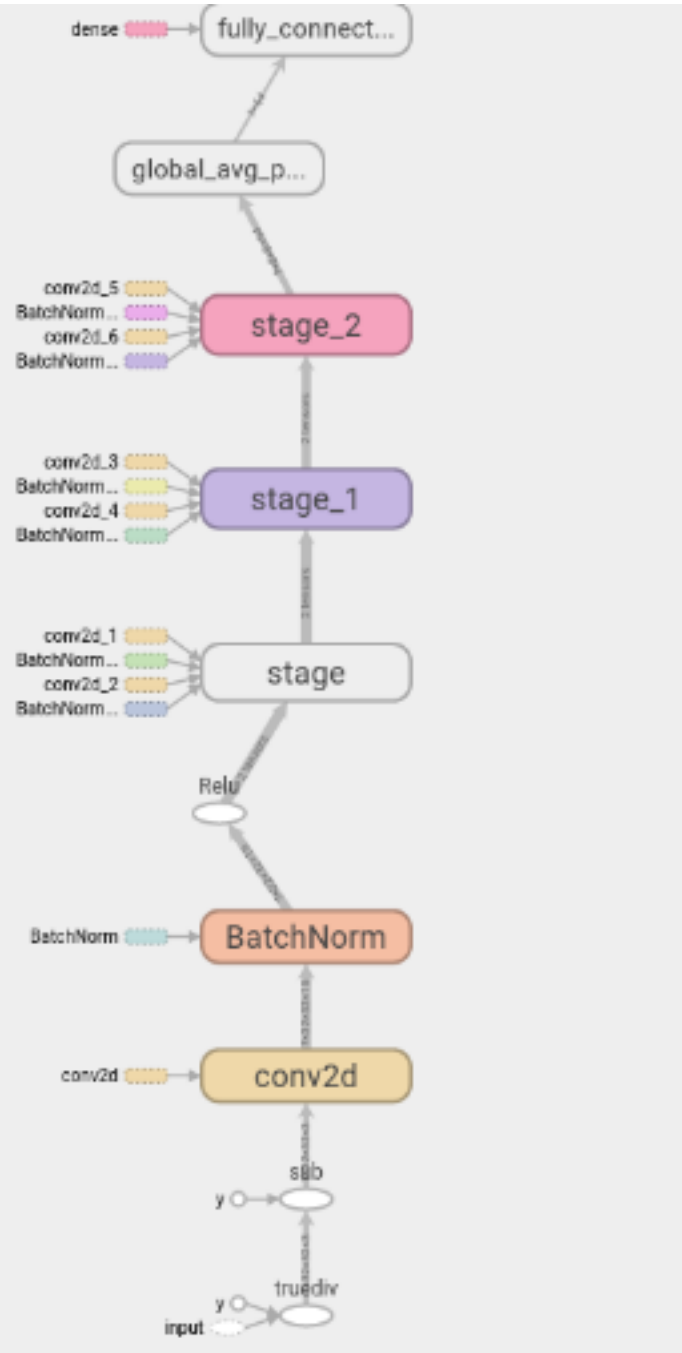
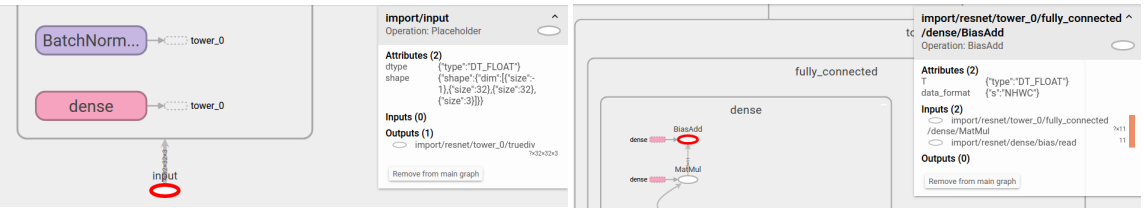


Figure 3.3: Graph of the Resnet-8 provided by Tensorboard

Figure 3.4a illustrates the input node on the left and the output node on the right.



(a) Input node **(b)** Output Node

Figure 3.4: Visualization of input and output node

Two other important commands which are given in the *tranform graph* tool are **quantizeweights** and **quantizenodes**.

- The command **quantizeweights** converts any large float Const op into an **eight-bit** equivalent, followed by a float conversion op so that the result is usable by subsequent nodes. This is mostly useful for shrinking file sizes, but also helps with the more advanced quantizenodes transform.

- The command **quantizenodes** replaces any calculation nodes with their eight-bit equivalents (if available), and adds in conversion layers to allow remaining float operations to interoperate. This is one of the most complex transforms, and involves multiple passes and a lot of rewriting. It's also still an active area of research, so results may vary depending on the platform and operations we are using in our model 3.5 Quantizeweights should be run first in order to ensure your Const ops are in eight-bit form. This commands needs the following arguments:

  1. the *inputmin* which is the lowest float value for any quantized placeholder inputs

  2. the *inputmax* which is the highest float value for any quantized placeholder inputs. If both inputmin and inputmax are set, then any float placeholders in the graph will be replaced with quantized versions, and consts will be created to pass the range to subsequent operations.

  3. The *fallbackmin* which is the lowest float value to use for requantizing activation layers.

  4. The *fallbackmax* which is the highest float value to use for requantizing activation layers. If both fallbackmin and fallbackmax are set, then instead of using RequantizationRange ops to figure out the useful range dynamically when converting the 32-bit output of ops like QuantizedConv2D and QuantizedBiasAdd, hardwired consts with these values will be used instead. This can help performance, if you know the range of your activation layers ahead of time.

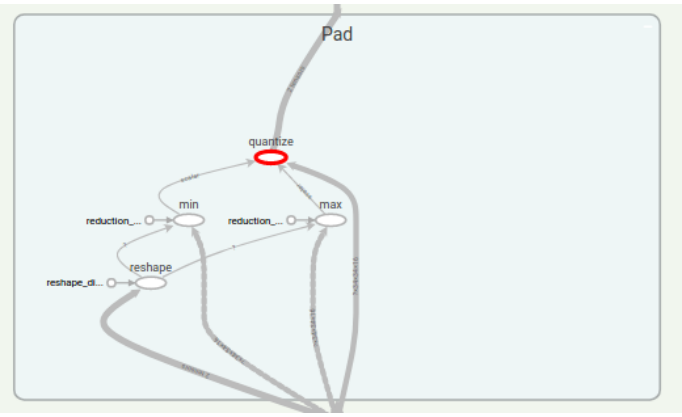
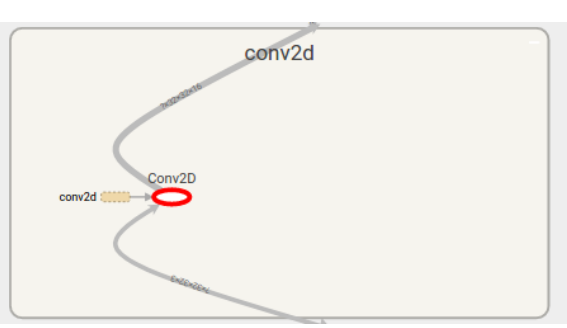
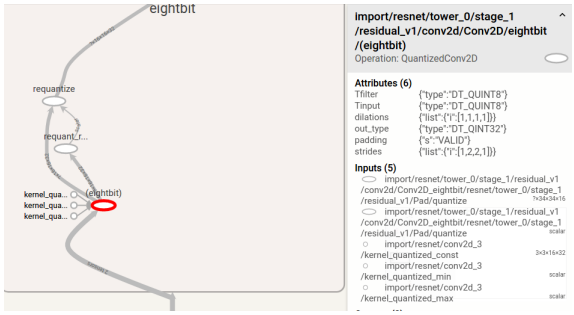


Figure 3.5: Quantization operation in the graph as described above



(a) Conv2D operation before quantization



(b) Conv2D operation (QuantizedConv2D) after quantization

Figure 3.6: Conv2D vs QuantizedConv2D

Finally the most important command is the **rename op**(old opname=QuantizedConv2D,new op name=AxConv2D). After the quantization, the old Conv2D operation becomes QuantizedConv2D. Now we want to be able to support approximate multipliers in the now quantized convolutional layer. In order to do that we use the rename op command which takes as input the current name of the operation (QuantizedConv2D) and transforms it to the name we want to change it to (AxConv2D). In other words, this command finds all ops with the given name, and changes them to the new one.

AxConv2D is an operation that is not registered in the Tensorflow Kernel. The authors in [1] implement this operation in C++ and after its implementation ,the operation is registered in the Tensorflow Kernel. This new operation implements quantized eight-bit versions of the convolution operations. However, the important difference between this AxConv2D and QuantizedConv2D is that AxConv2D supports the use of **approximate multipliers** in the convolutions operations. The memory layout of the data in this new operation is from biggest stride to smallest:

- input data = [input batches, input height, input width, input depth]
- filter data = [filter height, filter width, input depth, filter count]

- output data = [input batches, output height, output width, filter count]

Basically AxConv2D expects expects two 4D input tensors and produces another 4D tensor.The first input tensor represents a batch of 3D input images given in NHWC format ($Batch \times Height \times Width \times Channels$), where the number of channels corresponds with the fastest changing index. The second tensor is a set of 3D filters (or kernels of the convolution) stored in the $Height \times Width \times Channels \times FilterCount$ format, where Filter Count specifies the number of filters applied to the same input. The output of the convolution shares the same layout as the input data. However, the height and width are determined according to the shape of the kernel and the depth of each output image depends on the number of applied filters. This leads to a system of nested loops (over each input image in the batch, each output pixel, each output channel etc.) The approximate version of the 2D convolution is extended by four scalar inputs that provide the minimum and maximum values computed independently for each input vector

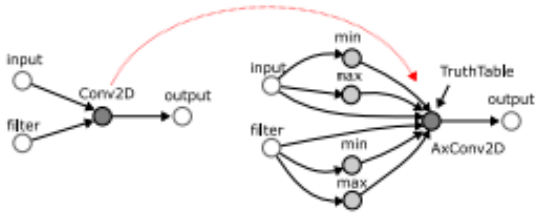


Figure 3.7: Introducing the approximate convolutional layer (AxConv2D) into the existing graph consisting of a single convolutional layer Conv2D [2]

Furthermore the new AxConv2D operation supports the use of **approxiamate multipliers** in the convolution operation. This means that the user can give as an input an arbitrary approximate multiplier of his choice and the convolution operations inside the neural network will be performed by this multiplier instead of the accurate.

3.2.3 Summary of the process

ResNet networks are chosen and trained to recognize images from CIFAR-10 dataset. The resulting NNs were frozen, quantized and convolutional layers were replaced by approximate multipliers by means of transform graph tool.

- **CIFAR-10**

The CIFAR-10 dataset consists of 60000 32×32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches

contain exactly 5000 images from each class. In the following figure 3.8 classes in the dataset, as well as 10 random images from each class are presented.

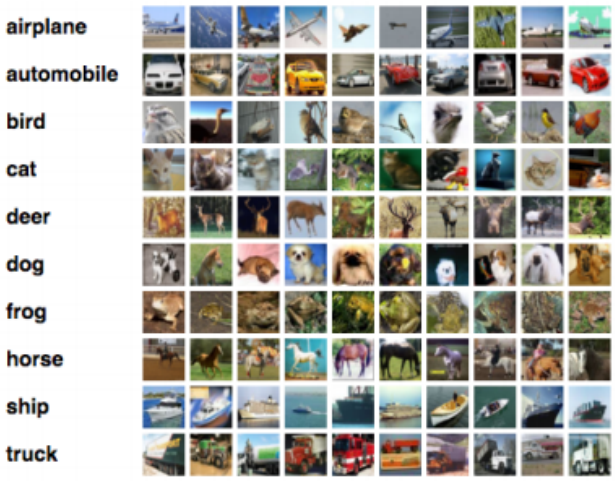


Figure 3.8: CIFAR 10 CNN dataset presentation

3.2.4 Usage and Example

The library is implemented symbolic library that is dynamically included. This means that firstly it must be build. This can be done in a Linux terminal with the following commands:

```
cd axqconv
make
```

Then it can be included to the run in a python script with the following commands:

```
import tensorflow as tf
tf.load_op_library('../axqconv/axqconv.so')
```

In order to better understand this library a simple example that approximates the ResNet-8 neural network trained for CIFAR-10 dataset, will be given. Firstly, the dataset must be downloaded and preprocessed. After the model has been trained, it is then frozen and the protobuf file resnet-8.pb is produced which contains the graph definition as well as the weights of t
Then, the frozen network must be quantized and the layer replaced by approximate implementation, which is something that is done with the help of tranform graph tool and thus the protobuf file resnet-8-quant-ax.pb is generated. In order to perform the inference of approximate Resnet-8 there is a script written in python named cifar10-ax-inference.py. This inference script approximates the input Neural Network with the same multiplier (uniform structure) and gives as an output the **inference accuracy** of the approximated neural network. The input arguments of this script are:

- **graph**: The frozen graph which in our case is resnet-8-quant-ax.pb

- **data-dir**: The directory where the CIFAR-10 input data is stored
- **mult**: Name of multiplier. Can be any multiplier the user wishes to use.
- **batch-size**:Number of images in the batch. By default it is 1000
- **iterations**:Number of iterations of batches, $batchsize \cdot iterations <= datasize$, $batchsize >> iterations$. By default it is 10

From a more technical point of view, the inference script searches the input graph for nodes that have execute the operation **AxConv2D**. These are as mentioned earlier convolutional 2D operations that have been quantized and replaced with approximate implementation with the use of approximate multipliers. Whenever it finds a node with this new operation, the operation of convolution with the use of the approximate multiplier that the user has specified is performed. This new operation has been registered as mentioned earlier in the Tensorflow Kernel and has been written in C++.

In order to better understand how this library [1] works , we are going to demonstrate an example. In the following example , we are going to give as input graph the resnet-8-quant-ax.pb, (i.e the Resnet-8 that has been trained with CIFAR-10 dataset and frozen) and as an input multiplier the accurate multiplier **mul8u-1JFF**. This multiplier is from EvoApproxLib [3], a library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. The python inference script is run on Linux using the following command the output screen is the this:

```
python3 cifar10_ax_inference.py cifar-10-graphs/
    resnet_8_quant_ax.pb --data-dir cifar-10-data/ --tune true --
    mult mul8u_1JFF
```

The inference script carries out 10 runs(iterations) each with a batch size of 1000. After the 10 runs the whole validation set consisting of 10000 images has been tested. Each run produces an number which represents the inference accuracy and after the termination of the inference script the average inference accuracy is produced which in our example is**0.833**

Chapter 4

Approximation Techniques

This chapter's goal is to present and describe the different approximation techniques we propose in this thesis. The approximation techniques we propose are extensions of the library already described in the previous chapter. What we want to achieve is to give the opportunity to any user to test his/her arbitrary multipliers on the proposed different approximation techniques. Thus with these proposed approximation techniques we offer a new extended library of Tensorflow.

4.1 Our Goals

Contrary to the library proposed in [1] , in our proposed extended library of approximation techniques, we offer two new main innovations:

1. Support of **multiple** approximate multipliers at the same time instead of only 1 as proposed in [1]

2. Approximations at different levels , which will be analyzed on the following sections

.

First of all, as mentioned above the library [1] , could support the use of only one multiplier for the convolution operations all over the Neural Network. We extended this library by providing the ability to any user to give as input up to 3 different arbitrary multipliers of his/her choice. This was done by modifying the operation AxConv2D which was mentioned in the previous chapter. From a more technical point of view, we modified the C++ file namely axconv.cc which defines the operation of AxConv2D as well as the inference python script. We enabled on both of these files to support the use of multiple multipliers instead of only one , by adding two more parameters in each of them.

Finally, regarding the proposed approximations at different levels, these were done with two different ways:

- Replacement of multiplications with diverse approximate components

- Computation skipping

These two methods were used in order to create the proposed approximation techniques on various levels which will be described below. These techniques will be described beginning from an upper level and gradually reducing the level of approximation.

It should be noted that all the approximation techniques are conducted on a Resnet-8, so from now on we will be reffering only to Resnet-8. Resnet-8 consists of 7 convolutional layers.

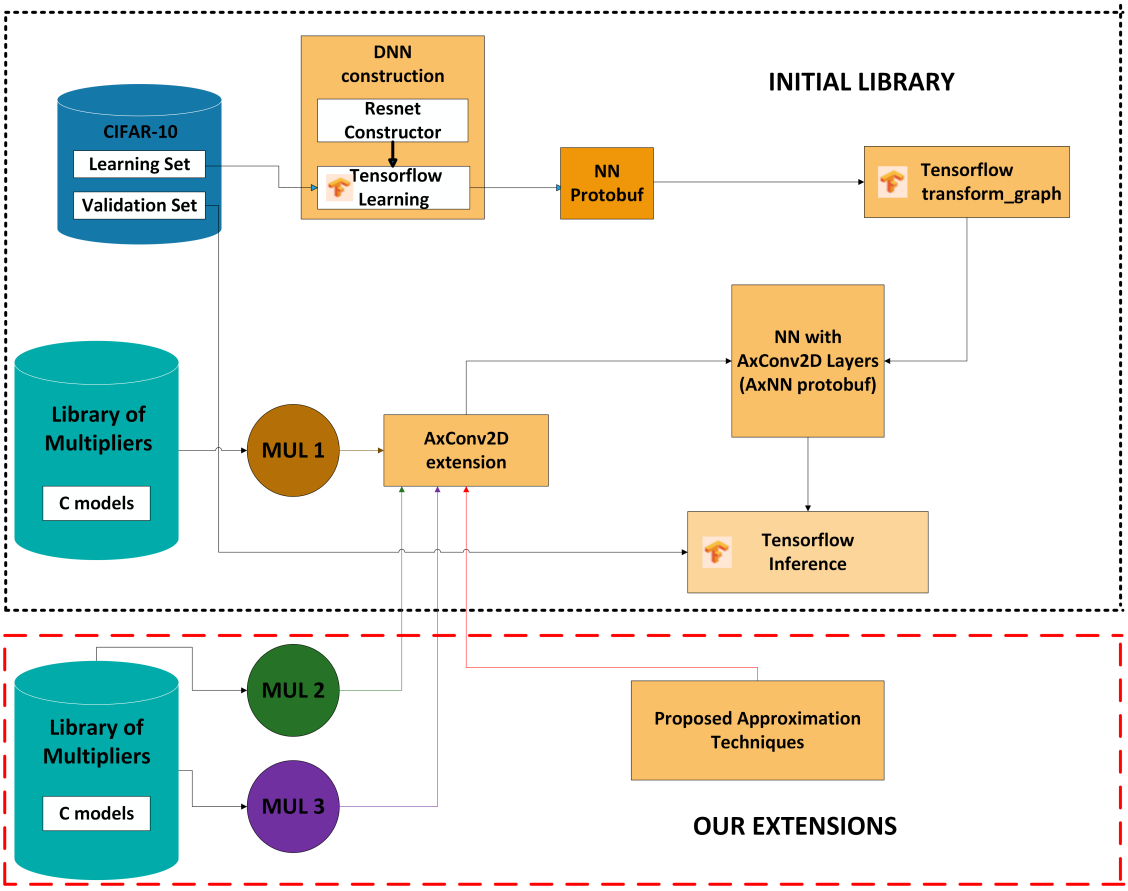


Figure 4.1: Our proposed extensions

4.2 First approximation technique

4.2.1 Mixed approximate-accurate layers (non-uniform structure per layer)

This technique was implemented by replacing the multiplications in various layers with diverse approximate components. More specifically, we approach a **non-uniform structure** per layer in the Neural Network. This means that not all layers have the same level of approximation. Some layers perform their convolution operations using an accurate multiplier whereas others perform their convolution operations using approximate multipliers. The Resnet-8 on which this approximation technique was performed has 7

convolutional layers. In order to fully understand, a more technical approach of this approximation will be given. Furthermore an example will help to fully understand this technique.

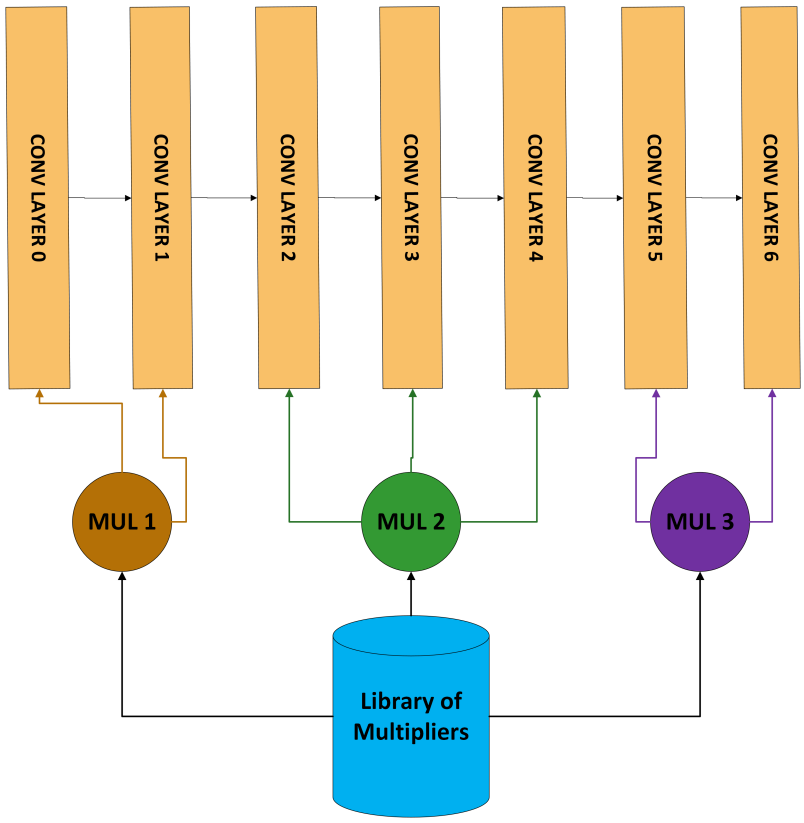


Figure 4.2: First Approximation Technique

4.2.2 Technical point of view

As mentioned, in Chapter 3 the C++ file axconv.cc implements the AxConv2D operation, meaning it executes the approximate multiplications in the convolutional multiplier given the approximate multipliers from the user. Resnet-8 consists of 7 convolutional layers. When the inference python script is executed , the axconv.cc file runs 7 times , for each convolutional layer separately. Knowing this, we created a variable named *conv-layer-counter* which increases each time the axconv.cc is called from the inference script. The value of the conv-layer-counter shows us in which convolutional layer we currently are. For example $counter = 0$ means that the convolutional operations in the **first** convolutional layer are being executed, while $counter = 4$ means that the convolutional operations in the **fourth** convolutional layer are being executed. We exploited the fact that we knew each time on which convolutional layer we currently are while executing the inference and with this way we could decide the level approximation we wanted for our Neural Network.

4.2.3 Example of this approximation technique

Table 4.1: Example with the 4 first layers approximated while the last 3 are have accurate implemantion

Layers	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6	Inference Accuracy
Multiplier	p=1	p=1	p=1	p=1	p=0	p=0	p=0	0.826

In table 4.1 an example of this approximation technique is given. The four first layers of Resnet-8 are approximated and the approximate multiplier is a multiplier [11] with perforation=1 and rounding=0 (p=1 and r=0). The inference accuracy using this approximation technique is 0.826.

The user can choose thanks to this approximation technique:

1. The level of the approximation he/she wants to implement (i.e which convolutional layers he wants to approximate)
2. Up to three different multipliers of his/her choice to approximate the selected convolutional layers

4.3 Second approximation technique

4.3.1 Mixed approximate-accurate filters per layer

This technique was implemented by replacing the multiplications with diverse approximate components.More specifically, each of the seven convolutional layers in the Resnet-8 has a different number of filters as shown is 4.2

Table 4.2: Number of filters per convolutional layer

Layers	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6
Number of filters	16	16	16	32	32	64	64

In this proposed approximation technique, we came up with the idea to split the number of filters in each layer into k equivalent parts, so that the sum of the k parts are equal to the number of filters 5.1 and the difference between the maximum and the minimum number from the sequence is minimized. If $Number\ of\ filters \mod k = 0$ then the minimum difference will always be 0 and the sequence will contain all equal numbers i.e. $Number\ of\ filters \div k$. Else, the difference will be 1 and the sequence will be $Number\ of\ filters \div k$, $Number\ of\ filters \div k$, ...,$(Number\ of\ filters \div k) + 1$, $(Number\ of\ filters \div k) + 1$.

$$number\ of\ filters_1 + number\ of\ filters_2 + \ldots number\ of\ filters_k = Number\ of\ filters \tag{4.1}$$

Each of this parts contains a specific number of filters. We assign in each of these parts a different multiplier [11] with a different perforation setting. In order for this to be better understood, we will demonstrate an example.

It should be noted that each filter kernel produces one output channel.

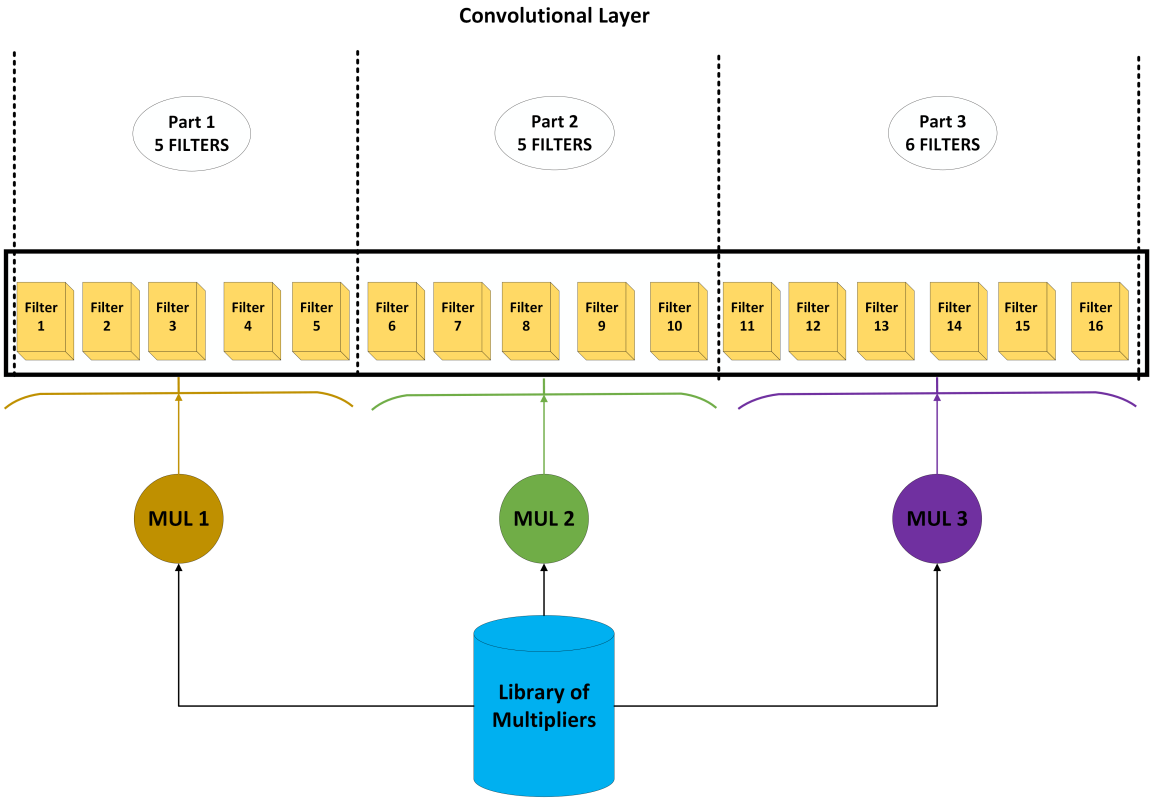


Figure 4.3: Second Approximation Technique

4.3.2 Example of this approximation technique

Layer0 has 16 filters. Let's say we want to split these 16 filters in $k = 3$ equivalent parts. The **first two parts** will contain 5 filters each, while the **third part** will contain 6 filters , so that $5 + 5 + 6 = 16$ filters. We assign in each of these parts a different or the same multiplier. These multiplier are from [11] changing only the perforation parameter. So the first part containing 5 filters will execute the convolution operations with these filters using radix4 multiplier with **perforation** $p = 1$ and the other 2 remaining parts will execute the convolution operations with these filters using radix4 multiplier with **perforation** $p = 2$. The same split is applied to the filters of the other layers and the assign of the multipliers in each part is the same. Let's take *Layer3*. Layer3 has 32 filters. We split these 32 filters as mentioned earlier in $k = 3$ equivalent parts. The **first part** will contain 10 filters, while the two last parts will contain 11 filters each. So the first part containing 11 filters will execute the convolution operations with these filters using radix4 multiplier with **perforation** $p = 1$ and the other 2 remaining parts will execute the

convolution operations with these filters using radix4 multiplier with **perforation** $p = 2$. This approximation technique is applied the same way in each layer.

In table 4.3 an example of this approximation technique is given:

Table 4.3: Mix of approximate filters in each layer with k=3

k=3	Multiplier1	Multiplier2	Multiplier3	Inference Accuracy
	p=1	p=2	p=2	0.798

4.3.3 Technical point of view

As mentioned in Chapter 3, the C++ file axconv.cc implements the AxConv2D operation, meaning it executes the approximate multiplications in the convolutional multiplier given the approximate multipliers from the user while the inference python script executes the inference and searches for all the nodes that execute the newly registered operation AxConv2D. In order the achieve the proposed approximation technique we added a new input argument in the inference python script namely *filter-parameter*. This new argument defines the number of parts in which we want to split the number of filters in each layer. Furthermore, this new argument in added in the C++ file axconv.cc as an attribute of the AxConv2D operation. This new attribute is named k.

4.4 Third approximation technique

In order to to understand better how this approximation technique is performed a brief introduction on how a convolution on 3D data is performed.

4.4.1 Convolution with 3D Data

The filter itself will be 3D. The depth of the filter will be chosen to match the number of color channels and our color image.

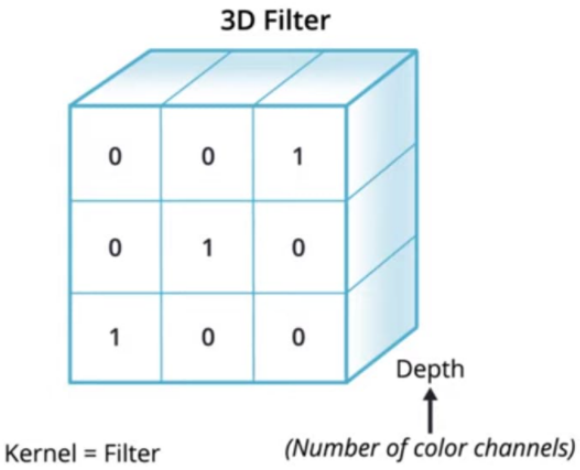


Figure 4.4: 3D Filter.

This is because we're going to convolve each color channel with its own two-dimensional filter. Therefore, if we're working with RGB images, our 3D filter will have a depth of three. 4.5

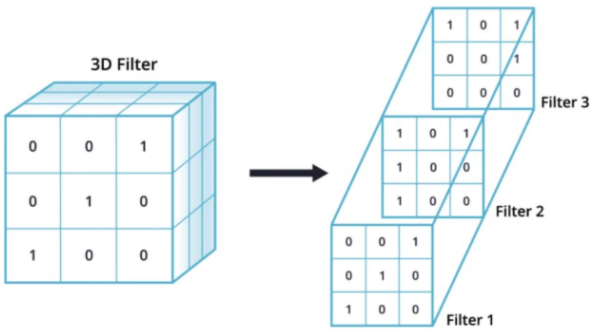


Figure 4.5: Analazying the 3D Filter.

Instead of a 6×6 image, an RGB image could be $6 \times 6 \times 3$ where the 3 here corresponds to the 3 color channels. We can think of this as a stack of three 6×6 images. In order to detect edges or some other feature in this image, we convolve it not with a 3×3 filter, but now with a 3 – dimensional filter. That's gonna be a $3 \times 3 \times 3$, so the filter itself will also have three layers corresponding to red, green and blue channels.

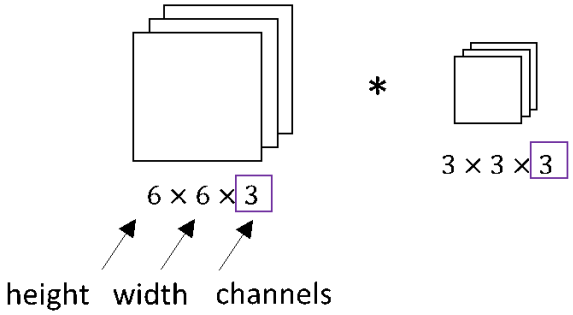


Figure 4.6: The RGB image with the corresponding filter. The 3rd dimension must be the same.

This first 6 here is the height of the image, the second 6 is the width, and the 3 is the number of channels. Similarly, the filter also have a height, width and the number of channels. Number of channels in the image must match the number of channels in our filter, so these two numbers have to be equal. The output of this will be a 4×4 image, and what should be noticed is that this is $4 \times 4 \times 1$, there's no longer 3 at the end.

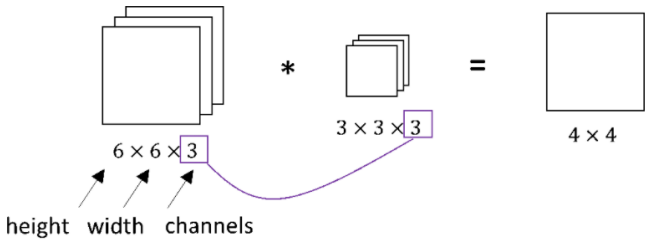


Figure 4.7: Result of a convolution applied on a RGB image

In order to understand this better, we are going to use a a more nicely drawn image.

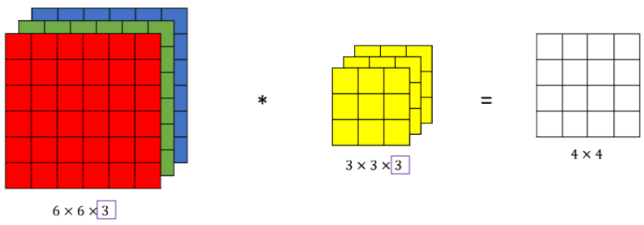


Figure 4.8: RGB image, corresponding filter for convolution and the result of a convolution

Here we can see the $6 \times 6 \times 3$ image and the $3 \times 3 \times 3$ filter. The last number is the number of channels and it matches between the image and the filter. To simplify the drawing the $3 \times 3 \times 3$ filter, we can draw it as a stack of three matrices.

To compute the output of this convolution operation, we take the $3 \times 3 \times 3$ filter and first place it in that most upper left position. Notice that $3 \times 3 \times 3$ filter has 27 numbers. We take each of these 27 numbers and multiply them with the corresponding numbers from the red, green and blue channel. So, take the first nine numbers from red channel, then

the three beneath it for the green channel, then three beneath it from the blue channel and multiply them with the corresponding 27 numbers covered by this $3 \times 3 \times 3$ filter. Then, we add up all those numbers and this gives us the first number in the output. To compute the next output we take this cube and slide it over by one. Again we do the twenty-seven multiplications sum up 27 numbers and that gives us the next output.

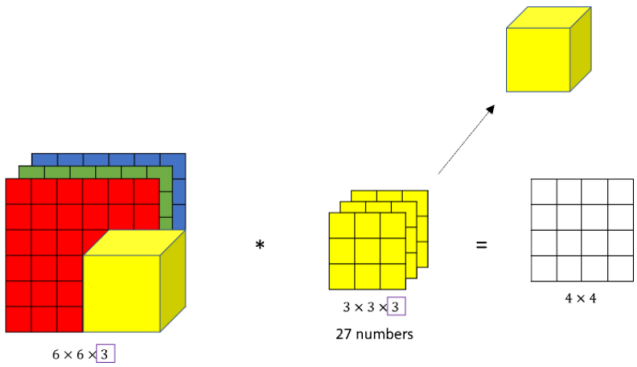


Figure 4.9: When we apply $3 \times 3 \times 3$ filter on the RGB image it is as we implement the volume

So now we have to answer another important question: Why our filter has tree channels and what are the coefficients in that filter? For example, we choose the first filter as $1, 0, -1, 1, 0, -1, 1, 0, -1$. This can be for a red color, for the green channel the values will be all zeros and for the blue filter as well. We stack these three matrices together to form our $3 \times 3 \times 3$ filter. Then, this would be a filter that detects vertical edges, but only in the red channel.

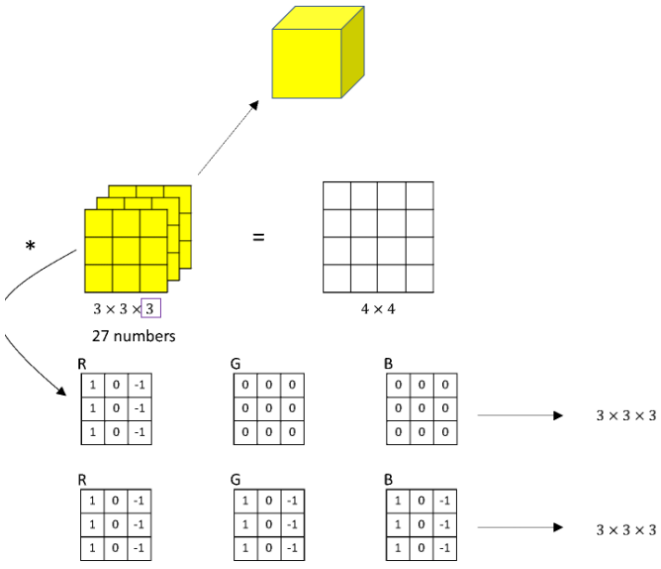


Figure 4.10: Red color edge detector and vertical edge detector for all 3 channels

Alternatively, if it is not important what color the vertical edges are, then we might have a filter with $1s$ and $-1s$ in all three channels. In this way we got a$3 \times 3 \times 3$ edge

detector that detects edges in any color. Different choices of the parameters will result in different feature detectors. By convention, in computer vision when you have an input with a certain height and width, and a number of channels, then your filter can have a different height and width, but number of channels will be the same. Again, the important thing here is that convolving a $6 \times 6 \times 3$volume with a $3 \times 3 \times 3$ gives a 4×4 , a **2D** output.

Another question that must be answered is what if we want to use multiple filters at the same time? We can add a new second filter denoted by orange color, which could be a horizontal edge detector. Convolving an image with the filters gives us different 4×4 outputs. These two 4×4 outputs, can be stacked together obtaining a $4 \times 4 \times 2$ output volume. The volume can be drawn this as a box of a $4 \times 4 \times 2$ volume, where 2 denotes the fact that we used two different filters. 4.11

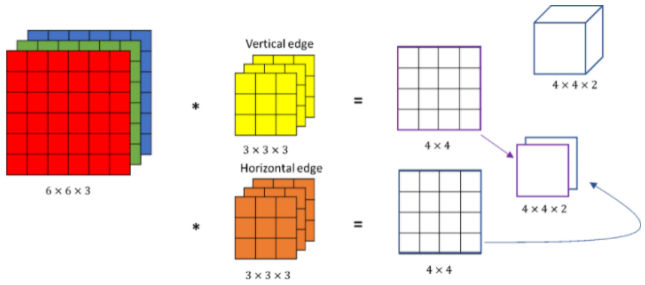


Figure 4.11: When we convolve with two different filters simultaneously

This means that each filter kernel produces one output channel. Another example is illustrated in 4.12

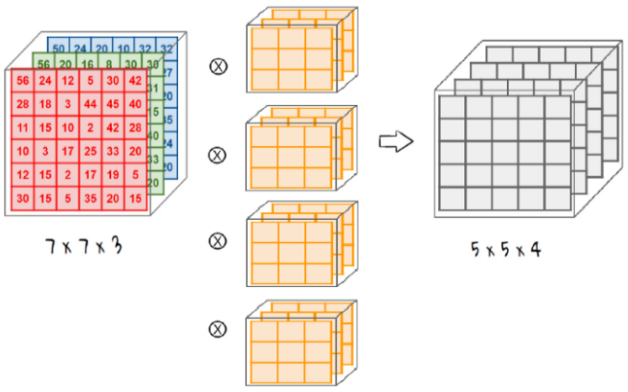


Figure 4.12: When we convolve with two different filters simultaneously

Here we are trying to detect the certain features with 4 filters on a $7 \times 7 \times 3$ volume and thus the convolution computation occurs for each filter. As mentioned earlier, the number of filters determines the depth of the outcome.

4.4.2 A) Approximations per filter via replacement of multiplications with diverse approximate components

This technique was implemented by replacing the multiplications with diverse approximate components. Each input image from the CIFAR-10 dataset has three dimensions: $height, width, depth$. The same thing applies also for filters. Filters also have three dimensions: $filter\ height, filter\ width, depth$. The depth of the filters for each layer is the same as the number of input channels for each layer. Contrary to the previous approximation technique, here we are currently inside the filter and we want to implement approximate multiplications on the convolution operations inside the filter.

The approximate multiplications occur either on $filter\ height$, or $filter\ width$ or $input\ depth$ (which is equal to the number of input channels in each layer). Below a $3 \times 3 \times 3$ filter is depicted as 3 matrices of dimensions 3×3 according to everything that was mentioned above.

Table 4.4: 3×3 filter: channel=1

a_1	a_2	a_3
a_4	a_5	a_6
a_7	a_8	a_9

Table 4.5: 3×3 filter: channel=2

a_{10}	a_{11}	a_{12}
a_{13}	a_{14}	a_{15}
a_{16}	a_{17}	a_{18}

Table 4.6: 3×3 filter: channel=3

a_{19}	a_{20}	a_{21}
a_{22}	a_{23}	a_{24}
a_{25}	a_{26}	a_{27}

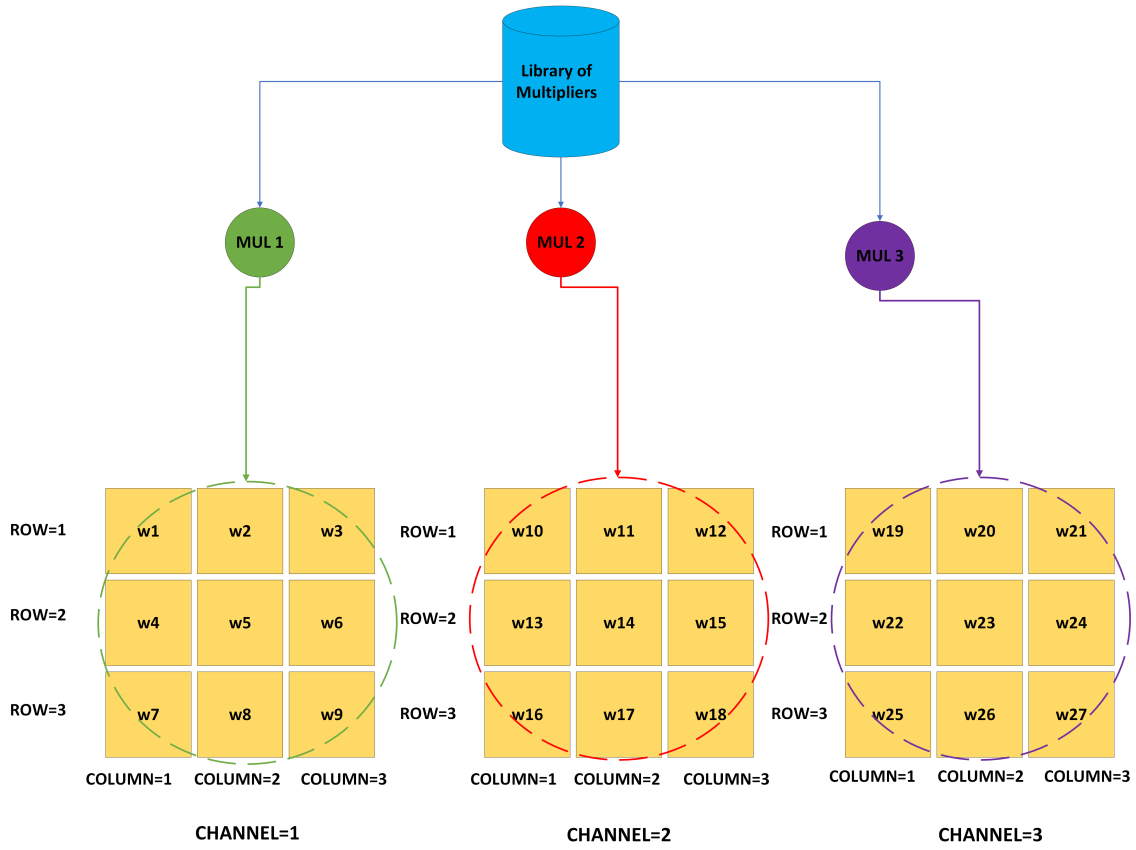


Figure 4.13: Third Approximation Technique: A' (per channel)

For example, let's take the first convolutional layer. The dimensions of the filters in this layer are $3 \times 3 \times 3$. Let's say that the multiplications with weights of $channel = 1$ will be executed with the accurate radix4 mutliplier with perforation $p = 0$, while the multiplications with weights of $channel = 2$ and $channel = 3$) will be executed with the approximate radix4 mutliplier with perforation $p = 2$. According to the 3 tables illustrated above the multiplications:

1. $a_1 \cdot i_1$, $a_2 \cdot i_2$, ... , $a_9 \cdot i_9$ will be executed with the approximate radix4 multiplier with perforation $p = 0$

2. $a_{10} \cdot i_{10}$, $a_{11} \cdot i_{11}$, ..., $a_{18} \cdot i_{18}$ will be executed with the approximate radix4 multiplier with perforation $p = 2$

3. $a_{19} \cdot i_{19}$, $a_{20} \cdot i_{21}$, ..., $a_{27} \cdot i_{27}$ will be executed with the approximate radix4 multiplier with perforation $p = 2$

,where $i_1, i_2, \ldots, i_{27}$ are the input sources value from the current position of the input image.

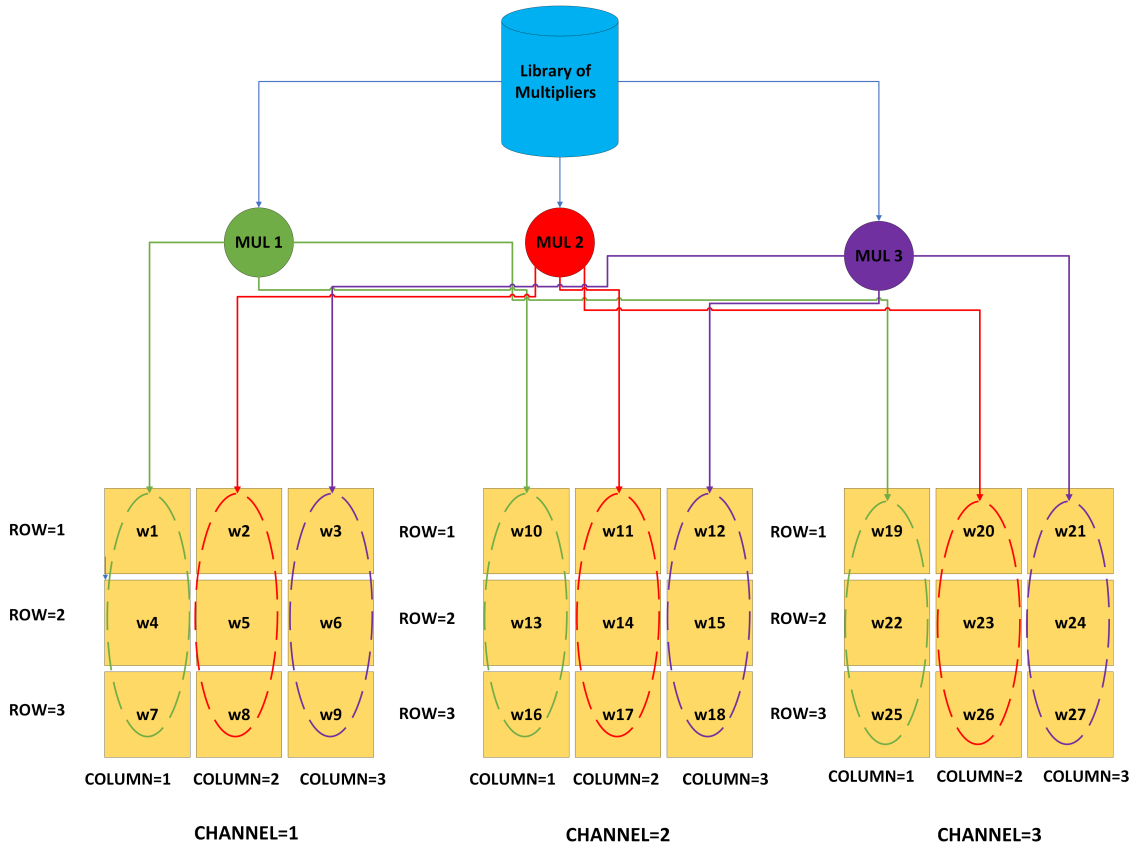


Figure 4.14: Third Approximation Technique: A' (per column)

For example, let's take again the first convolutional layer. The dimensions of the filters in this layer are $3 \times 3 \times 3$. Let's say that the multiplications with weights of $column = 1$) will be executed with the approximate radix4 mutliplier with perforation $p = 0$, while the multiplications with weights of $column = 2$) and $column = 3$) will be executed with the approximate radix4 mutliplier with perforation $p = 2$. According to the 3 tables illustrated above the multiplications:

1. $a_1 \cdot i_1$, $a_4 \cdot i_4$, $a_7 \cdot i_7$, $a_{10} \cdot i_{10}$, $a_{13} \cdot i_{13}$, $a_{16} \cdot i_{16}$, $a_{19} \cdot i_{19}$, $a_{22} \cdot i_{22}$, $a_{25} \cdot i_{25}$will be executed with the approximate radix4 multiplier with perforation $p = 0$

2. $a_2 \cdot i_2$, $a_5 \cdot i_5$, $a_8 \cdot i_8$, $a_{11} \cdot i_{11}$, $a_{14} \cdot i_{14}$, $a_{17} \cdot i_{17}$, $a_{20} \cdot i_{20}$, $a_{23} \cdot i_{23}$, $a_{26} \cdot i_{26}$ will be executed with the approximate radix4 multiplier with perforation $p = 2$

3. $a_3 \cdot i_3$, $a_6 \cdot i_6$, $a_9 \cdot i_9$, $a_{12} \cdot i_{12}$, $a_{15} \cdot i_{15}$, $a_{18} \cdot i_{18}$, $a_{21} \cdot i_{21}$, $a_{24} \cdot i_{24}$, $a_{27} \cdot i_{27}$ will be executed with the approximate radix4 multiplier with perforation $p = 2$

,where $i_1, i_2, \ldots, i_{27}$ are the input sources value from the current position of the input image.

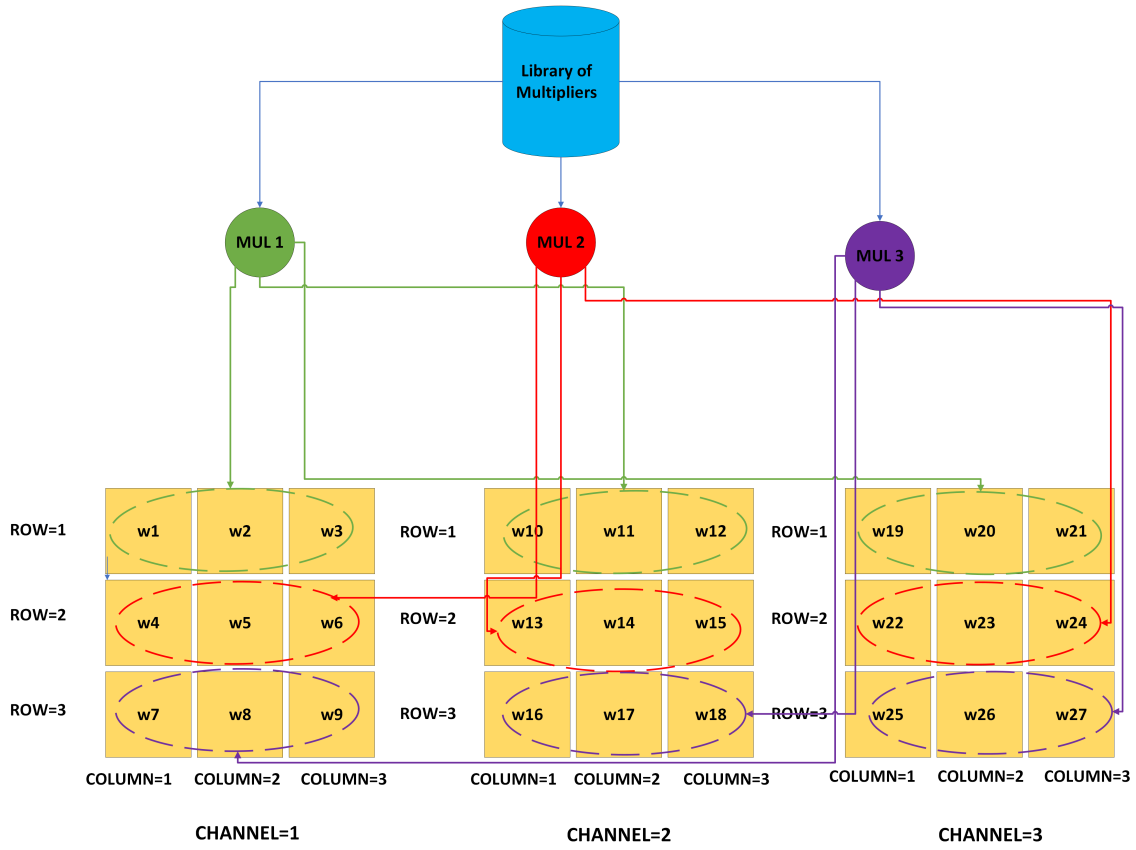


Figure 4.15: Third Approximation Technique: A' (per row)

For example, let's take again the first convolutional layer. The dimensions of the filters in this layer are $3 \times 3 \times 3$. Let's say that the multiplications with weights of $row = 1$ will be executed with the approximate radix4 multiplier with perforation $p = 0$, while the multiplications with weights of $row = 2$) and $row = 3$) will be executed with the approximate radix4 multiplier with perforation $p = 2$. According to the 3 tables illustrated above the multiplications:

1. $a_1 \cdot i_1$, $a_2 \cdot i_2$, $a_3 \cdot i_3$, $a_{10} \cdot i_{10}$, $a_{11} \cdot i_{11}$, $a_{12} \cdot i_{12}$, $a_{19} \cdot i_{19}$, $a_{20} \cdot i_{20}$, $a_{21} \cdot i_{21}$ will be executed with the approximate radix4 multiplier with perforation $p = 0$

2. $a_4 \cdot i_4$, $a_5 \cdot i_5$, $a_6 \cdot i_6$, $a_{13} \cdot i_{13}$, $a_{14} \cdot i_{14}$, $a_{15} \cdot i_{15}$, $a_{22} \cdot i_{22}$, $a_{23} \cdot i_{23}$, $a_{24} \cdot i_{24}$ will be executed with the approximate radix4 multiplier with perforation $p = 2$

3. $a_7 \cdot i_7$, $a_8 \cdot i_8$, $a_9 \cdot i_9$, $a_{16} \cdot i_{16}$, $a_{17} \cdot i_{17}$, $a_{18} \cdot i_{18}$, $a_{25} \cdot i_{25}$, $a_{26} \cdot i_{26}$, $a_{27} \cdot i_{27}$ will be executed with the approximate radix4 multiplier with perforation $p = 2$

,where $i_1, i_2, \ldots, i_{27}$ are the input sources value from the current position of the input image.

4.4.3 B) Approximations per filter via computation skipping

This technique was implemented by skipping (i.e not performing) the multiplications. Each input image from the CIFAR-10 dataset has three dimensions: $height, width, depth$. The same thing applies also for filters. Filters also have three dimensions: $filter\ height, filter\ width, depth$. The depth of the filters for each layer is the same as the number of input channels for each layer. In this approximation technique, instead of executing the multiplications with approximate multipliers , we just don't execute them at all What we actually do in this approximation technique is that we **eliminate** some partial products of our choice. The partial products have been referred above.

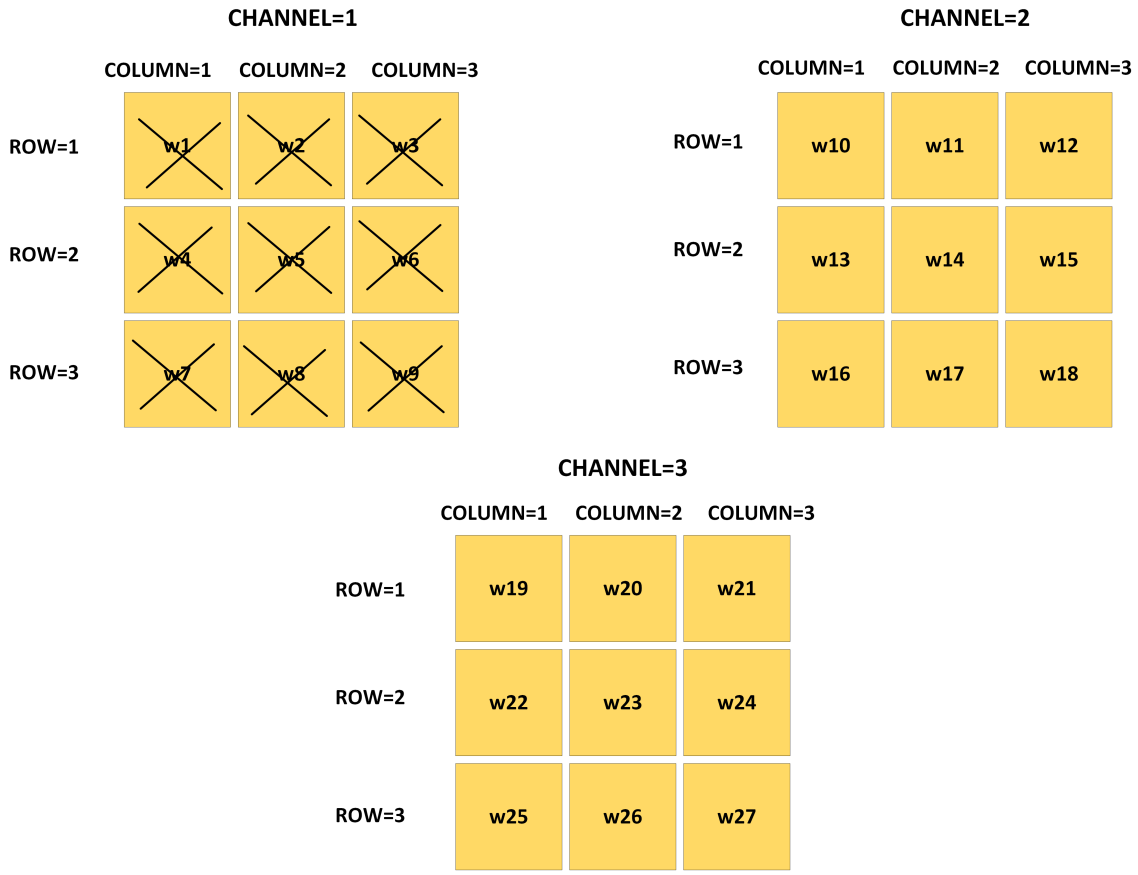


Figure 4.16: Third Approximation Technique: B'(per channel)

For example, let's take the input *input depth* dimension of the filter. We decide to execute only the multiplications with weights of $channel = 1$). This means that we only execute these multiplications: $a_1 \cdot i_1$, $a_2 \cdot i_2$, ... , $a_9 \cdot i_9$, while the rest multiplications $a_{10}, a_{11}, \ldots, a27$ are being skipped (i.e not executed). The same approximation technique can also be applied in the other 2 dimensions. For example, let's take the $filter\ width$ dimension of the filter. We decide to execute only the multiplications with weights of $column = 2$) . This means that we only execute these multiplications: $a_2 \cdot i_2$, $a_5 \cdot i_5$, $a_8 \cdot i_8$, $a_{11} \cdot i_{11}$, $a_{14} \cdot i_{14}$, $a_{17} \cdot i_{17}$, $a_{20} \cdot i_{20}$, $a_{23} \cdot i_{23}$, $a_{26} \cdot i_{26}$, while the rest multiplications

are being skipped (i.e not executed).

4.5 Fourth Approximation Technique

4.5.1 Approximations per kernel via computation skipping based on weight distribution

Each layer has a specific number of filters 4.2 . Every $3 \times 3 \times 3$ filter contains 27 values called weights. Since the weights have been quantized and we use unsigned multiplier the filter weights are in the range $[0, 255]$. Since every layer has its own number of filters, it is obvious that the weights of these filters are different between the layers. In this proposed approximation technique, we printed all the filter weights of each layer and came up with the conclusion that these weights follow a **normal distribution**. Each layer has its own **arithmetic mean** and **standard deviation** of the weights of its filter.

- *Normal distribution*

In probability theory, a normal distribution is a type of continuous probability distribution for a real-valued random variable. The general form of its probability density function is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{1}{2}\frac{x-\mu}{\sigma}^2} \tag{4.2}$$

The parameter μ is the arithmetic mean or expectation of the distribution (and also its median and mode), while the parameter σ is its standard deviation. The variance of the distribution is σ^2. A random variable with a Gaussian distribution is said to be normally distributed, and is called a normal deviate. A normal distribution is sometimes informally called a bell curve. The Standard Deviation is a measure of how spread out numbers are. When we calculate the standard deviation we find that generally:

About 68% of values drawn from a normal distribution are within one standard deviation σ away from the mean; about 95% of the values lie within two standard deviations; and about 99.7% are within three standard deviations

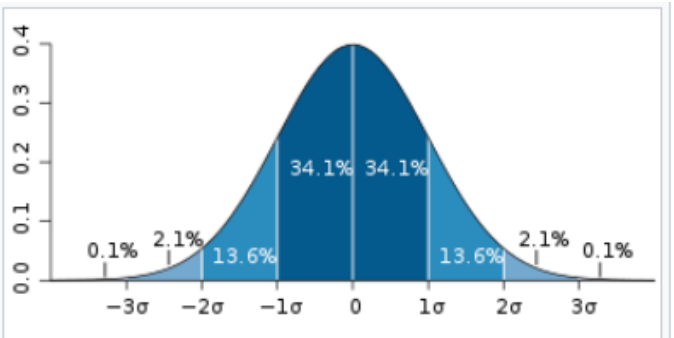


Figure 4.17: Normal distribution

4.5.2 Distribution of weights in each Layer

After we printed all the filter weights of each layer, we calculated the arithmetic mean and standard deviation of the weights of all the filters in each layer. The arithmetic mean μ is described by 5.5 while the standard deviation σ is described by 5.6

$$\mu = \frac{sum\ of\ the\ terms}{number\ of\ terms} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{4.3}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}} \tag{4.4}$$

, where n is the size of the population (number of terms) and x_i is each value from the population.

The table below 4.7 depicts the arithmetic mean and the standard deviation of the weights of all the filters for each layer separately:

Table 4.7: Arithmetic mean and standard deviation of the filters of each layer.

	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6
μ	126.7	133.7	147	154.4	133.1	134.3	115.8
σ	29.2	19.05	25	20.6	27.5	29	23.7

The following figures illustrate the normal distribution that the weights of the filters per layer follow:

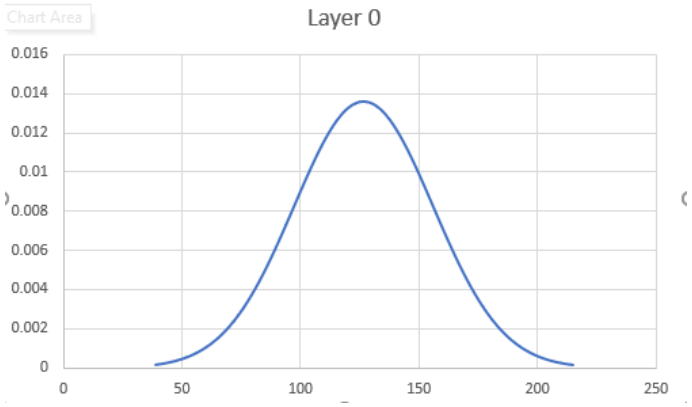


Figure 4.18: Distribution of the weights of the filters in Layer 0

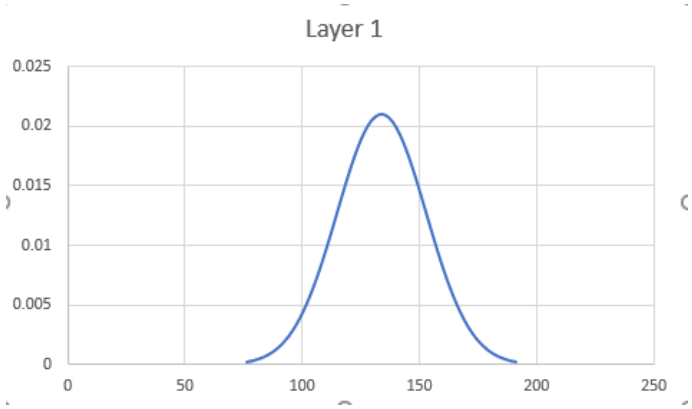


Figure 4.19: Distribution of the weights of the filters in Layer 1

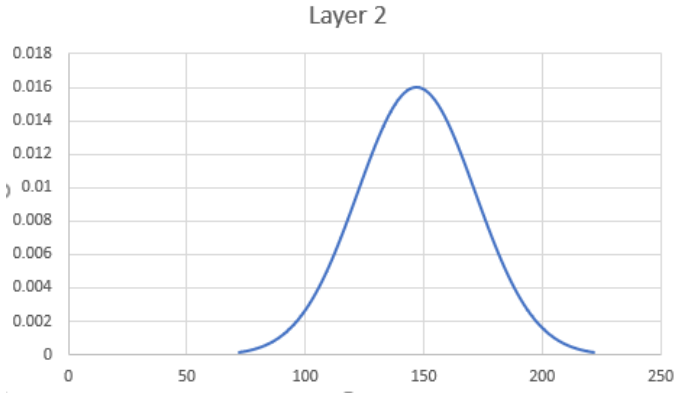


Figure 4.20: Distribution of the weights of the filters in Layer 2

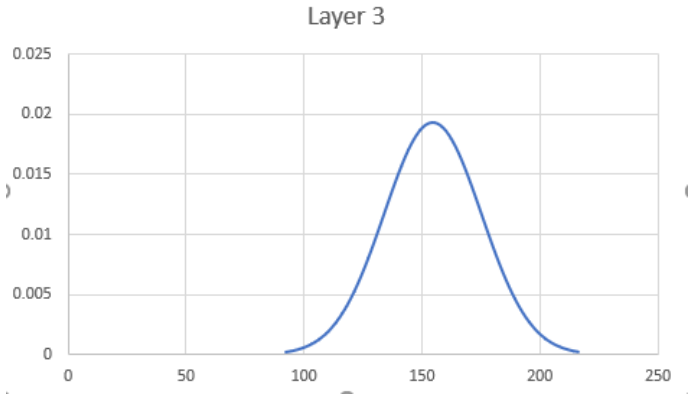


Figure 4.21: Distribution of the weights of the filters in Layer 3

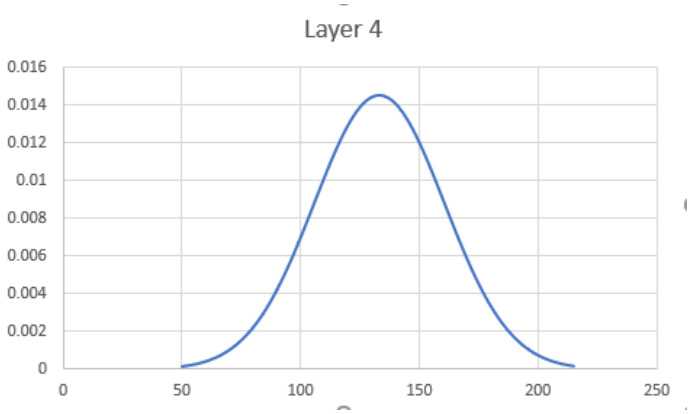


Figure 4.22: Distribution of the weights of the filters in Layer 4

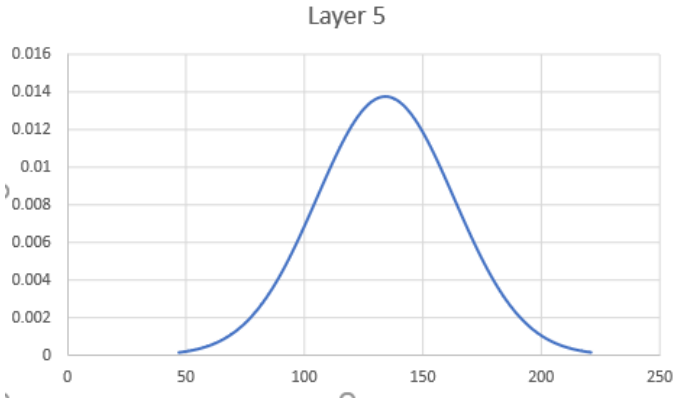


Figure 4.23: Distribution of the weights of the filters in Layer 5

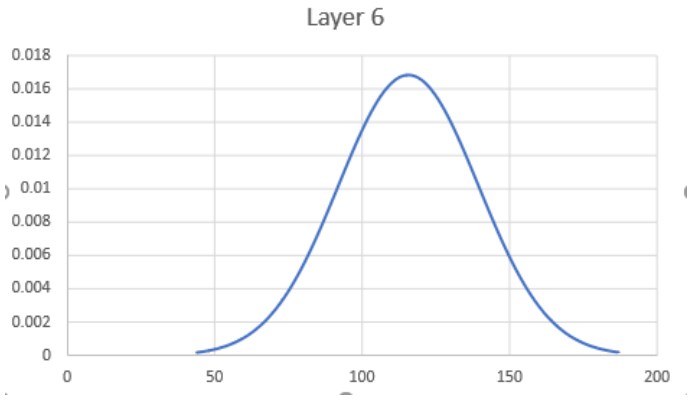


Figure 4.24: Distribution of the weights of the filters in Layer 6

4.5.3 How this technique works

In this approximation technique we propose to execute only the multiplications with the filter weights that belong to either this range $[\mu - \sigma, \mu + \sigma]$ or this $[\mu - 2\sigma, \mu + 2\sigma]$. This means that all the other multiplications with filter weights that do not belong in one

of these ranges are **skipped** (i.e not performed).

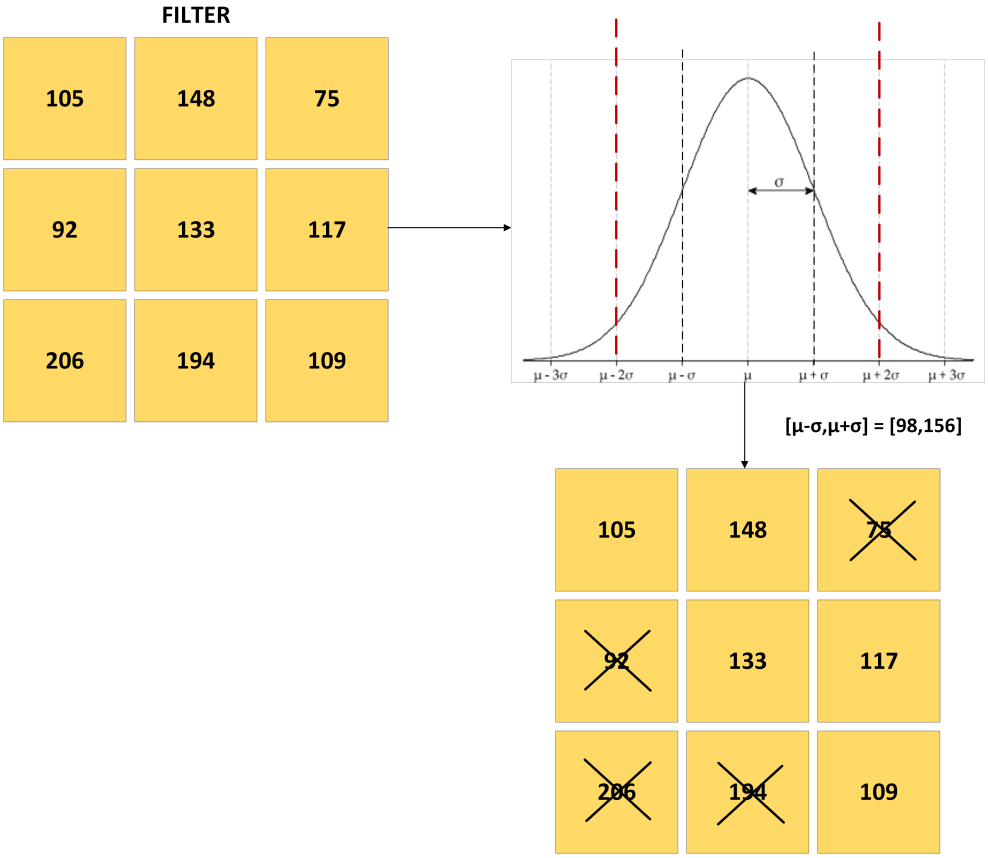


Figure 4.25: Fourth Approximation Technique

Chapter 5

Experimental Evaluation

This chapter's goal is to present the experimental results from were obtained from the multiple different approximations techniques we proposed. Using the library described in chapter 3 we measured how the classification accuracy was affected as well as the energy consumed for the inference of one input image.

5.1 Experimental Setup

All the experiments were conducted in Tensorflow 1.14. The open-source extension[1] of tensorflow described in chapter 3 was used in order to measure the inference accuracy of our experiments for the different parameters. Furthermore the Energy of our approximate multiplier was measured with the help of Synopsys DC using fabrication technology of 45nm [11] was used The input parameters are:

- The graph of Resnet-8, which is given as a protobuf file(.pb extension)
- Evaluation data set of 10000 images from CIFAR-10
- Four different multipliers from [**?**] with different settings in perforation
- $filter\ parameter$ (k) only for the approximation technique described in 4.3.1

The evaluation metrics are:

- Inference accuracy of the model that occurs from each technique using various combinations of approximate multipliers as well as the error which is calculated as $Error = 1 - Accuracy$
- Energy for the inference of one input image , which is calculated as the multiplication of the energy the multiplier used times the number of multiplications which occurred during the inference with that multiplier.

$$Energy = Energy\ of\ multiplier \times \#\ of\ multiplications\ with\ this\ multiplier \quad (5.1)$$

- Throughput (Frames per second) for the inference

For each approximation technique the **Pareto frontier** will be given as well as a subset of the **optimal results** that belong in the Pareto Frontier.

Pareto efficiency or Pareto optimality is a situation where no individual or preference criterion can be better off without making at least one individual or preference criterion worse off or without any loss thereof. The following three concepts are closely related:

1. Given an initial situation, a **Pareto improvement** is a new situation where some agents will gain, and no agents will lose.

2. A situation is called **Pareto dominated** if there exists a possible Pareto improvement.

3. A situation is called **Pareto optimal** or **Pareto efficient** if no change could lead to improved satisfaction for some agent without some other agent losing or if there's no scope for further Pareto improvement.

The **Pareto frontier** is the set of all Pareto efficient allocations, conventionally shown graphically. It also is variously known as the Pareto front or Pareto set.The notion of Pareto efficiency has been used in engineering. The Pareto frontier is a set of non-dominated solutions, being chosen as optimal, if no objective can be improved without sacrificing at least one other objective. On the other hand a solution x* is referred to as dominated by another solution x if, and only if, x is equally good or better than x* with respect to all objectives.

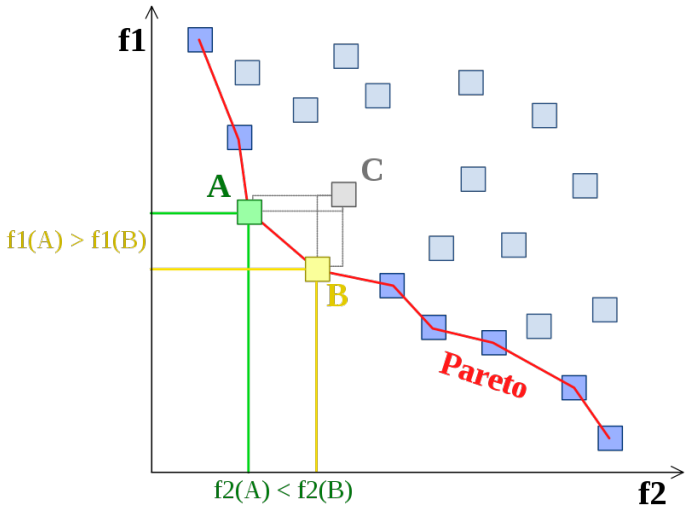


Figure 5.1: Example of Pareto Frontier

In 5.1 the boxed points represent feasible choices, and smaller values are preferred to larger ones. Point C is not on the Pareto frontier because it is strictly dominated by both point A and point B. Points A and B are not strictly dominated by any other, and hence lie on the frontier.

5.2 Fundamental Measurements

In this section we will present the approximate multipliers that were deployed in our experiments. More specifically, we will present the inference accuracy which was measured when only one approximate multiplier was used for all the convolutional operations inside the Renset-8 , as well as the Energy which is consumed for the inference of one input image for each of these multipliers.

The approximate multipliers which are deployed in our experiments are from [**?**]. The models of the approximate multipliers are written in C and are given as input parameter in the python inference script. The only configuration parameter which can be modified is the perforation paramater. Hence we have **perforation=0, perforation=1, perforation=2** and **perforation=3**.

The partial product perforation method dismisses the generation of k successive partial products starting from the least significant ones. This means that the multiplier with configuration parameter of perforation=1 , will have its least significant partial product eliminated, the multiplier with configuration parameter of perforation=2, will have its 2 least significant partial products eliminated e.t.c.

The table below presents the inference accuracy **when only one of these** 4 **multipliers is used for the inference of our model**:

Table 5.1: Inference accuracy of the approximate multiplier $p = 0, p = 1, p = 2, p = 3$

Multiplier	Inference Accuracy
p=0	0.833
p=1	0.815
p=2	0.78
p=3	0.193

In order to calculate the Energy of each of the 4 multipliers, we used the results from [**?**]. We had to to measure the Energy of each of these multipliers in 45nm fabrication technology and in 8-bit width. [**?**] presents the energy of the accurate multiplier (p=0) with 8 bits as well as 16 bits and on both 65nm and 45nm technology.

Table 5.2: Energy of accurate multiplier ($\mu W\dot{n}s$)

	8 bit-width	16 bit-width
65nm	1088.64	3735.91
45nm	384.43	1186.26

In order for us to obtain the energy of the other 3 multipliers we had to find two scaling factors: technology-scaling and bit-scaling.

$$technology\ scaling = \frac{Energy_{65nm,16bit}}{Energy_{45nm,16bit}} \approx 3.15 \tag{5.2}$$

$$bit\ scaling = \frac{Energy_{45nm,16bit}}{Energy_{45nm,8bit}} \approx 3.09 \tag{5.3}$$

Paper [11] presents the energy of the approximate multiplier $p = 1$, $p = 2$, $p = 3$ in 65nm technology and their bit-width is 16. In order for us to obtain the energy of these multiplier in 45nm technology and bit-width of 8 we performed the following operations.

$$temporary = \frac{Energy_{(65nm,16bit)}}{technology\ scaling} \tag{5.4}$$

$$Energy_{(45nm,8bit)} = \frac{temporary}{bit\ scaling} \tag{5.5}$$

Using the above transformation we can obtain the energy of our multipliers in 45nm technology and 8 bit-width.

Table 5.3: Energy of approximate multipliers ($\mu W\dot{n}s$)

	65nm, 16 bit-width	45nm, 8 bit-width
p=0	3748.5	385.725
p=1	2880	296.355
p=2	2472.48	254.421
p=3	2341.68	240.961

Now that we have measured the energy of multipliers in the proper settings , we can measure the energy for the inference of on input image using 5.1. **The number of multiplications for the inference of one input image is 12238848**.

Table 5.4: Energy for the inference of one input image (nJ)

	Energy
p=0	4720.7
p=1	3626.9
p=2	3113.8
p=3	2949.1

5.3 Evaluation of first approximation technique

As mentioned in chapter 4 , in this technique we approach a **non-uniform structure** per layer in the Neural Network. This means that not all layers have the same level of approximation. Some layers perform their convolution operations using an accurate multiplier whereas others perform their convolution operations using approximate multipliers.

The Resnet-8 on which this approximation technique was performed has 7 convolutional layers.

The number of multiplications in each convolutional layer for the inference of one input image is:

Table 5.5: Number of multiplications in each convolutional layer

	Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
Number of multiplications	442368	2359296	2359296	1179648	2359296	1179648	2359296

The tables below depicts the **inference accuracy**, **the error**, **the energy for the inference of one input image** as well as the **throughput (fps)** in three different scenarios:

1. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 1$
2. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 2$
3. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 3$

Table 5.6: P=0 and P=1 deployed

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Inference Accuracy	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	p=0	p=0	p=0	p=0	p=0	0.833	4720.7	18.34	0.167
p=1	p=0	p=0	p=0	p=0	p=0	p=0	0.832	4681.2	18.55	0.168
p=0	p=1	p=0	p=0	p=0	p=0	p=0	0.826	4509.9	19.04	0.174
p=0	p=0	p=1	p=0	p=0	p=0	p=0	0.829	4509.9	19.08	0.171
p=0	p=0	p=0	p=1	p=0	p=0	p=0	0.831	4615.3	19.12	0.169
p=0	p=0	p=0	p=0	p=1	p=0	p=0	0.828	4509.9	19.04	0.172
p=0	p=0	p=0	p=0	p=0	p=1	p=0	0.829	4615.3	19.08	0.171
p=0	p=0	p=0	p=0	p=0	p=0	p=1	0.832	4509.9	19.12	0.168
p=1	p=1	p=0	p=0	p=0	p=0	p=0	0.831	4470.3	19.19	0.169
p=1	p=1	p=1	p=0	p=0	p=0	p=0	0.829	4259.5	19.45	0.171
p=1	p=1	p=1	p=1	p=0	p=0	p=0	0.826	4154.1	19.76	0.174
p=1	p=1	p=1	p=1	p=1	p=0	p=0	0.824	3943.2	20	0.176
p=1	p=1	p=1	p=1	p=1	p=1	p=0	0.821	3837.8	20.16	0.179
p=1	p=1	p=1	p=1	p=1	p=1	p=1	0.82	3626.9	20.32	0.18
p=0	p=0	p=0	p=0	p=0	p=1	p=1	0.831	4404.4	19.26	0.169
p=0	p=0	p=0	p=0	p=1	p=1	p=1	0.83	4193.6	19.49	0.17
p=0	p=0	p=0	p=1	p=1	p=1	p=1	0.827	4088.2	19.72	0.173
p=0	p=0	p=1	p=1	p=1	p=1	p=1	0.822	3877.3	20	0.178
p=0	p=1	p=1	p=1	p=1	p=1	p=1	0.821	3666.5	20.24	0.179

As we can see from 5.6 there are not significant differences in the Inference Accuracy of these combinations when compared to the inference accuracy (**0.833**) which is achieved

when the accurate multiplier ($p = 0$) is employed uniformly. The accuracy loss compared to the accurate multiplier ranges from $0,1\% - 1,3\%$. Furthermore not any significant gains are achieved in the throughput when using the approximate multiplier as described above.

The scatter plot below illustrates the relation between the Energy and Error for the combinations above:

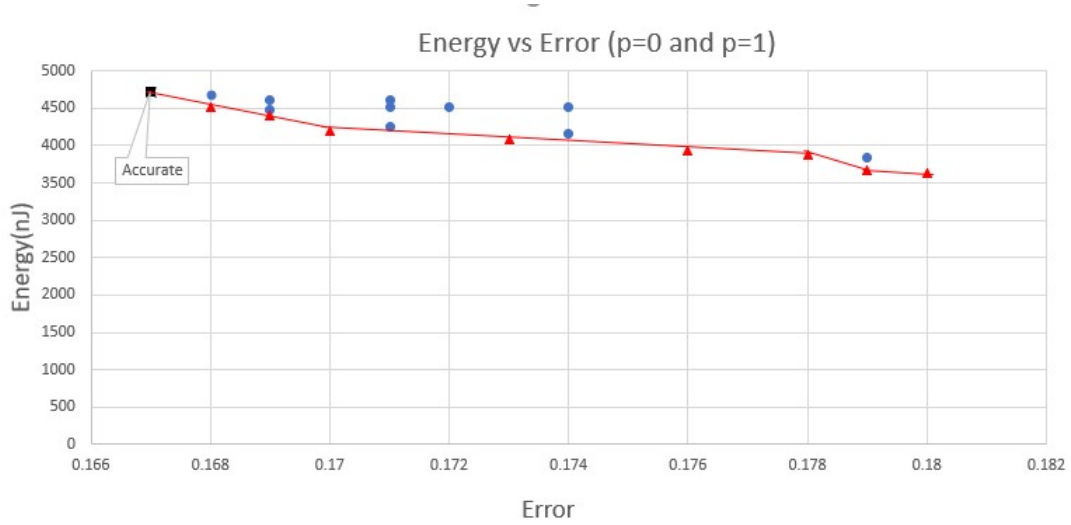


Figure 5.2: First Approximation technique: Energy vs Error when p=0 and p=1 are deployed

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. Those solutions are highlighted in the table 5.6. We can clearly see that most points that belong in the Pareto Frontier occur when the last layers are approximated. This means that it is preferable to employ the approximate multiplier $p = 1$ in the last layers (i.e starting from the Layer6 and going backwards to Layer0).As can be seen from 5.2, employing uniformly the accurate multiplier $p = 0$ provides a point that also belongs in the Pareto Frontier (Error=0.167,Energy=4720.768451 nJ). When comparing the rest points that belong in the Pareto Frontier with the point that is provided by the accurate multiplier we can clearly seen that the accuracy loss is very low, however there is not any significant saving in the Energy required for the inference of one input image.

Table 5.7: P=0 and P=2 deployed

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Inference Accuracy	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	p=0	p=0	p=0	p=0	p=0	0.833	4720.7	18.34	0.167
p=2	p=0	p=0	p=0	p=0	p=0	p=0	0.821	4662.6	18.79	0.179
p=0	p=2	p=0	p=0	p=0	p=0	p=0	0.819	4410.9	19.23	0.181
p=0	p=0	p=2	p=0	p=0	p=0	p=0	0.824	4410.9	19.19	0.176
p=0	p=0	p=0	p=2	p=0	p=0	p=0	0.826	4565.8	19.15	0.174
p=0	p=0	p=0	p=0	p=2	p=0	p=0	0.825	4410.9	19.26	0.175
p=0	p=0	p=0	p=0	p=0	p=2	p=0	0.829	4565.8	19.19	0.171
p=0	p=0	p=0	p=0	p=0	p=0	p=2	0.831	4410.9	19.30	0.169
p=2	p=2	p=0	p=0	p=0	p=0	p=0	0.798	4352.9	19.34	0.202
p=2	p=2	p=2	p=0	p=0	p=0	p=0	0.792	4043.1	19.64	0.208
p=2	p=2	p=2	p=2	p=0	p=0	p=0	0.789	3888.2	20	0.211
p=2	p=2	p=2	p=2	p=2	p=0	p=0	0.787	3578.4	20.28	0.213
p=2	p=2	p=2	p=2	p=2	p=2	p=0	0.786	3423.5	20.45	0.214
p=2	p=2	p=2	p=2	p=2	p=2	p=2	0.78	3113.8	20.66	0.22
p=0	p=0	p=0	p=0	p=0	p=2	p=2	0.828	4256.1	19.41	0.172
p=0	p=0	p=0	p=0	p=2	p=2	p=2	0.818	3946.3	19.72	0.182
p=0	p=0	p=0	p=2	p=2	p=2	p=2	0.811	3791.4	19.96	0.189
p=0	p=0	p=2	p=2	p=2	p=2	p=2	0.789	3481.6	20.36	0.211
p=0	p=2	p=2	p=2	p=2	p=2	p=2	0.786	3171.8	20.53	0.214

As we can see from 5.7 there are not significant differences in the Inference Accuracy of these combinations when compared to the inference accuracy (**0.833**) which is achieved when the accurate multiplier ($p = 0$) is employed uniformly. The accuracy loss compared to the accurate multiplier ranges from $0,2\% - 5,3\%$, which means that there is bigger accuracy loss compared to that caused by employing the $p = 1$ approximate multiplier.

The scatter plot below illustrates the relation between the Energy and Error for the combinations above:

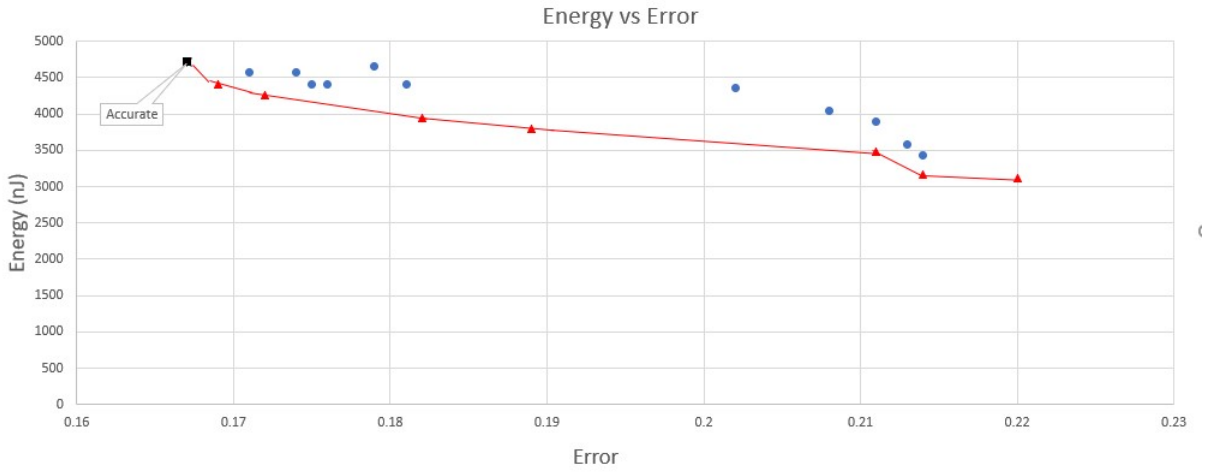


Figure 5.3: First Approximation technique:Energy vs Error when p=0 and p=2 are deployed

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. Those solutions are highlighted in the table 5.7.

Again we can clearly see that almost all of points that belong in the Pareto Frontier occur when the last layers are approximated. This means that it is preferable to employ the approximate multiplier $p = 2$ in the last layers (i.e starting from the Layer6 and going backwards to Layer0).

As can be seen from 5.3, employing uniformly the accurate multiplier $p = 0$ provides a point that also belongs in the Pareto Frontier (Error=0.167,Energy=4720.7 nJ). When comparing the rest points that belong in the Pareto Frontier with the point that is provided by the accurate multiplier we can clearly seen that the accuracy loss is a bit higher now (up to $5,3\%$).However, there are significant savings in the Energy required for the inference of one input image. For example applying $p = 0, p = 0, p = 0, p = 2, p = 2, p = 2, p = 2$ in each layer respectively we achieve an inference accuracy of **0,811** and a total energy for the inference of one input image equal to **3791.4** nJ , which compared to the solution provided by the employment of the accurate multiplier uniformly gives us only a $2,2\%$ in accuracy loss while providing a 929.3 nJ energy saving which is translated in a 19.7% compared to that of the accurate, quite significant given that the loss in accuracy is only $2,2\%$.

Table 5.8: P=0 and P=3 deployed

Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Inference Accuracy	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	p=0	p=0	p=0	p=0	p=0	0.833	4720.7	18.34	0.167
p=3	p=0	p=0	p=0	p=0	p=0	p=0	0.619	4656.7	18.93	0.381
p=0	p=3	p=0	p=0	p=0	p=0	p=0	0.599	4379.236762	19.34	0.401
p=0	p=0	p=3	p=0	p=0	p=0	p=0	0.741	4379.2	19.34	0.259
p=0	p=0	p=0	p=3	p=0	p=0	p=0	0.756	4550	19.30	0.244
p=0	p=0	p=0	p=0	p=3	p=0	p=0	0.756	4379.2	19.41	0.244
p=0	p=0	p=0	p=0	p=0	p=3	p=0	0.7759	4550	19.37	0.2241
p=0	p=0	p=0	p=0	p=0	p=0	p=3	0.76	4379.2	19.45	0.24
p=3	p=3	p=0	p=0	p=0	p=0	p=0	0.31	4315.1	19.49	0.69
p=3	p=3	p=3	p=0	p=0	p=0	p=0	0.255	3973.6	19.92	0.745
p=3	p=3	p=3	p=3	p=0	p=0	p=0	0.231	3802.9	20.24	0.769
p=3	p=3	p=3	p=3	p=3	p=0	p=0	0.204	3461.3	20.61	0.796
p=3	p=3	p=3	p=3	p=3	p=3	p=0	0.196	3290.6	20.79	0.804
p=3	p=3	p=3	p=3	p=3	p=3	p=3	0.193	2949.1	21.05	0.807
p=0	p=0	p=0	p=0	p=0	p=3	p=3	0.68	4208.4	19.56	0.32
p=0	p=0	p=0	p=0	p=3	p=3	p=3	0.585	3866.9	20	0.415
p=0	p=0	p=0	p=3	p=3	p=3	p=3	0.503	3696.1	20.32	0.497
p=0	p=0	p=3	p=3	p=3	p=3	p=3	0.435	3354.6	20.61	0.565
p=0	p=3	p=3	p=3	p=3	p=3	p=3	0.306	3013.1	20.83	0.694

As we can see from 5.8 many significant differences start to occur in the Inference Accuracy of these combinations when compared to the inference accuracy (**0.833**) which

is achieved when the accurate multiplier ($p = 0$) is employed uniformly. The accuracy loss compared to the accurate multiplier ranges from $5,7\% - 64\%$, which means that there is a huge accuracy loss compared to that caused by employing the $p = 0$ approximate multiplier. This is totally expected since the approximate multiplier $p = 3$ has a very big error.

What is truly remarkable is the fact that employing the approximate multiplier $p = 3$ in the latest layers has as a result a much better inference accuracy than employing it in the first layers. For example when $p = 3$ is employed only in the last two layers, the inference accuracy is $0,68\%$ while when it is employed only in the first two layers, the inference accuracy is $0,31\%$. There is a huge difference between this two accuracies. Another example is when $p = 3$ is employed only in the first layer where the inference accuracy is $0,619\%$ and when it is employed only in the last layer , where the inference accuracy is $0,76\%$. So we came up with the conclusion that when using this approximate technique it is better to employ the approximate multiplier in the latest layers rather than the first ones.

The scatter plot below illustrates the relation between the Energy and Error for the combinations above:

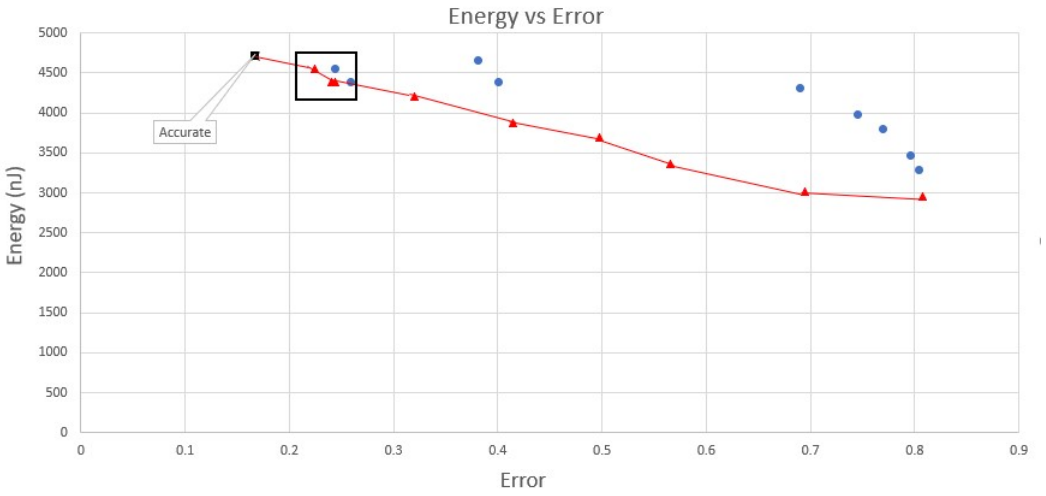


Figure 5.4: First Approximation technique: Energy vs Error when p=0 and p=3 are deployed

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. Those solutions are highlighted in the table 5.8.

Again we can clearly see that almost all of points that belong in the Pareto Frontier occur when the last layers are approximated. This means that it is preferable to employ the approximate multiplier $p = 3$ in the last layers (i.e starting from the Layer6 and going backwards to Layer0).

As can be seen from 5.3, employing uniformly the accurate multiplier $p = 0$ provides a point that also belongs in the Pareto Frontier (Error=0.167,Energy=4720768451 $\mu W \cdot ns$). When comparing the rest points that belong in the Pareto Frontier with the point that

is provided by the accurate multiplier we can clearly seen that the accuracy loss is a bit higher now (up to 64%). However, there are significant savings in the Energy required for the inference of one input image. For example applying $p = 0, p = 0, p = 0, p = 3, p = 3, p = 3, p = 3$ in each layer respectively we achieve an inference accuracy of **0.503** and a total energy for the inference of one input image equal to **3696.1** nJ , which compared to the solution provided by the employment of the accurate multiplier uniformly gives us a huge a 33% in accuracy loss while providing a 1024.5 nJ which is translated in a 21.7% compared to that of the accurate, quite significant. However the loss in the accuracy is very high and cannot be tolerated.

It is fair to say that employing $p = 3$ in more than two layers is not a good solution since the loss in the accuracy loss is really significant while the energy savings do not differ a lot from those achieved when employing the approximate multiplier $p = 2$

5.4 Evaluation of second approximation technique

As mentioned in Chapter 4, this technique was implemented by replacing the multiplications with diverse approximate components. More specifically, we split the number of filters in each layer into k equivalent parts, so that the sum of the k parts are equal to the number of filters. Each of this parts contains a specific number of filters. We assign in each of these parts a different multiplier [11] with a different perforation setting.

- $k = 3$ *parts*

The scatter plot below illustrates the relation between the Energy and Error when we split the number of filters in each layer in $k = 3$ parts and for six different scenarios:

1. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 1$
2. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 2$
3. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 3$
4. For all the possible combinations of the approximate multipliers $p = 1$ and $p = 2$
5. For all the possible combinations of the approximate multipliers $p = 1$ and $p = 3$
6. For all the possible combinations of the approximate multipliers $p = 2$ and $p = 3$

The scatter plot below illustrates the relation between the Energy and Error for the combinations above:

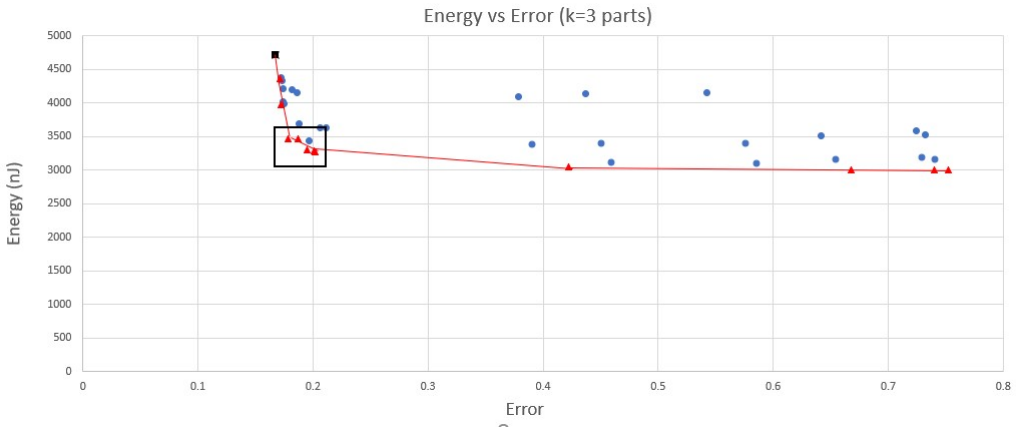


Figure 5.5: Second Approximation technique:Energy vs Error for k=3 parts in each layer

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. These solutions are presented in the table 5.9. The black square contains a subset of the Pareto optimal solutions which we regard as the most optimal solutions of this approximation technique based on two constraints:

1. We want the total energy for the inference of one input image to be at least 1000000000 $\mu W \cdot ns$ less than the energy required for the input image when the accurate multiplier is used (4720768451) , which means the optimal subset consists of solutions with total energy less than 3700.000000 nJ. This is because we only consider solutions with quite a significant energy saving
2. We want the error to be less than 22% since the error introduced using the accurate multiplier only is $16,7\%$

The solution of the optimal subset based on the upon constrains are highlighted in 5.9

Table 5.9: Pareto efficient solutions and their optimal subset for $k = 3$ parts

1^{st} *part*	2^{nd} *part*	3^{rd} *part*	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	p=0	4720.7	18.34	0,167
p=0	p=1	p=0	4364.1	19.49	0.171
p=0	p=1	p=1	3973.7	19.76	0.172
p=2	p=1	p=1	3464.2	20.61	0.178
p=1	p=2	p=1	3459.6	20.40	0.187
p=2	p=2	p=1	3296.9	21.09	0.195
p=2	p=1	p=2	3281.1	21.05	0.201
p=2	p=2	p=3	3055	21.59	0.422
p=2	p=3	p=3	3001.2	21.36	0.668
p=3	p=3	p=2	3007.8	21.83	0.74
p=3	p=2	p=3	3002.7	21.69	0.752

These four solutions of the optimal subset are produced when the approximate multipliers $p = 1$ and $p = 2$ are employed concurrently. They are very good solutions since comparing them with the solutions that occurs when $p = 0$ is employed uniformly, the accuracy loss is very small since it ranges between $1,1\% - 3,4\%$ while there is huge energy saving up to **1439.6** nJ, which translates in achieving up to 30.5% energy saving, quite significant and beneficial given that the drop in the accuracy is that low

The **heatmap** below illustrates how the inference accuracy changes when different approximate multipliers are deployed (for $k = 3$ parts):

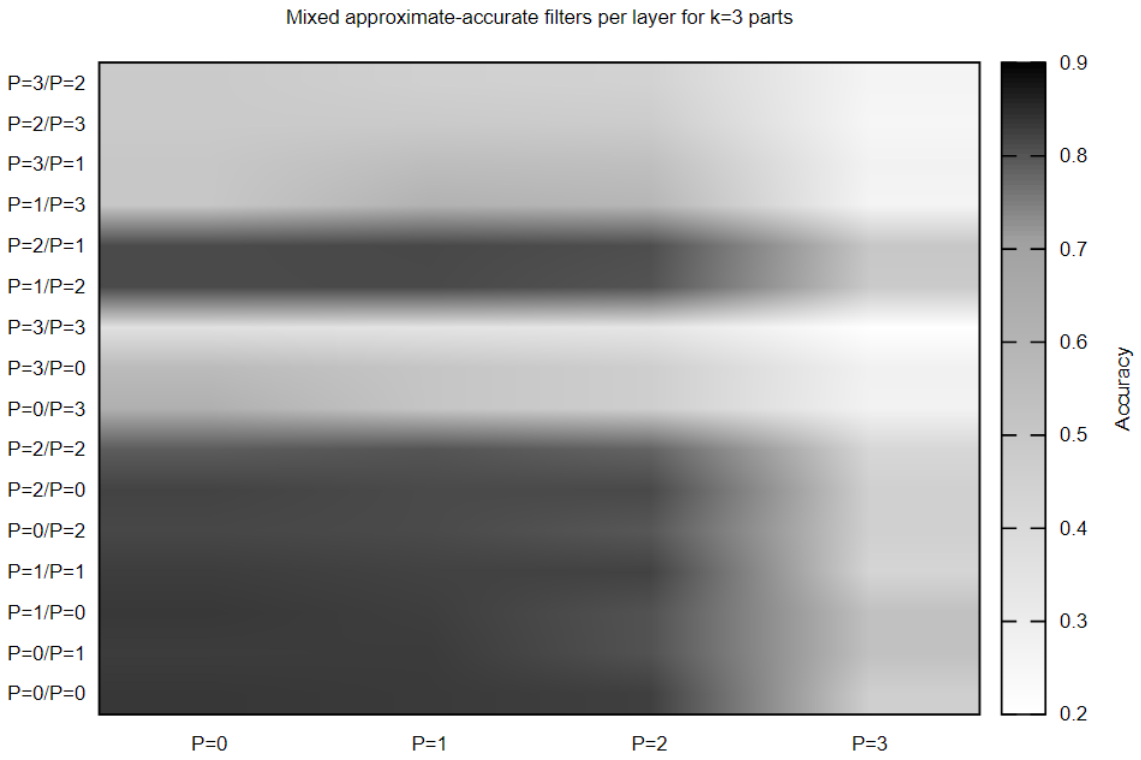


Figure 5.6: Second Approximation technique: Changes in Inference Accuracy for k=3 parts

As it can be clearly seen from the heatmap, the inference accuracy start to drops slightly when we start to employ approximate multipliers in our design. We can observe that using the various combinations of $p = 0$, $p = 1$ and $p = 2$ does not have a huge impact on the accuracy, since the area remains black. However , when we start to employ the approximate multiplier $p = 3$ we can clearly see its effect on the accuracy since it start to drops and this can be clearly seen from the more white areas

- $k = 2$ *parts*

The scatter plot below illustrates the relation between the Energy and Error when we split the number of filters in each layer in $k = 2$ parts and for the same six different scenarios as above:

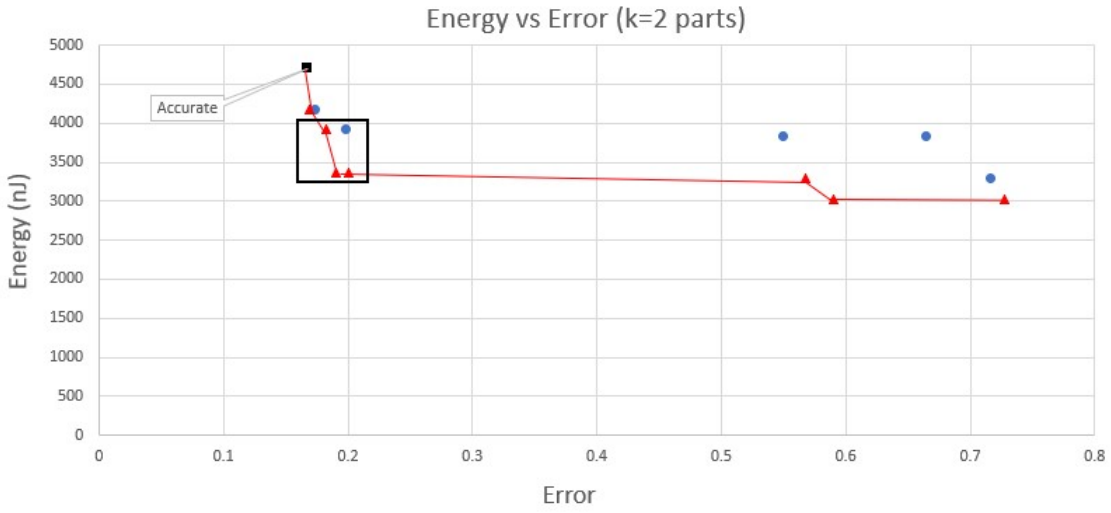


Figure 5.7: Second Approximation technique:Energy vs Error for k=2 parts in each layer

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. These solutions are presented in the table 5.10. The black square contains a subset of the Pareto optimal solutions which we regard as the most optimal solutions of this approximation technique based on two constraints:

1. We want the total energy for the inference of one input image to be at least 1000 nJ less than the energy required for the input image when the accurate multiplier is used (4720.7 nJ) , which means the optimal subset consists of solutions with total energy less than 3700 nJ. This is because we only consider solutions with quite a significant energy saving

2. We want the error to be less than 22% since the error introduced using the accurate multiplier only is $16,7\%$

The solution of the optimal subset based on the upon constrains are highlighted in 5.10

Table 5.10: Pareto efficient solutions and their optimal subset for $k = 2$ parts

1^{st} *part*	2^{nd} *part*	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	4720.7	18.34	0.167
p=1	p=0	4173.8	19.49	0.169
p=2	p=0	3917.2	19.53	0.182
p=2	p=1	3370.3	19.54	0.19
p=1	p=2	3370.3	19.55	0.2
p=1	p=3	3288.0	19.34	0.568
p=2	p=3	3031.4	19.88	0.59
p=3	p=2	3031.4	19.56	0.727

The solutions that belong in the optimal subset of the Pareto Frontier are very good solutions since comparing them with the solution that occurs when $p = 0$ is employed uniformly, the accuracy loss is very small since it ranges between $1,5\% - 3,3\%$ while the is huge energy saving up to **1350.3** nJ, which translates in achieving up to 28.6% energy saving, quite significant and beneficial given that the drop in the accuracy is that low

The **heatmap** below illustrates how the inference accuracy changes when different approximate multipliers are deployed (for $k = 2$ parts):

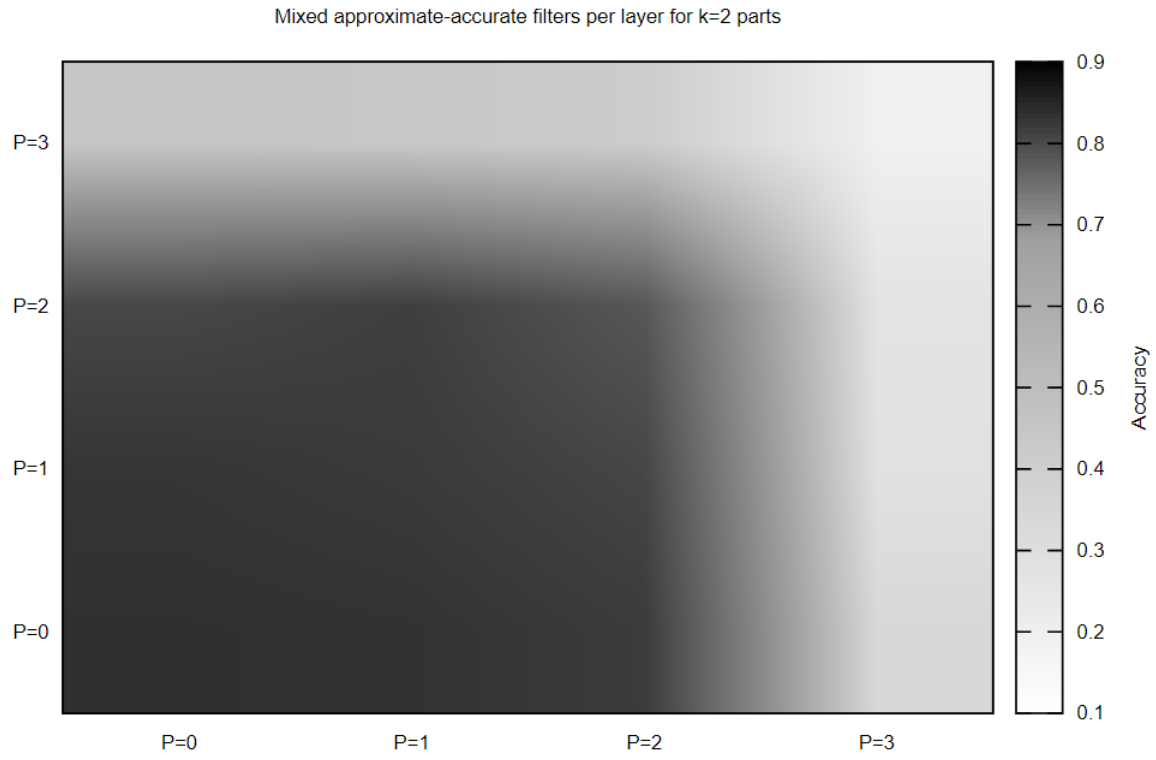


Figure 5.8: Second Approximation technique: Changes in Inference Accuracy for k=2 parts

As it can be clearly seen from the heatmap, the inference accuracy start to drops slightly when we start to employ approximate multipliers in our design. We can observe that using the various combinations of $p = 0$, $p = 1$ and $p = 2$ does not have a huge impact on the accuracy, since the area remains green. However , when we start to employ the approximate multiplier $p = 3$ we can clearly see its effect on the accuracy since it start to drops and this can be clearly seen from the more white areas

5.5 Evaluation of third approximation technique

5.5.1 Approximations per filter via replacement of multiplications with diverse approximate components

As mentioned in Chapter 4, this technique was implemented by replacing the multiplications with diverse approximate components. Contrary to the previous approximation technique, here we are currently inside the filter and we want to implement approximate multiplications on the convolution operations inside the filter.

- **Per channel**

The scatter plot below illustrates the relation between the Energy and Error for six different scenarios at *input depth* level:

1. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 1$
2. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 2$
3. For all the possible combinations of the approximate multipliers $p = 0$ and $p = 3$
4. For all the possible combinations of the approximate multipliers $p = 1$ and $p = 2$
5. For all the possible combinations of the approximate multipliers $p = 1$ and $p = 3$
6. For all the possible combinations of the approximate multipliers $p = 2$ and $p = 3$

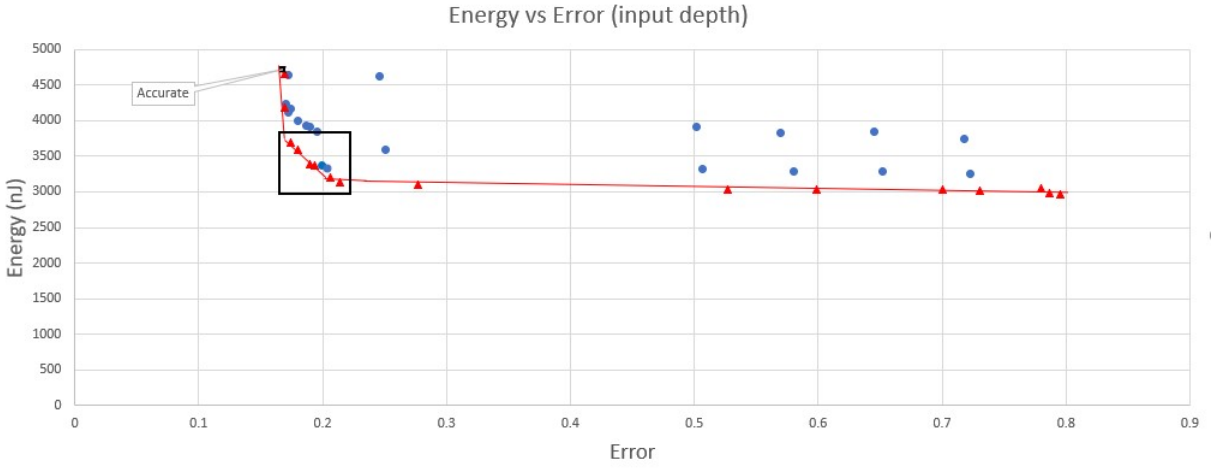


Figure 5.9: Third Approximation technique:Energy vs Error (per channel)

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. These solutions are presented in the table 5.11. The black square contains a subset of the Pareto optimal solutions which we regard as the most optimal solutions of this approximation technique based on two constraints:

1. We want the total energy for the inference of one input image to be at least 1000 nJ less than the energy required for the input image when the accurate multiplier is used (4720768451) , which means the optimal subset consists of solutions with total energy less than 3700 nJ. This is because we only consider solutions with quite a significant energy saving

2. We want the error to be less than 22% since the error introduced using the accurate multiplier only is $16,7\%$

The solution of the optimal subset based on the upon constrains are highlighted in 5.11

Table 5.11: Pareto efficient solutions and their optimal subset (per channel)

$Channel = 1$	$Channel = 2$	$Channel = 3$	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	p=0	4720.7	18.34	0,167
p=0	p=1	p=0	4661.4	19.76	0.168
p=1	p=0	p=0	4180.4	19.88	0.169
p=1	p=0	p=1	3686.2	19.96	0.174
p=1	p=2	p=1	3599.1	20.24	0.18
p=1	p=1	p=2	3395.1	20.24	0.19
p=2	p=1	p=1	3373.4	20.16	0.199
p=2	p=0	p=2	3200.9	20.20	0.206
p=2	p=1	p=2	3141.6	20.36	0.214
p=2	p=3	p=2	3104.8	20.79	0.277
p=2	p=2	p=3	3039.3	20.74	0.527
p=2	p=3	p=3	3030.4	21.05	0.599
p=3	p=2	p=2	3032.4	20.70	0.7
p=3	p=3	p=2	3023.5	21.05	0.73
p=3	p=0	p=3	3045.1	20.70	0.78
p=3	p=1	p=3	2985.8	20.83	0.787
p=3	p=2	p=3	2958.0	21.14	0.795

These six solutions of the optimal subset are produced when the approximate multipliers mainly $p = 1$ and $p = 2$ are employed concurrently. These are very good solutions since comparing them with the solution that occurs when $p = 0$ is employed uniformly, the accuracy loss is very small since it ranges between $0,7\% - 4,7\%$ while the is huge energy saving up to **1579.1** nJ, which translates in achieving up to 33.5% energy saving, quite significant and beneficial given that the drop in the accuracy is that low

The heatmap below illustrates how the inference accuracy changes when different approximate multipliers are deployed at input depth:

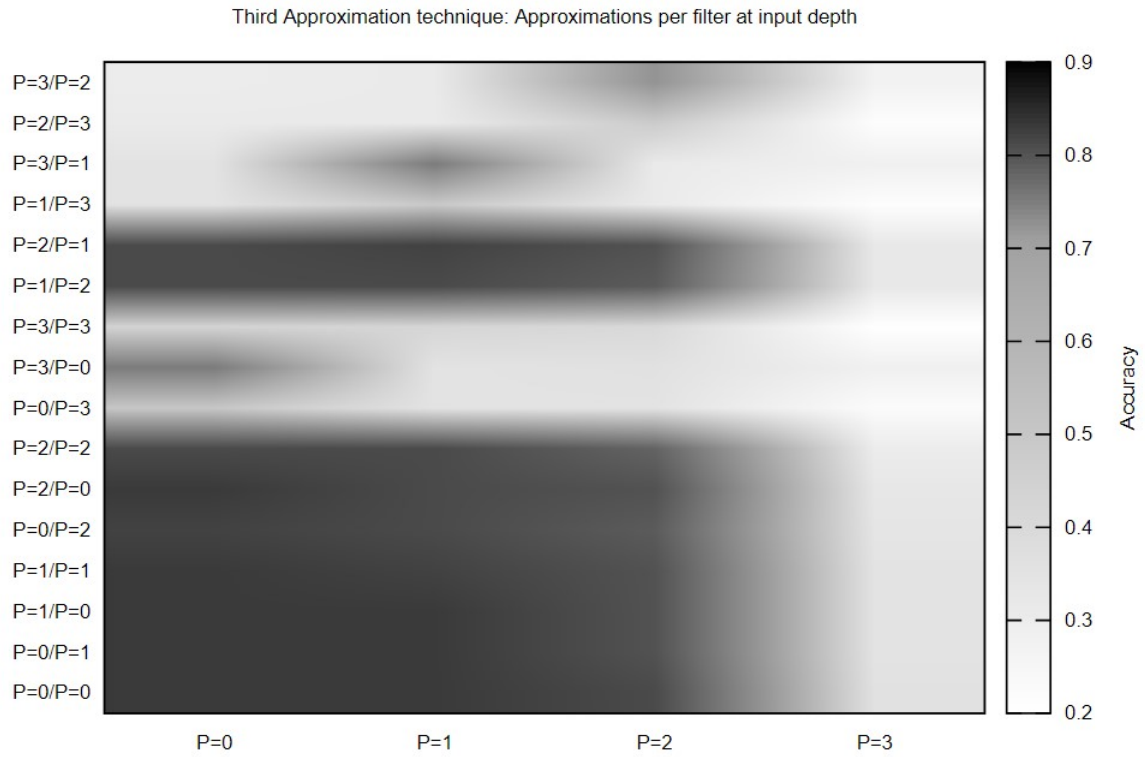


Figure 5.10: Third Approximation technique: Changes in Inference Accuracy (per channel)

As it can be clearly seen from the heatmap, the inference accuracy start to drops slightly when we start to employ approximate multipliers in our design. We can observe that using the various combinations of $p = 0$, $p = 1$ and $p = 2$ does not have a huge impact on the accuracy, since the area remains green. However , when we start to employ the approximate multiplier $p = 3$ we can clearly see its effect on the accuracy since it start to drops and this can be clearly seen from the more white areas

- **Per column**

The scatter plot below illustrates the relation between the Energy and Error for the same six different scenarios as above at *filter width* level:

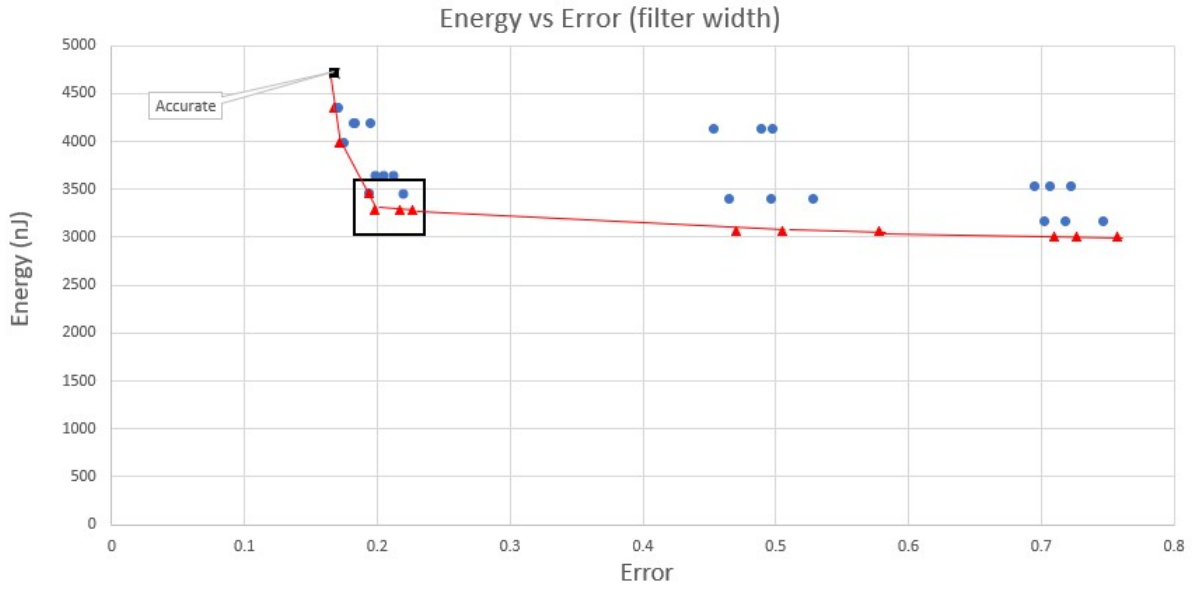


Figure 5.11: Third Approximation technique: Energy vs Error (per column)

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. These solutions are presented in the table 5.12. The black square contains a subset of the Pareto optimal solutions which we regard as the most optimal solutions of this approximation technique based on two constraints:

1. We want the total energy for the inference of one input image to be at least 1000 nJ less than the energy required for the input image when the accurate multiplier is used (4720.7 nJ) , which means the optimal subset consists of solutions with total energy less than 3700 nJ. This is because we only consider solutions with quite a significant energy saving

2. We want the error to be less than 23% since the error introduced using the accurate multiplier only is $16,7\%$

The solution of the optimal subset based on the upon constrains are highlighted in 5.12

Table 5.12: Pareto efficient solutions and their optimal subset at (per column)

$Column = 1$	$Column = 2$	$Column = 3$	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	p=0	4720.7	18.34	0,167
p=0	p=1	p=0	4356.1	19.80	0.168
p=1	p=0	p=1	3991.5	19.96	0.172
p=1	p=1	p=2	3455.9	20.28	0.194
p=2	p=1	p=2	3284.8	20.32	0.198
p=2	p=2	p=1	3284.8	20.40	0.217
p=1	p=2	p=2	3284.8	20.36	0.226
p=3	p=2	p=2	3058.8	20.70	0.47
p=2	p=2	p=3	3058.8	20.83	0.505
p=2	p=3	p=2	3058.8	20.74	0.578
p=3	p=3	p=2	3003.9	20.96	0.71
p=3	p=2	p=3	3003.9	21.09	0.726
p=2	p=3	p=3	3003.9	20.96	0.757

These four solutions of the optimal subset are produced when the approximate multipliers mainly $p = 1$ and $p = 2$ are employed concurrently. These are very good solutions since comparing them with the solution that occurs when $p = 0$ is employed uniformly, the accuracy loss is very small since it ranges between $2,7\% - 5,9\%$ while the is huge energy saving up to **1435.9** nJ, which translates in achieving up to 30.4% energy saving, quite significant and beneficial given that the drop in the accuracy is that low

The heatmap below illustrates how the inference accuracy changes when different approximate multipliers are deployed at input depth:

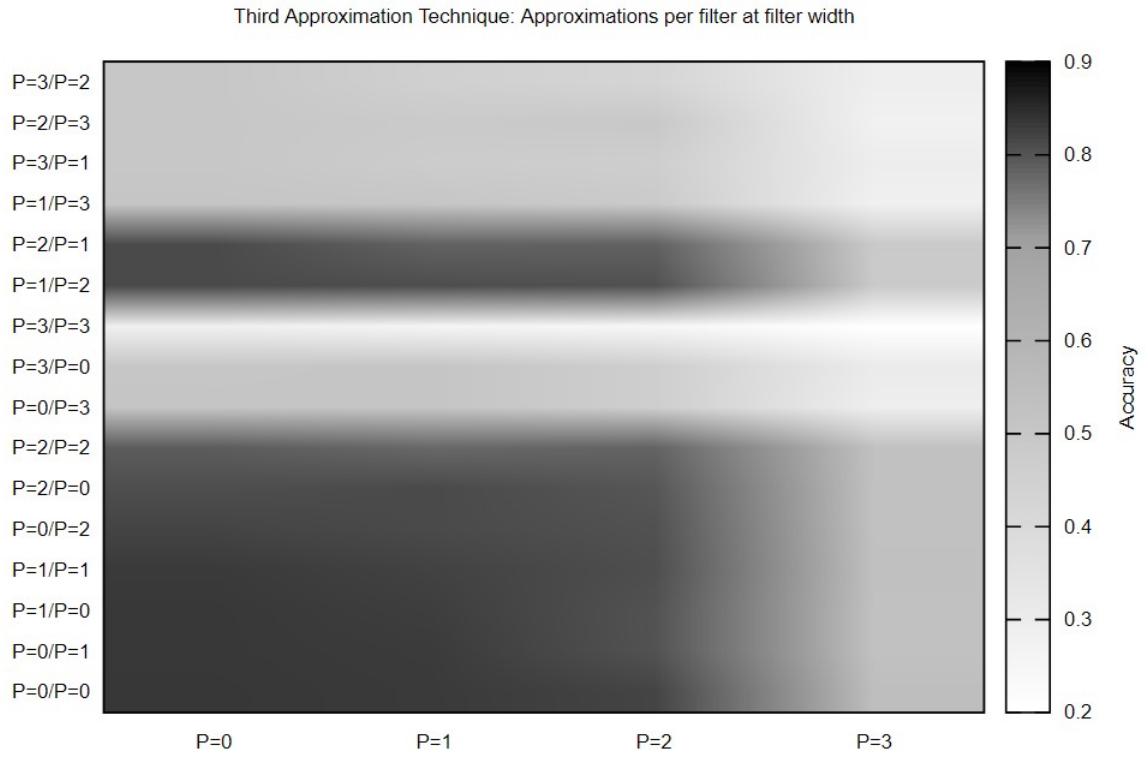


Figure 5.12: Third Approximation technique: Changes in Inference Accuracy (per column)

As it can be clearly seen from the heatmap, the inference accuracy start to drops slightly when we start to employ approximate multipliers in our design. We can observe that using the various combinations of $p = 0$, $p = 1$ and $p = 2$ does not have a huge impact on the accuracy, since the area remains green. However , when we start to employ the approximate multiplier $p = 3$ we can clearly see its effect on the accuracy since it start to drops and this can be clearly seen from the more white areas

- **Per row**

The scatter plot below illustrates the relation between the Energy and Error for the same six different scenarios as above at *filter height* level:

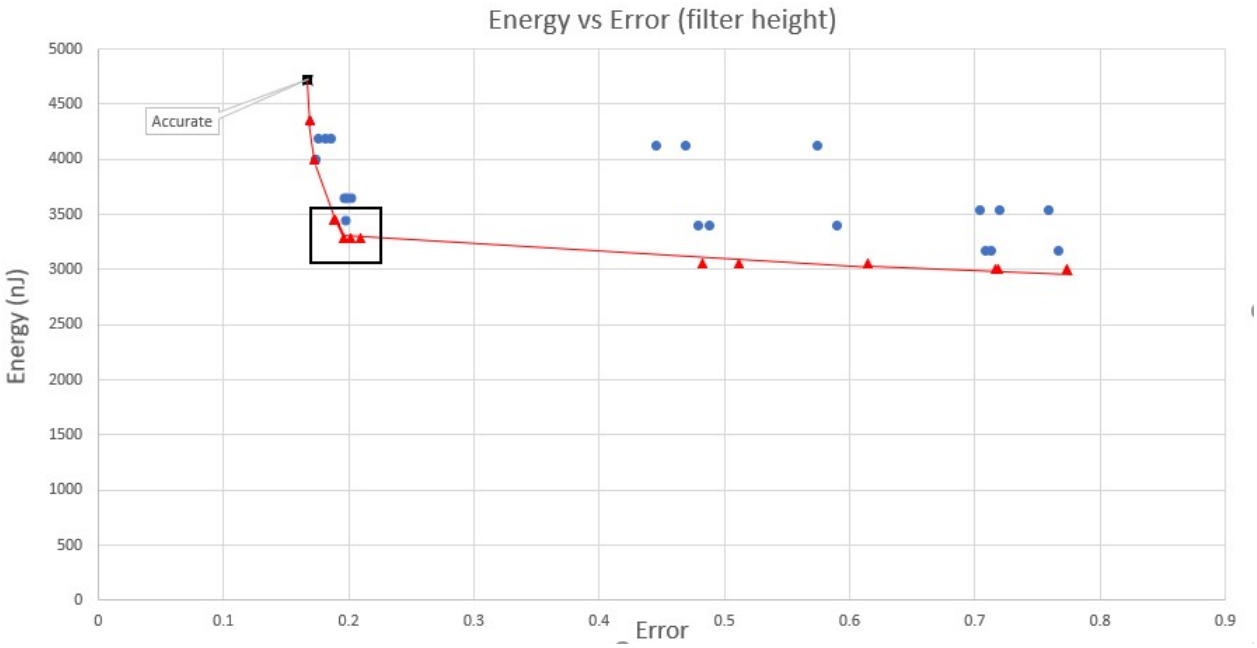


Figure 5.13: Third Approximation technique: Energy vs Error (per row)

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. These solutions are presented in the table 5.13. The black square contains a subset of the Pareto optimal solutions which we regard as the most optimal solutions of this approximation technique based on two constraints:

1. We want the total energy for the inference of one input image to be at least 1000 nJ less than the energy required for the input image when the accurate multiplier is used (4720.7 nJ) , which means the optimal subset consists of solutions with total energy less than 3700 nJ. This is because we only consider solutions with quite a significant energy saving

2. We want the error to be less than 22% since the error introduced using the accurate multiplier only is $16,7\%$

The solution of the optimal subset based on the upon constrains are highlighted in 5.13

Table 5.13: Pareto efficient solutions and their optimal subset (per row)

$Row = 1$	$Row = 2$	$Row = 3$	Energy(nJ)	Throughput(FPS)	Error
p=0	p=0	p=0	4720.7	18.34	0,167
p=1	p=0	p=0	4356.1	19.80	0.169
p=1	p=0	p=1	3991.5	19.92	0.173
p=2	p=1	p=1	3455.9	20.16	0.189
p=2	p=1	p=2	3284.8	20.40	0.196
p=1	p=2	p=2	3284.8	20.32	0.202
p=2	p=2	p=1	3284.8	20.36	0.209
p=2	p=2	p=3	3058.8	20.70	0.482
p=3	p=2	p=2	3058.8	20.70	0.512
p=2	p=3	p=2	3058.8	20.74	0.615
p=2	p=3	p=3	3003.9	21.09	0.716
p=3	p=2	p=3	3003.9	21.09	0.719
p=3	p=3	p=2	3003.9	21.18	0.773

These four solutions of the optimal subset are produced when the approximate multipliers mainly $p = 1$ and $p = 2$ are employed concurrently. These are very good solutions since comparing them with the solution that occurs when $p = 0$ is employed uniformly, the accuracy loss is very small since it ranges between $2,2\% - 4,2\%$ while the is huge energy saving up to **1435.9** nJ, which translates in achieving up to 30.4% energy saving, quite significant and beneficial given that the drop in the accuracy is that low

The heatmap below illustrates how the inference accuracy changes when different approximate multipliers are deployed at input depth:

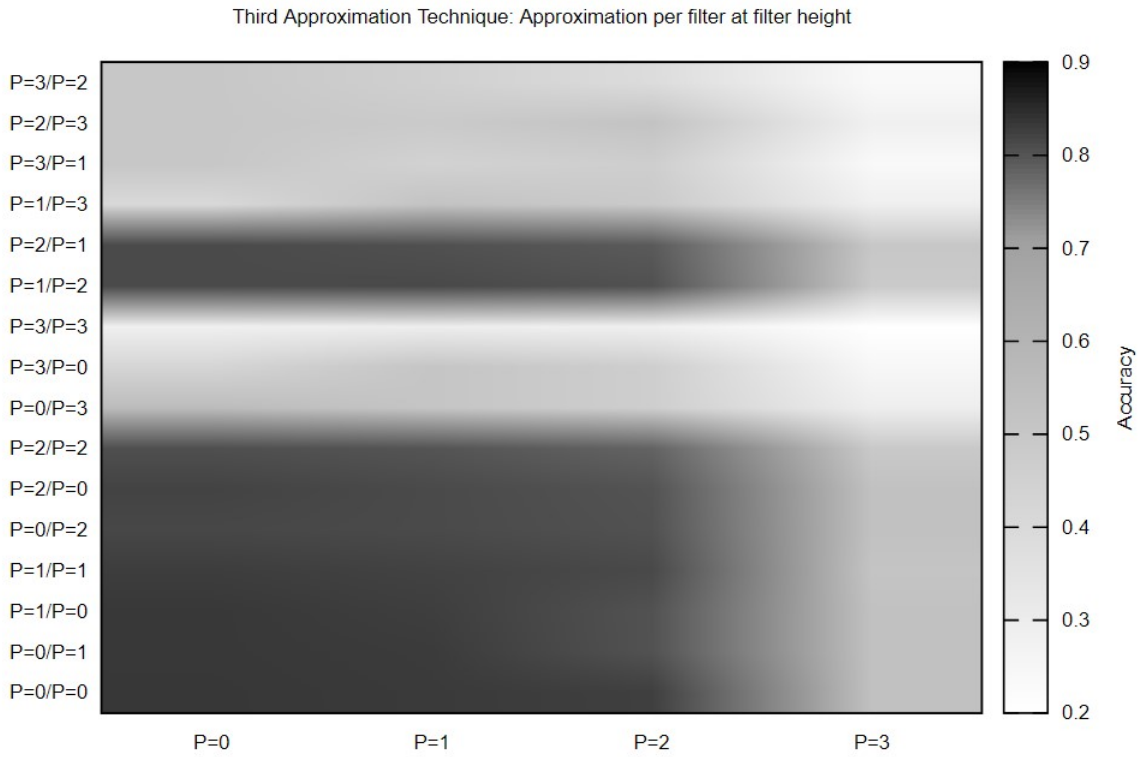


Figure 5.14: Third Approximation technique: Changes in Inference Accuracy (per row)

As it can be clearly seen from the heatmap, the inference accuracy start to drops slightly when we start to employ approximate multipliers in our design. We can observe that using the various combinations of $p = 0$, $p = 1$ and $p = 2$ does not have a huge impact on the accuracy, since the area remains green. However , when we start to employ the approximate multiplier $p = 3$ we can clearly see its effect on the accuracy since it start to drops and this can be clearly seen from the more white areas

5.5.2 Approximations per filter via computation skipping

As mentioned in Chapter 4, this technique was implemented by skipping (i.e not performing) the multiplications. In this approximation technique, instead of executing the multiplications with approximate multipliers , we just don't execute them at all.

- **Per Channel**

The resulting Inference Accuracy is acquired when we execute only the convolution operations that satisfy the condition in the first column of 5.14

Table 5.14: Inference Accuracy when skipping per channel

Execution of channel	Number of multiplications	Inference Accuracy	Energy(nJ)	FPS(Throughput)
$Channel = 1$	6045696	0.115	2331.9	52.63
$Channel = 2$	663552	0.105	255.9	62.5
$Channel = 3$	5529600	0.104	2132.8	54.05
$Channel = 1\ and\ 2$	6709248	0.12	2587.8	50
$Channel = 2\ and\ 3$	6193152	0.138	2388.8	52.63
$Channel = 1\ and\ 3$	11575296	0.507	4464.8	22.72

- **Per column**

The resulting Inference Accuracy is acquired when we execute only the convolution operations that satisfy the condition in the first column of 5.15

Table 5.15: Inference Accuracy when skipping per column

Execution of columns	Number of multiplications	Inference Accuracy	Energy(nJ)	FPS(Throughput)
$Column = 1$	4079616	0.11	1573.5	74.62
$Column = 2$	4079616	0.151	1573.5	81.30
$Column = 3$	4079616	0.082	1573.5	73.52
$Column = 1\ and\ 2$	8159232	0.227	3147.1	42.55
$Column = 2\ and\ 3$	8159232	0.163	3147.1	41.15
$Column = 1\ and\ 3$	8159232	0.205	3147.1	39.84

- **Per row**

The resulting Inference Accuracy is acquired when we execute only the convolution operations that satisfy the condition in the first column of 5.16

Table 5.16: Inference Accuracy when skipping per row

Execution of rows	Number of multiplications	Inference Accuracy	Energy(nJ)	FPS(Throughput)
$Row = 1$	4079616	0.092	1573.5	74.07
$Row = 2$	4079616	0.162	1573.5	54.64
$Row = 3$	4079616	0.098	1573.5	56.81
$Row = 1\ and\ 2$	8159232	0.219	3147.1	33.89
$Row = 2\ and\ 3$	8159232	0.178	3147.1	32.46
$Row = 1\ and\ 3$	8159232	0.127	3147.1	33.00

From the three tables above, we can clearly see that the Accuracy loss compared when we do not skip any operation, is huge. The inference accuracies provided by this approximation technique ranges from 0.082 to 0.227 with the only exception at input depth level where when we only skip the multiplications with weights of $channel = 1$ the inference accuracy is 0.507. We can say that this is expected since the number of multiplications with weights of $channel = 2$ is very low compared to the number of multiplications with weights of $channel = 1$ and $channel = 3$. This means that not a lot of multiplications are being skipped and this is why there is not such a big loss in the inference accuracy. However , the Throughput(FPS) is significantly increased which is something also expected since the reduction of the operations that are executed has as a result the reduction of the time that is needed to run the inference. As a consequence, the Throughput is significantly higher when compared to the other proposed approximation techniques Regardless, of this increase in the Throughput , we can easily conclude that this technique does not provide good results since the accuracy is much lower than the allowed levels. In order to better understand this, we will consider the following example: This approximation technique when used at filter width level and more specifically when we only execute the multiplications with weights of $column = 1$ *and* 2 , we can see from **??** that the **inference accuracy** achieved is **0.227** and the **energy** for the inference of one input image is **3147.1** nJ . We can compare this numbers with the results that occur from the second proposed approximation technique again at filter width level when we use the approximate multiplier $p = 2, p = 1, p = 2$ at $column = 1$,$column = 2$, $column = 3$ respectively and clearly see what computation skipping **is not a good choice**. This combination of approximate multipliers using the second technique has as a result an inference accuracy of **0.802** and the **energy** for the inference of one input image is **3284.8** nJ.

5.6 Evaluation of fourth approximation technique

As mentioned in Chapter 4, this technique was implemented by skipping (i.e not performing) the multiplications. More specifically, our approach is to execute only the multiplications with the filter weights that belong to either this range $[\mu - \sigma, \mu + \sigma]$ or this $[\mu - 2\sigma, \mu + 2\sigma]$, where μ and σ are the average arithmetic mean and standard deviation of the weights of all the filters in each layer separately. This means that all the other multiplications with filter weights that do not belong in one of these ranges are **skipped** (i.e not performed).

The table below depicts the ranges $[\mu - \sigma, \mu + \sigma]$ as well as $[\mu - 2\sigma, \mu + 2\sigma]$ for each layer separately.

Table 5.17: Ranges for each layer

	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6
$[\mu-\sigma, \mu+\sigma]$	[98,156]	[114,153]	[122,172]	[133,175]	[105,161]	[105,163]	[92,140]
$[\mu-2\sigma, \mu+2\sigma]$	[69,185]	[95,172]	[97,197]	[112,196]	[77,189]	[76,192]	[68,164]

Table 5.18: Number of multiplications in each layer for these ranges.

	Layer0	Layer1	Layer2	Layer3	Layer4	Layer5	Layer6	**Total Sum**
$[\mu-\sigma, \mu+\sigma]$	339968	1264640	1703936	829952	1702400	792320	1617472	**8250688**
$[\mu-2\sigma, \mu+2\sigma]$	418816	1783808	2238464	1118720	2241792	1107584	2228928	**11138112**

Executing only the convolution operations when the filter weight belongs in the range $[\mu-\sigma, \mu+\sigma]$ has as a result the execution of a total of **8250688** multiplications for the inference of one input image, while the executing only the convolution operations when the filter weight belongs in the range $[\mu-2\sigma, \mu+2\sigma]$ has as a result the execution of a total of **11138112** multiplications. Those are in contrast with the total **12238848** multiplications that are required for the inference of one input image. This is totally expected since as we mentioned earlier in this proposed approximation technique we skip some computations (i.e multiplications) according to the filter weight distribution of each layer.

1. Executing only the multiplications with filter weights that belong in the range $[\mu-\sigma, \mu+\sigma]$, while skipping the rest, has as a result an **Inference Accuracy** of **0.553** and the **Total Energy** for the inference of one input image is **3182.4** nJ, which translates in achieving up to 32.6% energy saving, quite significant. However, the drop in the inference accuracy is very big and cannot be tolerated

2. Executing only the multiplications with filter weights that belong in the range $[\mu-2\sigma, \mu+2\sigma]$, while skipping the rest, has as a result an **Inference Accuracy** of **0.686** and the **Total Energy**for the inference of one input image is **4296.1** nJ, which translates in achieving up to 9% energy saving

5.7 Approximation Techniques Comparison

Our goal here is to determine in which of the proposed approximation techniques provides the better results regarding the **inference accuracy** and the **energy** required for the inference of one input image. For this purpose, we will compare the subset of the optimal solutions that occurred from each technique and come up with the best solutions. Obviously , we won't compare the results from the third approximation technique using computation skipping as well as from the fourth approximation technique since the the loss in the accuracy is intolerable. Figure **??** illustrates all the optimal subsets that occurred from each technique.

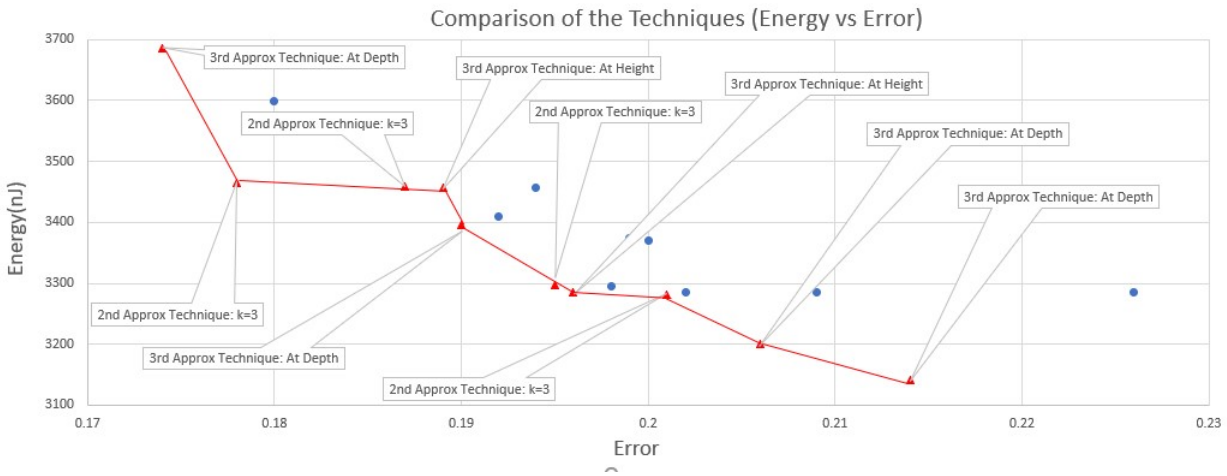


Figure 5.15: Comparison of approximation techniques

The red line is the **Pareto frontier** and all the points that belong to this line consist of the **Pareto optimal** solutions. These solutions are presented in the table 5.19.

Table 5.19: Pareto efficient solutions from the comparison of the techniques

Technique	$0-1$	$1-2$	$2-3$	Energy(nJ)	Error	Energy Saving %
3^{rd}: *Per channel*	p=1	p=0	p=1	3686.2	0.174	21.9%
2^{nd}: $k=3$	p=2	p=1	p=1	3464.2	0.178	26.62%
2^{nd}: $k=3$	p=1	p=2	p=1	3459.6	0.187	26.71%
3^{rd}: *Per row*	p=2	p=1	p=1	3455.9	0.189	26.79%
3^{rd}: *Per channel*	p=1	p=1	p=2	3395.1	0.19	28.08%
2^{nd}:$k=3$	p=2	p=2	p=1	3296.9	0.195	30.16%
3^{rd}: *Per row*	p=2	p=1	p=2	3284.8	0.196	30.41%
2^{nd}: $k=3$	p=2	p=1	p=2	3281.1	0.201	30.5%
3^{rd}: *Per channel*	p=2	p=0	p=2	3200.9	0.206	32.2%
3^{rd}: *Per channel*	p=2	p=1	p=2	3141.6	0.214	33.45%

We can clearly see from the above table that most Pareto Optimal solutions occur when we apply the third approximation technique(i.e Approximations per filter via replacement of multiplications with diverse approximate components) and specifically when we apply this technique at input depth level. However, all the optimal solutions that occurred from the second approximation technique when we divided all the filters in $k = 3$ parts, are included in the Pareto optimal solutions. This means that we can fairly say that the second approximation technique (i.e Mixed approximate-accurate filters per layer via replacement of multiplications with diverse approximate components) is a really good technique that provides great trade-offs between Inference Accuracy and Energy for the inference of one input image.

So overall we come up with the conclusion that the second approximate technique using $k = 3$ *parts* as well as the third approximation technique (via replacement of multiplications with diverse approximate components) at **input depth level** are the **best** approximation techniques

5.8 Comparison with State-of-the-art approximate multipliers

In this section, we will compare the inference accuracy when some of the best 8-bit approximate multipliers (i.e in terms of error) from EvoApproxLib[3] ,are being employed uniformly without any approximation technique with the inference accuracy provided by the same approximate multipliers when they are being employed using the **best** approximation techniques described above. Furthermore, we will compare the energy for the inference of one input image consumed when the same multipliers from EvoApproxLib are being employed uniformly with the energy consumed when our best approximation techniques are implemented, while they employ the multipliers from [11] (i.e approximate radix4 multipliers with different perforation setting).

In the table below, the power(μW) and the delay(ns) some of the best multipliers from EvoApproxLib in terms of error when employed uniformly, as well as the energy of each multiplier which is calculated via 5.6

$$Energy\ of\ multiplier = Power \times Delay \tag{5.6}$$

Table 5.20: Energy of best multipliers from EvoApproxLib [3] (in terms of error)

Multiplier	Power(μW)	Delay(ns)	Energy of Multiplier($\mu W \cdot ns$)
mul8u 2AC	311	1.39	432.29
mul8u 2HH	302	1.44	434.88
mul8u 2P7	386	1.42	548.12
mul8u 14VP	364	1.38	502.32
mul8u 150Q	360	1.39	500.4
mul8u 1446	388	1.35	523.8
mul8u GS2	356	1.38	491.28
mul8u NGR	276	1.37	378.12
mul8u ZFB	304	1.13	343.52

In the next table the inference accuracy, the energy and the error of the above selected multipliers from EvoApproxLib when they are employed uniformly on Resnet-8 without any implementation of our best proposed approximation techniques are presented:

Table 5.21: Inference accuracy,Error and total energy for the inference of one input image of the selected multipliers [3]

Multiplier	Inference Accuracy	Energy(nJ)	Error
mul8u 2AC	0.798	5290.7	0.202
mul8u 2HH	0.767	5322.4	0.233
mul8u 2P7	0.829	6708.3	0.171
mul8u 14VP	0.828	6147.8	0.172
mul8u 150Q	0.83	6124.3	0.17
mul8u GS2	0.826	6012.7	0.174
mul8u NGR	0.77	4627.7	0.23
mul8u ZFB	0.767	4204.2	0.233

Our best solutions provided by our best proposed approximation techniques mentioned in the previous subsection are far better than the solutions provided individually by the approximate multipliers from the EvoApproxLib in terms of error and energy. This can be clearly seen from 5.16

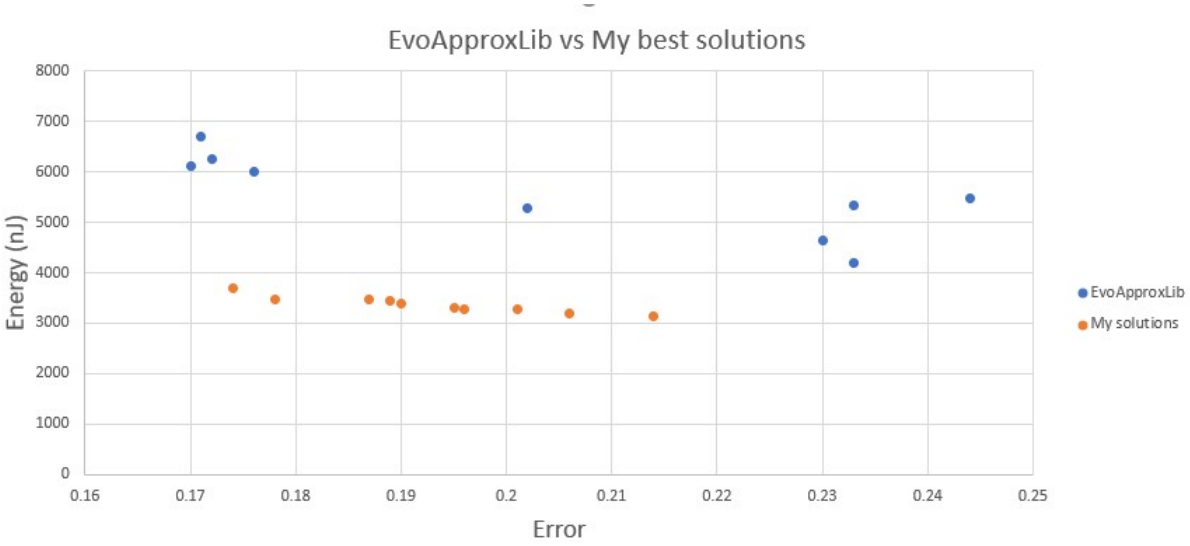


Figure 5.16: Comparison of with state-of-the-art approximate multipliers

We will compare the inference accuracy and energy provided by some of the multipliers mentioned in 5.20 with the inference accuracy and energy provided from the optimal solutions mentioned previously in 5.19.

For example employing uniformly without any approximation technique the approximate multiplier mul8u GS2, the error in the inference accuracy of the network is 17.4% and the energy for the inference of one input image is **6012.7** nJ , while when employing our third proposed approximation technique at input depth level using our multipliers $p = 1$ and $p = 0$ with the following order : $p = 1$, $p = 0$, $p = 1$, we achieve the same

error as above 17.4% with only **3686.2** nJ which means the energy saving is 38.7%, quite significant .

Similarly, when we employ uniformly without any approximation technique the approximate multiplier mul8u 2AC, the error in the inference accuracy is 20.2% and the energy for the inference of one input image is **5290.7** nJ, while when employing our second proposed approximation technique with $k = 3$ parts at each filter using our multipliers with the following order $p = 2$, $p = 1$, $p = 2$, the error in the inference accuracy is 20.1% and the energy for the inference of one input image is **3281.1** nJ , which means that not only we have a lower error but also the energy consumed is significantly lower (38%) this achieving a high energy saving.

These significant energy savings with almost the same error are a result of the use of these specific approximate multiplier with different perforations settings. However, it is also thanks to our proposed approximation techniques that allow the employ of this approximate multipliers in order to achieve these significant energy savings along with almost the same error or in same cases lower error.

Chapter 6

Conclusion

Approximate computing forms a design alternative that exploits the intrinsic error resilience of various applications and produces energy-efficient circuits with small accuracy loss.

In this current thesis we aimed at developing new approximation techniques using approximate multipliers at Deep Neural Network (i.e Resnet-8), that achieve a good trade-off between the inference accuracy of the network and the energy required for the inference of one input image. Our research was carried out at four different levels of the Neural Network in order to come up with the best techniques that combine both high accuracy and low energy. Thus, a library of novel approximation techniques using approximate multipliers in Deep Neural Networks were presented. Our proposed library is an extension of the open source library of Approximate Convolutional Layers in Tensorflow [1]. More specifically, our proposed extended library of approximation techniques is better than the library in [1] , because while this library restricts the users to use only one approximate multiplier, our proposed library provides the users the opportunity to use more than 1 approximate multiplier simultaneously. Furthermore, while the library in [1] provides only one way of approximation (i.e uniform approximation using one approximate multiplier all around the Deep Neural Network), our proposed extended library provides users the opportunity to test their approximate multipliers at different approximation levels using our proposed various approximation techniques.

After developing the new approximation techniques we tested test their efficiency by counting the inference accuracy and the energy required for the inference of one input image that each approximation technique has as a result using various combinations of approximate multipliers. The approximate multipliers we used are from [11] and they are radix4 multipliers with different perforation setting. After testing and comparing they efficiency of each of our proposed approximation techniques, in terms of accuracy and energy trade-off, which was done by finding the pareto optimal solutions of each technique, we came up to the conclusion that the third approximation technique using replacement of multiplications with diverse approximate components as well as the second approximation technique while splitting the number of filters at each layer into $k = 3$ equivalent parts are

the most efficient approximation techniques since they provide the best trade-offs between inference accuracy and energy. More specifically, using the third approximation technique at input depth and an appropriate combination of approximate multipliers with different perforation settings we achieved up to **1579138007** $\mu W \cdot ns$ saving in energy compared to the energy required when the accurate multiplier is employed uniformly which is up to 33.5% saving while having only a slight loss in the inference accuracy ranging between $0.7\% - 4.7\%$. Furthermore, using the second approximation technique while splitting the number of filters at each layer into $k = 3$ equivalent parts and as mentioned before an appropriate combination of approximate multipliers with different perforation settings, we achieved up to **1439637903** $\mu W \cdot ns$ saving in energy compared to the energy required when the accurate multiplier is employed uniformly which is up to 30.4% saving while having only a slight loss in the inference accuracy ranging between $1.1\% - 3.4\%$.

Finally, in a further effort to test the efficiency of our best proposed approximation techniques we compared the inference accuracy when some approximate multipliers from the EvoApproxLib [3] were employed uniformly in the Resnet-8 without any use of our approximation techniques with the inference accuracy that occurred when the same multipliers were employed in the same Neural Network using our best approximation techniques and the results were positive. More particularly, the inference accuracy from the later was up to 0.5% higher than the accuracy from the former thus showing the efficiency of our approximation techniques and the improvement they can achieve.

6.1 Future Work

- **Evaluate the proposed Approximation Techniques on Different Type of Networks**

The approximate techniques we proposed in this thesis can be tested in different types of state of the art neural networks, such as LeNet,Recurrent Neural Networks (RNN) and higher versions of the ResNet (i.e more convolutional layers) in order to evaluate how their behavior is affected by this techniques and also the scalability of them.

- **Evaluate the proposed Approximation Techniques on non-quantized deep neural networks**

If the deep neural network is not quantized, then we can employ approximate multipliers not only with perforation but also with rounding. Thus we can we evaluate the accuracy that occurs from the employment of multipliers with rounding

- **Extend the approximations in more operations inside the Deep Neural Network**

The main purpose of approximate computing, is to insert approximate operations in almost every aspect of the design.There is a huge amount of research in the field of approximate

adders. The basic adder operation inside the MAC operation can be replaced with an appropriate approximate adder that fulfils the design's goals. The main goal is to maintain a high classification accuracy but depending on different engines, adders that satisfy even more criteria must be examined. The selection of the appropriate approximate adder is up to the user and every adder can be used since it does not highly downgrade the classification accuracy. Thus the user can combine the use of appropriate approximate adders with the best of our approximation techniques employing approximate multipliers in order to achieve better energy savings. Furthermore, approximate components that perform the ReLu and Max-Pooling operations can be designed.

- **Design of more energy-efficient approximate multipliers**

It would be very interesting to design and create more energy-efficient approximate multipliers with low error and test these approximate multipliers in our proposed approximation techniques.

Bibliography

[1] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, "Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2019, pp. 1–8.

[2] F. Vaverka, V. Mrazek, Z. Vasicek, and L. Sekanina, "Tfapprox: Towards a fast emulation of dnn approximate hardware accelerators on gpu," *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar 2020. [Online]. Available: http://dx.doi.org/10.23919/DATE48585.2020.9116299

[3] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 258–261.

[4] N. Zhu, W. Goh, and K. S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," 01 2010, pp. 69 – 72.

[5] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *2011 24th Internatioal Conference on VLSI Design*, 2011, pp. 346–351.

[6] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014, pp. 1–4.

[7] H. Jiang, J. Han, F. Qiao, and F. Lombardi, "Approximate radix-8 booth multipliers for low-power and high-performance operation," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2638–2644, 2016.

[8] G. Zervakis, S. Xydis, K. Tsoumanis, D. Soudris, and K. Pekmestzi, "Hybrid approximate multiplier architectures for improved power-accuracy trade-offs," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015, pp. 79–84.

[9] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi, "Design of approximate radix-4 booth multipliers for error-tolerant computing," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1435–1441, 2017.

[10] V. Leon, G. Zervakis, D. Soudris, and K. Pekmestzi, "Approximate hybrid high radix encoding for energy-efficient inexact multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 421–430, 2018.

[11] V. Leon, G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi, "Walking through the energy-error pareto frontier of approximate multipliers," *IEEE Micro*, vol. 38, no. 4, pp. 40–49, Jul./Aug. 2018.

[12] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–9.

[13] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, 03 2016.

[14] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[15] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, "Accelerating CNN inference on fpgas: A survey," *CoRR*, vol. abs/1806.01683, 2018. [Online]. Available: http://arxiv.org/abs/1806.01683

[16] S. Mouselinos, V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi, "Tf2fpga: A framework for projecting and accelerating tensorflow cnns on fpga platforms," in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2019, pp. 1–4.

[17] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[18] V. Leon, S. Mouselinos, K. Koliogeorgi, S. Xydis, D. Soudris, and K. Pekmestzi, "A tensorflow extension framework for optimized generation of hardware cnn inference engines," *Technologies*, vol. 8, p. 6, 01 2020.

[19] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda, "Understanding the impact of precision quantization on the accuracy and energy of neural networks," 2016.

[20] G. Lentaris, G. Chatzitsompanis, V. Leon, K. Pekmestzi, and D. Soudris, "Combining arithmetic approximation techniques for improved cnn circuit design," in *2020 27th*

IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2020, pp. 1–4.

[21] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[23] A. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," *2008 Design, Automation and Test in Europe*, pp. 1250–1255, 2008.

[24] K. Du, P. Varman, and K. Mohanram, "High performance reliable variable latency carry select addition," *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1257–1262, 2012.

[25] H. Jiang, J. Han, and F. Lombardi, "A comparative review and evaluation of approximate adders," 05 2015, pp. 343–348.

[26] T. Yang, T. Ukezono, and T. Sato, "Design of a low-power and small-area approximate multiplier using first the approximate and then the accurate compression method," 05 2019.

[27] M. Masadeh, O. Hasan, and S. Tahar, "Comparative study of approximate multipliers," 2018.

[28] V. Leon, K. Asimakopoulos, S. Xydis, D. Soudris, and K. Pekmestzi, "Cooperative arithmetic-aware approximation techniques for energy-efficient multipliers," in *Design Automation Conference (DAC)*, Jun. 2019, pp. 160:1–160:6.

[29] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan, "Scalable effort hardware design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 2004–2016, 2014.

[30] B. Grigorian, N. Farahpour, and G. Reinman, "Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 615–626.

[31] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: Energy-efficient neuromorphic systems using approximate computing," in *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2014, pp. 27–32.

[32] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate storage for energy efficient spintronic memories," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[33] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 279–290, 2013.

[34] Y. Yetim, M. Martonosi, and S. Malik, "Extracting useful computation from error-prone processors for streaming applications," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 202–207.

[35] V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi, "Energy-efficient VLSI implementation of multipliers with double LSB operands," *IET Circuits, Devices & Systems*, vol. 13, pp. 816–821(5), September 2019.

[36] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, 2004.

[37] J. Huang, J. Lach, and G. Robins, "A methodology for energy-quality tradeoff using imprecise hardware," in *DAC Design Automation Conference 2012*, 2012, pp. 504–509.

[38] V. Leon, T. Paparouni, E. Petrongonas, D. Soudris, and K. Pekmestzi, "Improving power of DSP and CNN hardware accelerators using approximate floating-point multipliers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5, 2021.

[39] P. Yin, C. Wang, W. Liu, and F. Lombardi, "Design and performance evaluation of approximate floating-point multipliers," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016, pp. 296–301.

[40] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5784–5789, 2018.

[41] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016.

[42] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: An approximate computing framework for artificial neural network," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 701–706.

[43] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–7.

[44] S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, "Energy-efficient neural computing with approximate multipliers," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 14, pp. 1–23, 07 2018.

[45] J. Choi and S. Venkataramani, *Approximate Computing Techniques for Deep Neural Networks: Methodologies and CAD*, 01 2019, pp. 307–329.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf