Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Collaborative Filtering Based DNN Partitioning and Offloading on Heterogeneous Edge Computing Systems

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ανδρέα Κοσμά Κακολύρη

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
Αθήνα, Οκτώβριος 2022

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

Collaborative Filtering Based DNN Partitioning and Offloading on Heterogeneous Edge Computing Systems

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ανδρέα Κοσμά Κακολύρη

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31^{η} Οκτωβρίου, 2022.

....	....	....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Χ.Π.Α

Αθήνα, Οκτώβριος 2022

....

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

Copyright © – All rights reserved Ανδρέας Κοσμάς Κακολύρης, 2022.
Με επιφύλαξη παντός δικαιώματος.

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

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

στην οικογένεια μου

Περίληψη

Τα Βαθιά Νευρωνικά Δίκτυα (DNNs) διαδραματίζουν έναν αυξανόμενα σημαντικό ρόλο σε πολλές σύγχρονες εφαρμογές που εντοπίζονται στις παρυφές του δικτύου. Παρά το γεγονός ότι τα DNNs είναι ιδιαίτερα αποτελεσματικά για τα προβλήματα που καλούνται να λύσουν, μπορεί να είναι υπολογιστικά απαιτητικά, συχνά σε απαγορευτικό βαθμό όταν οι περιορισμοί πόρων επεξεργασίας και ενέργειας ληφθούν υπόψιν. Με σκοπό να ξεπεραστούν αυτά τα εμπόδια, η ιδέα του partitioning και του offloading μέρους των DNNs έχει προταθεί ως μία πιθανή λύση. Παρά το γεγονός ότι υπάρχουσες προσεγγίσεις πάνω σε αυτό το πρόβλημα έχουν προτείνει σχήματα διαχείρησης πόρων, η ισχυρά δυναμική φύση του εν λόγω περιβάλλοντος συχνά αγνοείται, τόσο από άποψη ποικιλομορφίας των μοντέλων DNN όσο και από άποψη ετερογένειας του υποβόσκωντος υλισμικού. Σε αυτή την Διπλωματική Εργασία, παρουσιάζουμε ένα πλαίσιο για DNN partitioning και offloading σε συστήματα edge computing. Το πλαίσιο που προτείνουμε, χρησιμοποιεί έναν μηχανισμό Collaborative Filtering βασισμένο σε γνώση που συγκεντρώνεται κατά τη διάρκεια προγενέστερού profiling, προκειμένου να προβεί σε γρήγορες και ακριβείς εκτιμήσεις για την επίδοση (από άποψη χρόνου εκτέλεσης) και την ενεργειακή κατανάλωση της εκτέλεσης στρωμάτων Νευρωνικών Δικτύων πάνω από ένα ποικιλόμορφο σύνολο ετερογενών edge συστημάτων. Μέσω της συγκέντρωσης αυτής της πληροφορίας και την χρήση ενός έξυπνου αλγορίθμου partitioning, η λύση μας παράγει ένα σύνολο Pareto βέλτιστων σχημάτων partitioning, ανταλλάσοντας αυξημένο χρόνο επεξεργασίας για μειωμένη κατανάλωση ενέργειας και το αντίστροφο. Αξιολογούμε την λύση μας για ένα σύνολο ευρέως διαδεδομένων αρχιτεκτονικών DNN με σκοπό να αποδείξουμε οτι η προσέγγιση μας υπερτερεί έναντι σύγχρονων μεθοδολογιών, πετυχαίνοντας κατά μέσο όρο επιτάχυνση της τάξης των 9.58 φορών και μέχρι 88.73% μείωση στην κατανάλωση ενέργειας, προσφέροντας ταυτόχρονα, υψηλή ακρίβεια στις εκτιμήσεις χρόνου υπολογισμού και ενέργειας, περιορίζοντας το σφάλμα πρόβλεψης μέχρι τα επίπεδα του 3.19% και 0.18% αντίστοιχα, λειτουργώντας παράλληλα με έναν ελαφρύ και δυναμικό τρόπο.

Λέξεις Κλειδιά — Cloud, Edge Computing, Διαχείριση Πόρων, Νευρωνικά Δίκτυα, Offloading, Partitioning, Collaborative Filtering

Abstract

Deep Neural Networks (DNNs) are an increasingly important part of many contemporary applications that reside at the edge of the Network. While DNNs are particularly effective at their respective tasks, they can be computationally intensive, often prohibitively so, when the resource and energy constraints of the edge computing environment are taken into account. In order to overcome these obstacles, the idea of partitioning and offloading part of the DNN computations to more powerful servers is often being proposed as a possible solution. While previous approaches have suggested resource management schemes to address this issue, the high dynamicity present in such environments is usually overlooked, both in regards to the variability of the DNN models and to the heterogeneous nature of the underlying hardware. In this thesis, we present a framework for DNN partitioning and offloading for edge computing systems. Our DNN partitioning and offloading framework utilizes a Collaborative Filtering mechanism based on knowledge gathered previously during profiling, in order to make quick and accurate estimates for the performance (latency) and energy consumption of the Neural Network layers over a diverse set of heterogeneous edge devices. Via the aggregation of this information and the utilization of an intelligent partitioning algorithm, our framework generates a set of Pareto optimal Neural Network splittings that trade-off between latency and energy consumption. Our framework is evaluated by using a variety of prominent DNN architectures to show that our approach outperforms current state-of-the-art methodologies by achieving a 9.58× speedup on average and up to 88.73% less energy consumption, simultaneously offering high estimation accuracy by limiting the prediction error down to 3.19% when it comes to latency and 0.18% when energy is concerned, while being lightweight and performing in a dynamic manner.

Keywords — Cloud, Edge Computing, Resource Management, Neural Networks, Offloading, Collaborative Filtering, Partitioning

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή κ. Δημήτριο Σούντρη για την καθοδήγηση και τις ευκαιρίες που μου προσέφερε τα τελευταία χρόνια. Θα ήθελα επίσης να ευχαριστήσω τους υποψήφιους διδάκτορες Δημοσθένη Μασούρο και Εμμανουήλ Κατσαραγάκη για την υποστήριξη και την βοήθεια τους καθώς και τα υπόλοιπα μέλη του Microlab που μου με καλωσόρισαν στην παρέα του εργαστηρίου. Ευχαριστώ επίσης τους φίλους μου σε Αλεξανδρούπολη και Αθήνα για την υπομονή που δείχνουν και την συμπαράστασή τους όλα αυτά τα χρόνια. Τέλος, το πιο μεγάλο ευχαριστώ πηγαίνει στους γονείς μου Στυλιανό και Μαρίνα που είναι πάντα δίπλα μου όλα αυτά τα χρόνια.

Κακολύρης Ανδρέας Κοσμάς
Οκτώβριος 2022

Contents

Περίληψη vii

Abstract ix

Ευχαριστίες xi

Contents xiii

Figure List xv

Table List xvi

Εκτεταμένη Ελληνική Περίληψη 1

1 Εκτεταμένη Ελληνική Περίληψη 1
- 1.1 Εισαγωγή 1
- 1.2 Σχετική Βιβλιογραφία 3
- 1.3 Partitioning και Offloading Νευρωνικών Δικτύων σε Edge Συστήματα 4
  - 1.3.1 Ορισμός Προβλήματος 4
  - 1.3.2 Προτεινόμενη Υλοποίηση 4
  - 1.3.3 Offline Στάδιο 5
  - 1.3.4 DNN profiler 5
  - 1.3.5 Network Profiler 5
  - 1.3.6 Online Στάδιο 6
  - 1.3.7 Predictor 6
  - 1.3.8 Offloader 6
- 1.4 Αξιολόγηση 7
  - 1.4.1 Πειραματική Διάταξη 7
  - 1.4.2 Αποτελέσματα 7
- 1.5 Συμπεράσματα και Μελλοντική δουλειά 11

2 Introduction 13
- 2.1 Contributions 14
- 2.2 Thesis Structure 15

3 Related Work 17

4 Background on DNN architectures and layers, Network protocols, Network profiling and Collaborative Filtering 21
- 4.1 Neural Network Layers 21
  - 4.1.1 Convolutional Layers 21
  - 4.1.2 Normalization Layers 22
  - 4.1.3 Activation Layers 22

4.1.4 Pooling Layers . . . 23
4.1.5 Fully Connected Layers . . . 23
4.2 Neural Network Architectures . . . 24
4.2.1 AlexNet . . . 24
4.2.2 VGG . . . 25
4.2.3 ResNet . . . 26
4.2.4 MobileNetV2 . . . 27
4.2.5 Comments on Residual Architectures . . . 27
4.3 Collaborative Filtering . . . 28
4.3.1 Brief Description . . . 28
4.3.2 Algorithms . . . 28
4.3.3 Matrix Factorization Collaborative Filtering for Predicting Latency and Energy . 30
4.4 Network Communication Protocols . . . 31
4.4.1 ZeroMQ Networking Library . . . 31
4.4.2 FTP Protocol . . . 32
4.5 Network Profiling . . . 33

5 DNN Partitioning and Offloading **35**
5.1 Problem Description . . . 35
5.2 Proposed Methodology for DNN partitioning/offloading . . . 35
5.2.1 Offline Phase . . . 36
5.2.2 Online Phase . . . 38
5.3 Collaborative Filtering Mechanism . . . 38
5.3.1 Offloader . . . 39
5.4 Lifetime of an Inference Request . . . 40
5.4.1 layerLib . . . 40
5.4.2 nnLib . . . 41
5.4.3 offloadingServicesClient . . . 41
5.4.4 offloadingManager . . . 41
5.4.5 offloadingServicesServer . . . 41

6 Experimental Evaluation **43**
6.1 NVIDIA Jetson Family of devices . . . 43
6.1.1 Device Specifications . . . 43
6.2 Experimental Setup . . . 45
6.2.1 Hardware Infrastructure . . . 45
6.2.2 Technical Implementation . . . 45
6.2.3 Examined DNN models . . . 46
6.2.4 Reference Baselines . . . 46
6.3 Evaluation . . . 46
6.3.1 Performance and Energy Evaluation . . . 46
6.3.2 Prediction Accuracy . . . 48
6.3.3 DNN Offload Analysis . . . 49

7 Conclusion and Future Work **51**
7.1 Conclusion . . . 51
7.2 Future Work . . . 51

Bibliography **53**

Figure List

1.1.1 Βέλτιστα σχήματα DNN partitioning ως προς *a)* διαφορετικούς στόχους βελτιστοποίησης, *β)* ετερογένεια των συσκευών. 2
1.3.1 Γενικό σχήμα της λύσης που προτείνουμε. 5
1.4.1 Σύκριση ενέργειας και απόδοσης της προτεινόμενης υλοποίησης ενάντια σε άλλες προσεγγίσεις για CPU και GPU συσκευές, για ένα ένα σύνολο διαφορετικών DNNs. 8
1.4.2 Root Mean Square Error(RMSE) of execution time and energy consumption over alternative Collaborative Filtering fill percentages. 9
1.4.3 Ακρίβεια πρόβλεψης ενέργειας και χρόνου εκτέλεσης για διαφορετικά μοντέλα DNN και edge συσκευές. 10
1.4.4 Ποσοστό offloading για διάφορά DNNs πάνω από ένα ετερογενές σύνολο edge συσκευών. 10

2.0.1 DNN optimal partitioning schemes w.r.t. *a)* different optimization goals and *b)* device heterogeneity. 14

4.2.1 Alexnet Architecture deployed on two GPUs, from the original paper [26] 24
4.2.2 AlexNet Latency and Energy per layer, grouped by device 25
4.2.3 VGG Network Configuration Table, as proposed in the original paper [24] 26
4.2.4 VGG Architecture, figure by Davi Frossard 27
4.2.5 VGG11 Latency and Energy per layer, grouped by device 28
4.2.6 Different Configurations of ResNet architectures, from the original paper [25] 29
4.2.7 ResNet18 Latency and Energy per layer, grouped by device 30
4.2.8 MobileNetV2 Latency and Energy per layer, grouped by device 31
4.2.9 Example of a Residual Block 32
4.5.1 Network performance and energy metrics per device 34

5.2.1 Overview of Online and Offline RoaD RuNNer Architecture. 36
5.3.1 Decomposition of R matrix into $P \times Q$ 39
5.3.2 Offloading Request and Response 40

6.1.1 Devices from the NVIDIA Jetson family 44
6.3.1 Performance and Energy Comparison of RoaD-RuNNer framework against other approaches for CPU and GPU nodes for alternative DNN workloads. 47
6.3.2 Root Mean Square Error(RMSE) of execution time and energy consumption over alternative Collaborative Filtering fill percentages. 48
6.3.3 Execution Latency and Energy Consumption Prediction Accuracy for alternative DNN workloads and Edge nodes. 49
6.3.4 DNN percentage offloading over heterogeneous Edge nodes for latency and energy optimization objectives. 49

Table List

1.1 Τεχνικά χαρακτηριστικά των εξεταζόμενων συσκευών 7

6.1 Technical characteristics of heterogeneous Edge nodes and Cloud Server [35] 45
6.2 Configurations by Power Mode table for all devices, highlighted entries are the ones utilized during the experiment [36] 45

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Τα τελευταία χρόνια οι εφαρμογές που χρησιμοποιούν προηγμένες τεχνικές Μηχανικής Μάθησης (ML) προκειμένου να εξάγουν συμπεράσματα από μεγάλους όγκους δεδομένων γνωρίζουν μεγάλη άνθηση, με αυτή την τάση να αναμένεται μόνο να αυξηθεί στο μέλλον. Σε αυτό το πλαίσιο, τα Βαθιά Νευρωνικά Δίκτυα (DNNs) έχουν υιοθετηθεί ευρέως σε μία μεγάλη ποικιλία τομέων που εντοπίζονται από την αυτόνομη οδήγηση [1], μέχρι τις βιοιατρικές εφαρμογές [2] και τους Intelligent Personal Assistants (IPAs) [3], κυρίως χάρις την υψηλή ακρίβεια προβλέψεων που παρέχουν.

Τέτοιες περιπτώσεις εφαρμογών μπορούν να βρεθούν συχνα σε edge computing δίκτυα υπολογιστών, κοντά στην τοποθεσία παραγωγής των δεδομένων, με σκοπό να παρέχουν βελτιωμένη ασφάλεια και να μειώσουν τον χρόνο μεταφοράς των δεδομένων σε εξυπηρετητές του Cloud [4]. Όπως αναφέρθηκε νωρίτερα, παρά το γεγονός ότι τα Βαθιά Νευρωνικά Δίκτυα παρέχουν υψηλή ακρίβεια, συχνά συνοδεύονται από υψηλές απαιτήσεις σε υπολογιστική ισχύ και μέγεθος μνήμης, δυσκολεύοντας έτσι τις δυνατότητες ευρείας υιοθέτησής του σε συστήματα περιορισμένων δυνατοτήτων, όπως αυτά που συναντώνται συχνά σε περιβάλλονται edge computing [5]. Επιπροσθέτως, το βάθος (και κατά συνέπεια η υπολογιστική πολυπλοκότητα) των Νευρωνικών Δικτύων συνεχώς αυξάνεται με περισσότερα στρώματα να ενσωματώνονται στις αρχιτεκτονικές των μοντέλων [6], καθιστώντας το πρόβλημα που αναφέρθηκε νωρίτερα ακόμα πιο έντονο.

Μία συνήθης τακτική για την αντιμετώπιση της αύξησης των απαιτήσεων σε υπολογιστικούς πόρους είναι η εκφόρτωση των αντίστοιχων διεργασιών σε εξυπηρετητές υψηλών επιδόσεων του Cloud. Την τρέχουσα στιγμή όταν η εκτέλεση inference στα δίκτυα edge γίνεται υπολογιστικά αδύνατη, τα αντίστοιχα δεδομένα μεταφέρονται για επεξεργασία σε ισχυρούς σέρβερ που γίνονται διαθέσιμοι μέσω υπηρεσιών Cloud (όπως Amazon Elastic Inference ή Azure Machine Learning). Ωστόσο, το κύριο μειονέκτημα αυτής της προσέγγισης είναι ότι παράγει τεράστιο όγκο δεδομένων, η μεταφορά των οποίων μέσω του δικτύου επιφέρει μεγάλα κόστη καθυστέρησης και ενέργειας. Εντείνοντας τους ήδη υπάρχοντες προβληματισμούς, το offloading μεγάλου όγκου αιτημάτων μπορεί να φέρει σε κορεσμό τον ρυθμό επεξεργασίας του Cloud, καθιστώντας το ανήμπορο να υποστηρίξει τις διαρκώς αυξανόμενες απαιτήσεις για υπολογιστικούς πόρους [7]. Λαμβάνοντας αυτά υπόψιν και με τον σκοπό να δημιουργήσουν πιο αποδοτικά υπολογιστικά συστήματα, οι σχεδιαστές υλικού (π.χ., Nvidia, Xilinx) εισάγουν ικανές και εξειδικευμένες συσκευές για το edge περιβάλλουν που συχνά ενσωματώνουν κλασσικές CPU μαζί με επιταχυντές υλικού και εξειδικευμένους επιταχυντές Νευρωνικών Δικτύων σε ένα ολοκληρωμένο SoC. Και ενώ αυτές οι συσκευές μπορεί να είναι ικανές να παρέχουν την απαραίτητη υπολογιστική δύναμη για τον υπολογισμό των DNNs, οι επιταχυντές που ενσωματώνουν είναι εξαιρετικά ενεργοβόροι και άρα ακατάλληλοι για εφαρμογές που έχουν περιορισμούς ενέργειας (π.χ. συσκευές που λειτουργούν με μπαταρίες).

Προσπαθώντας να βρεθεί ένας βιώσιμος συμβιβασμός ανάμεσα στον χρόνο εκτέλεσης και την κατανάλωση ενέργειας κατά την εκτέλεση του Inference στις παρυφές του δικτύου, ελαχιστοποιώντας ταυτόχρονα

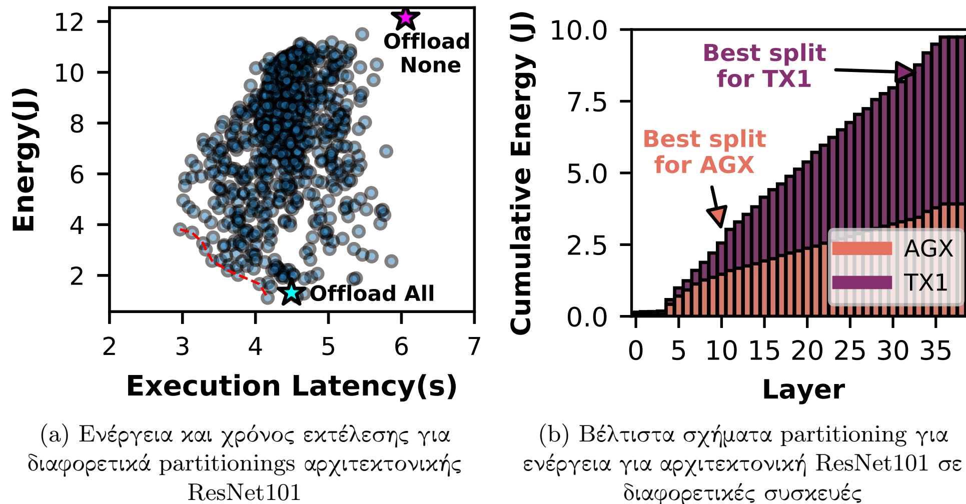
τα μειονεκτήματα της μεταφοράς δεδομένων μέσω του δικτύου, μία λύση που ερευνάται είναι αυτή του partitioning και του offloading Νευρωνικών Δικτύων.

Ο κύριος στόχος της παραπάνω μεθόδου είναι να παραχθεί ένα "σπάσιμο" του Νευρωνικού Δικτύου έτσι ώστε μέρος των υπολογισμών να εκτελεστεί τοπικά ένα ένα άλλο τμήμα τους να εκφορτωθεί στο Cloud. Ωστόσο η εύρεση τέτοιων partitionings μπορεί να είναι απαιτητική καθώς εξαρτάται από μία τριάδα παραμέτρων όπως:

- Η αρχιτεκτονική του DNN (π.χ., χρόνος/ενέργεια για την εκτέλεση, χρόνος μετάδοσης κάθε στρώματος).
- Η φύση της εφαρμογής (περιορισμοί χρόνου εκτέλεσης/ενέργειας, ύπαρξη προθεσμιών).
- Τα χαρακτηριστικά ενέργειας και επεξεργαστικής ισχύος του διαθέσιμου hardware.

Επιβεβαιώνοντας τα παραπάνω, το Σχήμα 1.1.1a δείχνει την ενέργεια και τον χρόνο εκτέλεσης για όλα τα πιθανά σπασίματα ένος ResNet101 DNN.

Όπως φαίνεται, μέσω της προτεινόμενης μεθόδου μπορούμε να αναμένουμε σημαντικά γρηγορότερη εκτέλεση και μειωμένη κατανάλωση ενέργειας σε σχέση με την πλήρως τοπική ή την εξ'ολοκλήρου Cloud εκτέλεση. Επιπλέον, το Σχήμα 1.1.1b δείχνει πως μεταβάλλονται τα σημεία στα οποία γίνεται η τμηματοποίηση του ίδιου νευρωνικού δικτύου όταν αλλάζουν οι edge συσκευές. Όλα τα παραπάνω επιβεβαιώνουν ότι το πρόβλημα της εύρεσης τέτοιων σπασιμάτων είναι πραγματικά ένα δύσκολο πρόβλημα.



(a) Ενέργεια και χρόνος εκτέλεσης για διαφορετικά partitionings αρχιτεκτονικής ResNet101

(b) Βέλτιστα σχήματα partitioning για ενέργεια για αρχιτεκτονική ResNet101 σε διαφορετικές συσκευές

Figure 1.1.1: Βέλτιστα σχήματα DNN partitioning ως προς *α)* διαφορετικούς στόχους βελτιστοποίησης, *β)* ετερογένεια των συσκευών.

Στοχεύοντας στο να αντιμετωπίσουν τις προκλήσεις που αναφέρθηκαν νωρίτερα, υπάρχουσες δουλειές έχουν επιχειρήσει να δώσουν λύση στο πρόβλημα του DNN partitioning και offloading [8]–[10]. Ωστόσο, αρκετά από αυτά τα έργα βασίζονται σε θεμελιώδεις υποθέσεις σχετικά με την φύση του προβλήματος, όπως η πρότερη γνώση του υλικού πάνω στο οποίο θα εκτελεστεί το inference ή η ακριβής φύση των χαρακτηριστικών του DNN, επιτρέποντας με αυτό τον τρόπο την δημιουργία σχημάτων partitioning μέσω εκτεταμένου profiling. Δεδομένου του χαρακτήρα που έχουν τα edge computing συστήματα, με νέες συσκευές και εφαμργογές να εισέρχονται δυναμικά στο δίκτυο, οι υποθέσεις που αναφέρθηκαν νωρίτερα μπορεί να μην είναι εφαρμόσιμες. Μάλιστα, αναδυόμενες αρχιτεκτονικές που συνοδεύονται από προηγμένες τεχνικές, όπως skip-layer connections και early exits, δείχνουν ότι η δυναμική αυτή φύση κατά πάσα πιθανότητα δεν θα αλλάξει.

Σε αυτή την εργασία, επικεντρωνόμαστε σε δύο διαφορετικούς τομείς. Αρχικά δείχνουμε πως το Collaborative Filtering (CF) μπορεί να χρησιμοποιηθεί για τη πρόβλεψη της ενέργειας και του χρόνου εκτέλεσης στρωμάτων νευρωνικών δικτύων σε μία συστοιχία ετερογενών edge συσκευών. Αναλυτικότερα,

α) Αναπτύσσουμε έναν αλγόριθμο CF που βασίζεται σε παραγοντοποίηση πινάκων *α)* Επιταχύνουμε τον αλγόριθμο με χρήση `C++` και `OpenMP` *γ)* Εκπαιδεύουμε τον αλγόριθμο σε δεδομένα ενέργειας και latency διαφορετικών συσκευών από ένα υποσύνολο των στρωμάτων των DNNs *δ)* Δείχνουμε ότι ο αλγόριθμος αυτός μπορεί να παράξει ακριβείς προβλέψεις για άγνωστα ζεύγη συσκευής-στρώματος. Σε δεύτερο στάδιο αναπτύσσουμε ένα πλαίσιο που πραγματοποιεί DNN partitioning και offloading δημιουργώντας ένα σύνολο Pareto-optimal λύσεων, ανάμεσα σε χρόνο εκτέλεσης και ενέργεια. Για αυτό τον λόγο *α)* επιλέγουμε ένα σύνολο νευρωνικών δικτύων από διαδεδομένες DNN αρχιτεκτονικές *β)* Εκτελούμε profiling ενός τυχαίου υποσυνόλου στρωμάτων DNN για ενέργεια και latency σε ένα cluster NVIDIA edge συσκευών που είναι εφοδιασμένες με GPU, καθώς και σε έναν x86 εξυπηρετητή *γ)* Χρησιμοποιούμε τον CF αλγόριθμο που αναφέρθηκε νωρίτερα για την πρόβλεψη τιμών ενέργειας και latency για όλα τα ζεύγη συσκευής-στρωμάτων *δ)* Υλοποιούμε έναν αλγόριθμο πολυπλοκότητας $O(N^2)$ που εξερευνά διαφορετικά partitionings *ϵ)* Δείχνουμε πως αυτά τα partitionings αλλάζουν με την μεταβολή διαφόρων παραμέτρων και *στ)* Συγκρίνουμε τη λύση μας με υπάρχουσες προσεγγίσεις, ανάμεσα τους και ένα framework με το όνομα Neurosurgeon [8], δείχνοντας ότι μπορούμε να έχουμε κέρδη σε ταχύτητα εκτέλεσης και ενέργεια.

1.2 Σχετική Βιβλιογραφία

Ο Λασκαρίδης κ.α [11] παρουσιάζουν ένα σύνολο τεχνικών προκειμένουν να μειωθεί ο χρόνος εκτέλεσης των Νευρωνικών Δικτύων σε edge συσκευές. Η προσέγγιση που προτάθηκε συνδυάζει μεθόδους όπως early exits, DNN partitioning και offloading, όπως και επίσης την μείωση της αριθμητικής ακρίβειας για τα δεδομένα που μεταφέρονται προσπαθώντας να μειωθεί το κόστος μεταφοράς μέσω του δικτύου. Η λύση που προτάθηκε μπορεί να μεταβάλει ποικιλοτρόπως την εκτέλεση του DNN, κάνοντας offloading ή εξετάζοντας διάφορα σημεία early exit μέχρι να ξεπεραστεί ένα συγκεκριμένων κατώφλι βεβαιότητας. Η επιλογή των διαφόρων μεθόδων εκτέλεσης γίνεται με δεδομένα profiling που έχουν συγκεντρωθεί νωρίτερα, ενώ λαμβάνονται επίσης και πιθανές διακυμάνσεις στην κατάσταση του δικτύου καθώς και ο φόρτος εργασίας του εξυπηρετητή offloading.

Ο Kang κ.α [8] προτείνουν ένα πλαίσιο το οποίο θα πραγματοποιεί partitioning και offloading Νευρωνικών Δικτύων μεταξύ του edge και του cloud, εκτελώντας τοπικά μέχρι ένα στρώμα i και συνεχίζοντας την εκτέλεση με offloading. Η προτεινόμενη αρχιτεκτονική μπορεί να παράγει δύο σχήματα offloading ένα για βέλτιστο χρόνο εκτέλεσης και ένα για ελάχιστοποίηση της κατανάλωσης ενέργειας, ενώ οι προβλέψεις για τον χρόνο εκτέλεσης των στρωμάτων καθώς και της ενεργειακής τους κατανάλωσης γίνονται με την βοήθεια τεχνικών Machine Learning. Η λύση υπολογίζει επίσης το φορτίο υπό το οποίο βρίσκεται το Cloud, εισάγοντάς το σαν παράμετρο στα Machine Learning μοντέλα που χρησιμοποιούνται. Μία υπόθεση που γίνεται είναι ότι τα βάρη των Νευρωνικών δικτύων προς offloading προϋπάρχουν στο Cloud και άρα δεν χρειάζεται να μεταφερθούν κατά τη διάρκεια του offloading.

Οι συγγραφείς του DeepThings [5] χρησιμοποιούν μία διαφορετική τεχνική partitioning σε σχέση με αυτές που παρουσιάστηκαν μέχρι τώρα. Αντί να χωρίσουν τα στρώματα του νευρωνικού σε αυτά που εκτελούνται τοπικά και αυτά που εκτελούνται απομακρυσμένα, ενώνουν πολλά στρώματα σε ένα ενιαίο κομμάτι επεξεργασίας το οποίο μετά χωρίζεται σε "πλακίδια" επεξεργασίας μέσω της χρήση ενός Fused Tile Partitioning αλγορίθμου που φροντίζει να εξαλείφει τις εξαρτήσεις μεταξύ γειτονικών πλακιδίων εισάγοντας μία μικρή επικάλυψη μεταξύ τους. Με αυτό τον τρόπο, το framework πετυχαίνει εκτός από επιτάχυνση, μείωση και στην απαιτούμενη μνήμη, επιτρέποντας την εκτέλεση Νευρωνικών Δικτύων σε συσκευές που δεν θα μπορούσαν τα εκτελέσουν προηγουμένως.

Στο MoDNN [4] οι συγγραφείς εισάγουν μία τεχνική spectral clustering προκειμένου να ομαδοποιήσουν τα μη-μηδενικά βάρη των πλήρως συνδεδεμένων στρωμάτων σε πυκνούς (dense) πίνακες στοιχείων, οι οποίο μετά μπορούν να διανεμηθούν σε ένα σύνολο συσκευών ώστε να πραγματοποιηθεί ο υπολογισμός των στρωμάτων. Τα συνελικτικά στρώματα μοιράζονται και αυτά στο cluster των συσκευών, χωρίζοντας τα κατά μήκος της μεγαλύτερής τους διάστασης. Παρόμοιες ιδέες χρησιμοποιούνται και στο [10] με το πρόγραμμα του partitioning να ανάγεται σε ένα πρόβλημα Ακέραιου Γραμμικού Προγραμματισμού (ILP), προσθέτοντας στον ορισμό του προβλήματος το κόστος που εισάγει το δίκτυο για καθυστέρηση και ενέργεια.

Στο [12] χρησιμοποιούνται διάφορες νέες μέθοδοι προκειμένου να αποκαλυφθεί και να αξιοποιηθεί η παρ-

αλληλία στα γραμμικά και συνελικτικά στρώματα των νευρωνικών δικτύων. Αναλυτικότερα, οι συγγραφείς πειραματίζονται με το partitioning των καναλιών της εισόδου (channel splitting) και με το partitioning των τανιστών εισόδου κατά μήκος των διαστάσεων X και Y (spatial splitting). Μια άλλη τεχνική που ερευνάται είναι αυτή του filter splitting στην οποία μοιράζονται τα φίλτρα των συνελικτικών στρωμάτων. Με αυτές τις τεχνικές, οι συγγραφείς στοχεύουν στην αύξηση της ταχύτητας του inference καθώς και στην μείωση των απαιτήσεων στο σύστημα μνήμης, ιδιαίτερα σε συστήματα περιορισμένων πόρων, όπως είναι τα Raspberry Pi που χρησιμοποιούνται κατά την αξιολόγηση της λύσης.

Το πρόβλημα του partitioning και του offloading έχει μελετηθεί και με προσεγγίσεις οι οποίες ξεπερνούν τα όρια της επιστήμης των υπολογιστών. Στο [13] το πρόβλημα της ανάθεσης διεργασιών σε ένα cluster edge συσκευών μοντελοποιείται με βάση την οικονομική θεωρία. Πιο συγκεκριμένα, οι συσκευές λαμβάνουν ανά περιοδικά διαστήματα ένα ποσό *Χρημάτων* ανάλογα με τις υπολογιστικές ικανότητες που διαθέτουν εκείνες, οι υπόλοιπες συσκευές στο cluster καθώς και το gateway. *Ξοδεύοντας* αυτά τα χρήματα, μπορούν να *Αγοράσουν* υπολογιστικούς πόρους στο gateway μέσω μίας δημοπρασίας. Η ορθή λειτουργία του συστήματος αυτού βασίζεται σε αρχές της οικονομικής θεωρίας όπως το ισοζύγιο προσφοράς και ζήτησης. Μία άλλη μέθοδος η οποία διαπερνά τα στεγανά του Computer Science είναι η εισαγωγή της Θεωρίας Παιγνίων για την μοντελοποίηση του προβλήματος, όπως γίνεται στο [14]. Υπό αυτή τη σκοπιά, το πρόβλημα της ιδανικής ανάθεσης διεργασιών σε συσκευές λύνεται με την χρήση Stackelberg games προκειμένου να διαπιστωθεί αν μία διεργασία μπορεί να εκφορτωθεί, με την συσκευή που θα εκτελέσει το task να καθορίζεται μέσω μίας δημοπρασίας Vickrey (δεύτερης τιμής).

Τέλος, η χρήση Collaborative Filtering προκειμένου να γίνουν προβλέψεις σε ετερογενή περιβάλλοντα έχει εξερευνηθεί στο [15], όταν μία παρόμοια τεχνική εφαρμόστηκε για να βρεθεί ο βέλτιστος από ένα σύνολο servers προκειμένου να ανατεθεί σε αυτόν ένα συγκεκριμένο workload. Αυτό επιτυγχάνεται μέσω του σύντομου profiling της εφαρμογής σε δύο τυχαία επιλεγμένους εξυπηρετητές και περνώντας τα αποτελέσματα από τον αλγόριθμο CF, δημιουργώντας έτσι προβλέψεις για το σύνολο του cluster εξυπηρετητών.

1.3 Partitioning και Offloading Νευρωνικών Δικτύων σε Edge Συστήματα

1.3.1 Ορισμός Προβλήματος

Όπως φάνηκε στο Σχήμα 1.1.1a υπάρχουν πολλοί τρόποι για να γίνει το partitioning ενός Νευρωνικού Δικτύου. Από αυτούς τους τρόπους, μερικοί ορίζουν ένα μέτωπο Pareto βέλτιστων λύσεων ανάμεσα σε στόχους ενέργειας και επίδοσης, στο οποίο κανένο στοιχείο δεν μπορεί να βελτιώσει τον έναν στόχο χωρίς να χειροτερεύσει τον άλλο. Σε αυτό το κεφάλαιο θα περιγράψουμε τις προσπάθειες μας για να ανακαλύψουμε και να εξερευνήσουμε αυτό το σύνολο λύσεων.

1.3.2 Προτεινόμενη Υλοποίηση

Η υλοποίηση που προτείνουμε, αντιμετωπίζει το πρόβλημα του partitioning και offloading Νευρωνικών Δικτύων, πάνω από ένα σύνολο ετερογενών CPU/GPU edge συστημάτων. Κάθε συσκευή φιλοξενεί ένα σύνολο διεργασιών για inference τα οποία και πρέπει να εκτελεστούν. Ο στόχος που θέτουμε είναι να βρεθεί το σύνολο των Pareto-optimal λύσεων που αναφέρθηκαν νωρίτερα σε βαθμό λεπτομέρειας στρώματος Νευρωνικού Δικτύου, εκφορτώνοντας υπολογιστικά κομμάτια των μοντέλων με σκοπό να βελτιωθεί η συνολικά καταναλισκόμενη ενέργεια ή/και ο χρόνος εκτέλεσης. Στο Σχήμα 1.3.1 παρουσιάζεται η αρχιτεκτονική της λύσης μας η οποία χωρίζεται σε δύο στάδια που διαθέτουν από δύο μηχανισμούς το κάθε ένα. Αρχικά υπάρχει ένα *offline* στάδιο που αποτελείται από τον *DNN Profiler* και τον *Network Profiler*, και το οποίο ακολουθείται από ένα *online* στάδιο που περιέχει τον *Predictor* και τον *Offloader*. Παρακάτω, θα αναλύσουμε τον ακριβή τρόπο λειτουργίας της υλοποίησης μας.

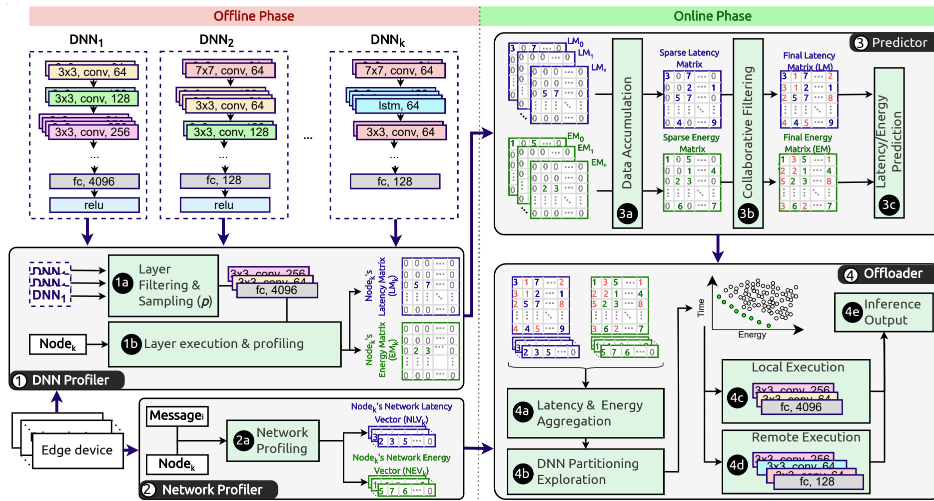


Figure 1.3.1: Γενικό σχήμα της λύσης που προτείνουμε.

1.3.3 Offline Στάδιο

Το *Offline στάδιο* της εκτέλεσης αποτελείται από δύο μηχανισμούς, τον *DNN Profiler* (❶) και τον *Network Profiler* (❷), οι οποίοι είναι υπεύθυνοι για την συγκέντρωση των δεδομένων που θα δοθούν ως είσοδος στους runtime μηχανισμούς του *Online σταδίου* αργότερα. Ως είσοδο στο *Offline στάδιο* δίνουμε ένα σύνολο διαφορετικών DNN αρχιτεκτονικών στους κόμβους του edge computing δικτύου.

1.3.4 DNN profiler

Προσπαθώντας να εκμεταλευτούμε την εγγενώς κατανεμηνέμη φύση του edge computing υπολογιστικού μοντέλου, την ετερογένεια των συσκευών και των αρχιτεκτονικών Νευρωνικών Δικτύων, υλοποιούμε ένα μηχανισμό Collaborative Filtering [15], [16] με σκοπό να εκπαιδεύσουμε το σύστημα μας έτσι ώστε να προβλέπει αποδοτικά την ενεργειακή κατανάλωση και τον χρόνο εκτέλεσης κάθε στρώματος Νευρωνικού Δικτύου για κάθε συσκευή. Σαν πρώτο βήμα σε αυτό τον μηχανισμό εισάγουμε ένα σύστημα ***Φιλτραρίσματος και Δειγματοληψίας*** (1a) το οποίο θα επιλέξει στρώματα από τα DNN που δίνουμε σαν είσοδο για περαιτέρω επεξεργασία. Αναλυτικότερα, ένα μικρό ποσοστό p όλων των στρωμάτων των νευρωνικών θα επιλεχθεί τυχαία προκειμένου να εξεταστεί αναλυτικά για τον χρόνο εκτέλεσης και την κατανάλωση ενέργειας. Με αυτό τον τρόπο εξασφαλίζεται οτι ο συνολικός χρόνος του profiling που απαιτείται δεν θα εκτιναχθεί όταν αυξηθεί ο αριθμός των στρωμάτων των Νευρωνικών Δικτύων. Το ποσοστό αυτό p λαμβάνεται μέσω πειραματισμού.

Στην συνέχεια, τα στρώματα που επιλέχθηκαν θα δοθούν σαν είσοδος στον μηχανισμό ***Εκτέλεσης και Profiling Στρωμάτων*** (1b). Σε αυτή την περίπτωση, κάθε ένα από τα επιλεγμένα στρώματα θα εκτελεστούν τοπικά στον edge κόμβο. Η εκτέλεση τους θα μετρηθεί για τον χρόνο και την ενέργεια που απαιτήθηκε μέχρι να ολοκληρωθεί ο υπολογισμός τους. Από τα μεγέθη που μετρήθηκαν δημιουργούνται δύο πίνακες οι *Node's Latency Matrix* και *Node's Energy Matrix* οι οποίοι αργότερα θα δοθούν σαν είσοδος στον μηχανισμό *Predictor*.

1.3.5 Network Profiler

Οι δυνατότητες ενός συστήματος edge computing συστήματος είναι άρρηκτα συνδεδεμένες με την δυνατότητα του να λειτουργεί πάνω από τις υπάρχουσες δικτυακές υποδομές. Επιπλέον, ένας μηχανισμός partitioning και offloading όπως αυτός που προτείνεται, θα πρέπει να λαμβάνει υπόψιν το κόστος latency και ενέργειας που επιβάλει η μεταφορά δεδομένων μέσω του δικτύου. Για αυτό τον λόγο ενσωματώνουμε σε κάθε υπολογιστικό κόμβο k ένα σύστημα *Network Profiling* (2a). Σαν είσοδο αυτού του τμήματος δίνουμε ένα σύνολο από μηνύματα διαφορετικών μεγεθούν που κυμαίνονται από το μέγεθος των KB μέχρι

και το μέγεθος μερικών GB. Η ενέργεια και ο χρόνος μετάδοσης του κάθε μηνύματος καταγράφεται τόσο για την διαδικασία απόστολής όσο και για την λήψη του. Όταν ολοκληρωθεί αυτή η διαδικασία ο κάθε κόμβος k μπορεί να χαρακτηριστεί ως προς το δίκτυο από δύο διανύσματα, ένα διάνυσμα ενέργειας και ένα χρόνου ενέργειας (Network Latency Vector ή NLV_k και Network Energy Vector ή NEV_k αντίστοιχα). Τα παραγόμενα διανύσματα χρησιμοποιούνται για να δημιουργηθούν πολυωνυμικές καμπύλες (4ου βαθμού) οι οποίες θα μπορούν να χρησιμοποιηθούν στο runtime (*Online στάδιο*) για την εκτέλεση προβλέψεων σχετικά με το overhead που εισάγει το δίκτυο.

1.3.6 Online Στάδιο

Ένα αποδοτικό framework διαχείρησης πόρων θα πρέπει να είναι ικανό να λαμβάνει αποφάσεις με δυναμικό τρόπο στο run-time. Για αυτό τον λόγο, ενσωματώνουμε στην αρχιτεκτονική μας δύο βασικά συστήματα: τον *Predictor* (❸) και τον *Offloader* (❹). Το πρώτο είναι υπεύθυνο για την πρόβλεψη του απαιτούμενου χρόνου και ενέργειας για την εκτέλεση κάθε στρώματος, ενώ το δεύτερο είναι υπεύθυνο για την συγκέντρωση των δεδομένων από το profiling του δικτύου και των αποτελεσμάτων του Collaborative Filtering, χρησιμοποιώντας τα για την εύρεση των partitionings και την εκτέλεση του offloading.

1.3.7 Predictor

Οι πίνακες χρόνου εκτέλεσης και ενέργειας που παράγονται από τον *DNN Profiler* (❶) σε κάθε κόμβο k συγκεντρώνονται στον edge server κατά το στάδιο (3a). Αυτά τα δεδομένα οργανώνονται σε δύο νέους αραιούς πίνακες, τους *Sparse Latency Matrix* και *Sparse Energy Matrix* αντίστοιχα. Σε αυτούς τους πίνακες, κάθε γραμμή του πίνακα αντιστοιχεί σε μία edge συσκευή ενώ κάθε στήλη του πίνακα αντιστοιχεί σε ένα είδος στρώματος Νευρωνικού Δικτύου. Η δημιουργία αυτών των πινάκων πυροδοτεί την εκτέλεση του αλγορίθμου Collaborative Filtering (3b), ο οποίος βασίζεται στην παραγοντοποίηση των αραιών πινάκων σε ένα γινόμενο μικρότερων πινάκων, κάνοντας χρήση gradient descent. Όταν ολοκληρωθεί ο αλγόριθμος, οι κενές θέσεις των αραιών πινάκων γεμίζονται με τις προβλέψεις, δημιουργώντας δύο νέους πίνακες, τον *Final Latency Matrix* και τον *Final Energy Matrix*. Όταν ολοκληρωθούν οι προβλέψεις, κάθε κόμβος του cluster καλείται να παραλάβει τα αποτελέσματα που τον αφορούν, τα οποία θα χρησιμοποιηθούν κατα τελική *Πρόβλεψη Χρόνου εκτέλεσης/Ενέργειας* (3c). Σε αντίθεση με προηγούμενες λύσεις που βασίζονται σε εκτεταμένο προγενέστερο profiling, το framework που προτείνουμε εμείς είναι ικανό να ανταποκρίνεται σε δυναμικά σενάρια. Νέοι κόμβοι που εντάσσονται στο cluster ενσωματώνονται στους πίνακες του Collaborative Filtering, ενώ νέες αρχιτεκτονικές DNN μπορούν να επωφεληθούν από τα ήδη υπάρχοντα στρώματα.

1.3.8 Offloader

Το τελευταίο κομμάτι του *Online Σταδίου* είναι υπεύθυνο για την συγκέντρωση των δεδομένων του δικτύου και του Collaborative Filtering, την πραγματοποίηση του partitioning και την εκτέλεση του offloading. Σε κάθε κόμβο k τα διανύσματα που προκύπτουν από το profiling του δικτύου (NLV_k, NEV_k) καθώς και πίνακες δεδομένων του Collaboative Filtering (LM, EM) συγκεντρώνονται στο σημείο (4a), προκειμένου να προκύψουν οι τελικές προβλέψεις για κάθε στρώμα, συμπεριλαμβανομένου του κόστους επεξεργασίας και μετάδοσης. Στην συνέχεια, περνάμε στο σύστημα *Εξερεύνησης DNN Partitionings* (4b), με σκοπό να δημιουργήσουμε ένα σύνολο από Pareto-optimal λύσεις που βελτιστοποιούν την ενέργεια και τον χρόνο του inference. Η εξέταση των διαφόρων partitionings γίνεται μέσω ενός έξυπνου αλγορίθμου πολυπλοκότητας $O(N^2)$ ο οποίος διατρέχει όλα τα πιθανά partitionings ενός νευρωνικού δικτύου σε δύο σημεία, μεταβάλλοντας δύο δείκτες i και j. Από αυτά δημιουργείται το Pareto Optimal set των λύσεων, ενώ αξιολογούνται επίσης σαν πιθανές λύσεις οι περιπτώσεις της πλήρως τοπικής ή της πλήρως offloaded εκτέλεσης, οι οποίες παραμένουν βιώσιμες εναλλακτικές αν ανήκουν στο Pareto optimal σύνολο. Κάθε partitioning (i, j) που αξιολογείται, αντιστοιχεί σε ένα σχήμα εκτέλεσης κατά το οποίο τα στρώματα του Νευρωνικού από 0 έως i εκτελούνται τοπικά στην συσκευή, τα στρώματα $i + 1$ μέχρι και j γίνονται offload στον edge server, με τον έλεγχο της εκτέλεσης να επιστρέφει τοπικά στην συσκευή για την εκτέλεση των στρωμάτων $j+1$ μέχρι και το τέλος του DNN. Σε αντίθεση με υπάρχουσες προσεγγίσεις όπως αυτή που παρουσιάζεται στο [8], η λύση που προτείνουμε εμείς μπορεί να αντιμετωπίσει

την ύπαρξη εξαρτήσεων τύπου shortcut που εμφανίζονται στα Residual DNNs. Σε αυτή την περίπτωση, η εξάρτηση επιλύεται περιορίζοντας την μέσα σε ένα ατομικό μπλοκ από στρώματα, τα οποία μπορούν να γίνουν offload σαν ενιαίο πακέτο, λύνοντας έτσι το πρόβλημα που παρουσιάστηκε νωρίτερα.

1.4 Αξιολόγηση

1.4.1 Πειραματική Διάταξη

Για την αξιολόγηση του framework που προτείναμε, χρησιμοποιούμε ένα σύστημα ετερογενών CPU/GPU συσκευών της NVIDIA, με τα ακριβή χαρακτηριστικά τους να παρουσιάζονται στον Πίνακα 1.1. Σαν μηχάνημα offloading χρησιμοποιούμε έναν ισχυρό server αρχιτεκτονικής x86, ο οποίος διαθέται μία κάρτα γραφικών NVIDIA V100, δημιουργώντας έτσι μία διάταξη από συσκευές που είναι διαδεδομένη στο edge και το cloud [17].

Table 1.1: Τεχνικά χαρακτηριστικά των εξεταζόμενων συσκευών

Device	CPU	L2	L3	DRAM	GPU
Jetson AGX	8×Carmel@2.2GHz ARMv8.2 64-bit	8MB	4MB	32GB	512×Volta@1.4GHz
Jetson NX	6×Carmel@1.4GHz ARMv8.2 64-bit	6MB	4MB	8GB	384×Volta@1.1GHz
Jetson Nano	4×Cortex-A57@1.4GHz ARMv8 64-bit	2MB	-	4GB	128×Maxwell@0.9GHz
Jetson TX1	4×Cortex-A57@1.7GHz ARMv8 64-bit	2.5MB	-	4GB	256×Maxwell@1.0GHz

Η υλοποίηση του framework είναι γραμμένη σε γλώσσα Python, ενώ όλες οι συσκευές επικοινωνόυν μεταξύ τους μέσω ενός ασυρμάτου δικτύου ταχύτητας 80 MB/s. Για την αποστολή μηνυμάτων ελέγχου χρησιμοποιείται το πρωτόκολλο ZeroMQ, ενώ ο κύριως όγκος δεδομένων μεταφέρεται μέσω FTP. Με σκοπό να ξεπεράσουμε τις διαφορές των ετερογενών αρχιτεκτονικών που χρησιμοποιούμε, η λύση μας είναι ενσωματωμένη μέσα σε docker containers. Η υλοποίηση του CF αλγορίμου έχει επιταχυνθεί αρχικά με χρήση `C++` και έπειτα με `OpenMP` [18] προκειμένου να πιάσει υψηλές επιδόσεις. Τέλος, οι αρχιτεκτονικές νευρωνικών δικτύων όπως και οι υλοποιήσεις των στρωμάτων GPU σε CUDA έχουν ληφθεί από το PyTorch [19].

Η αξιολόγηση γίνεται χρησιμοποιώντας ένα σύνολο ευρέως διαδεδομέων αρχιτεκτονικών Νευρωνικών Δικτύων. Εξετάζουμε αρχιτεκτονικές όπως το AlexNet, VGG και ResNets καθώς και το δίκτυο MobileNetV2, όλα εκ των οποίων χρησιμοποιούνται σε εφαρμογές ανίχνευσης αντικειμένων και κατηγοριοποίσης εικόνων στο περιβάλλον του edge.

Θα αξιολογήσουμε την λύση που προτείνουμε για τρεις βασικές μετρικές: *i)* Απόδοση *ii)* Κατανάλωση ενέργειας και *iii)* Ακρίβεια προβλέψεων της Collaborative Filtering μεθόδου μας. Δύο βασικές υλοποιήσεις εναντίων των οποίων θα συγκριθούμε είναι η πλήρως τοπική και η πλήρως offloaded επεξεργασία με ονόματα *Offload None* και *Offload All* αντίστοιχα. Επιπλέον υλοποιούμε και συγκρινόμαστε εναντίον ενός framework που πραγματοποιεί DNN partitioning και offloading με το όνομα Neurosurgeon [8]. Επειδή το Neurosurgeon λειτουργεί με την υπόθεση ότι τα βάρη των Νευρωνικών Δικτύων έχουν μεταφερθεί εκ των προτέρων στο Cloud, υλοποιούμε και μία έκδοση του στην οποία η αποστολή των στρωμάτων γίνεται στο run-time, ενώ υλοποιούμε και μία τροποποιημένη εκδοχή του δικού μας framework, στην οποία το offloading γίνεται a priori, προκειμένου η σύγκριση με το Neurosurgeon να είναι δίκαια.

1.4.2 Αποτελέσματα

Απόδοση και Κατανάλωση ενέργειας

Σαν πρώτο πείραμα, αξιολογούμε την λύση μας κατά τον τρόπο που αναφέρθηκε προηγουμένως. Δεδομένου του γεγονότος ότι το Neurosurgeon δεν μπορεί να λειτουργήσει με την χρήση Residual DNNs, χρησιμοποιούμε για την αξιολόγηση μόνο τις αρχιτεκτονικές VGG και AlexNet κάτι το οποίο φαίνεται στα Σχήματα 1.4.1a,1.4.1b για CPU και GPU αντίστοιχα. Στα διαγράμματα του Σχήματος 1.4.1, το επίπεδο χωρίζεται σε 4 τεταρτημόρια:

- Ένα πράσινο τεταρτημόριο στο οποίο η λύση μας κερδίζει τόσο από άποψη απόδοσης όσο και από άποψη ενέργειας.
- Δύο πορτοκαλί τεταρτημόρια στα οποία η λύση μας κερδίζει είτε μόνο ως προς απόδοση είτε μόνο ως προς ενέργεια.
- Ένα κόκκινο τεταρτημόριο στο οποίο η λύση μας είναι χειρότερη και για τις δύο μετρικές έναντια στην υλοποίηση με την οποία συγκρίνεται.

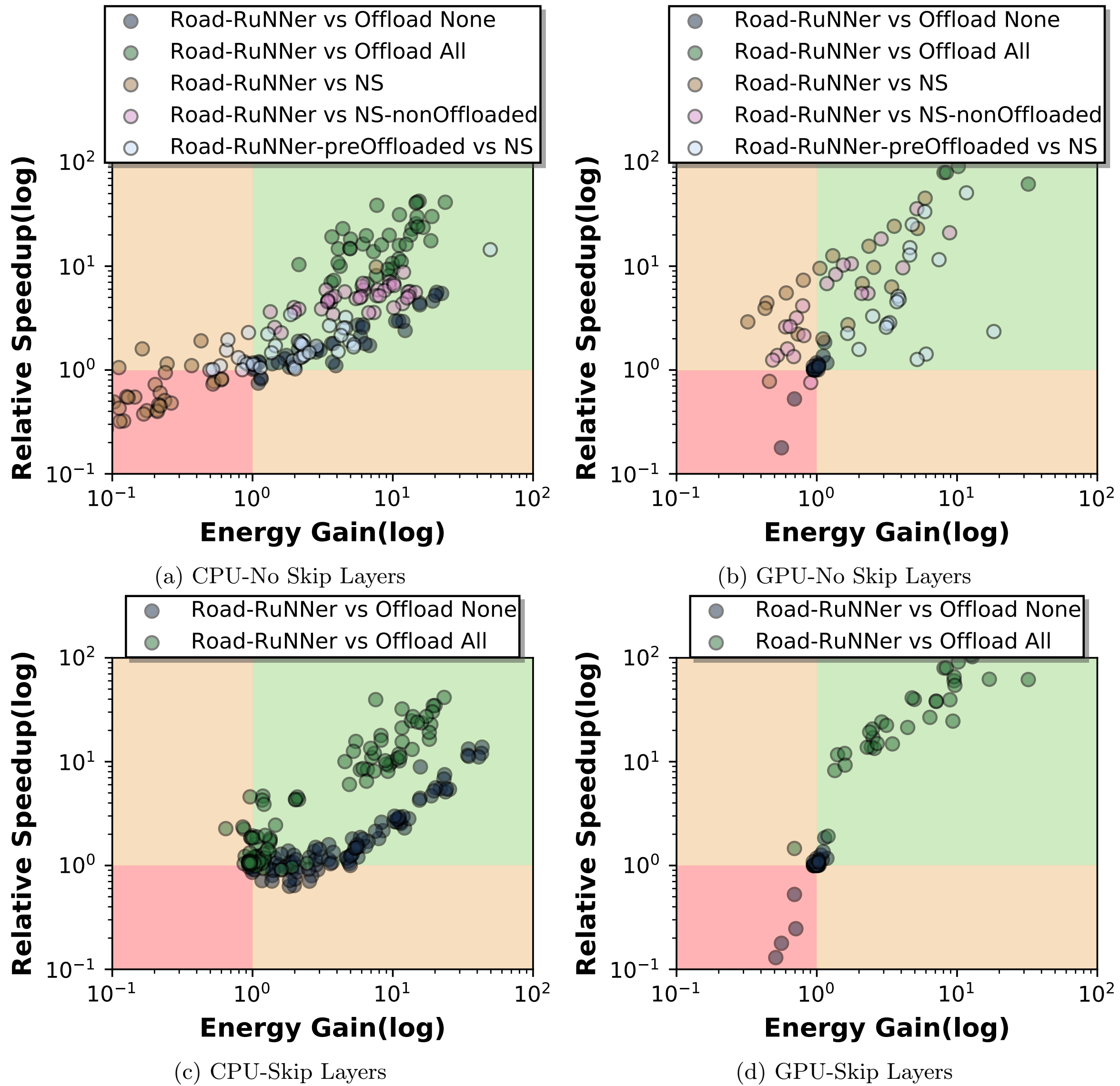


Figure 1.4.1: Σύκριση ενέργειας και απόδοσης της προτεινόμενης υλοποίησης ενάντια σε άλλες προσεγγίσεις για CPU και GPU συσκευές, για ένα ένα σύνολο διαφορετικών DNNs.

Όπως φαίνεται παρουσιάζουμε για εκτελέσεις CPU:

- Μέχρι και 45.41x επιτάχυνση και 95.84% κέρδη ενέργειας ενάντια σε πλήρως offloaded εκτέλεση.
- Μέχρι και 6.61x επιτάχυνση και 95.87% κέρδη ενέργειας ενάντια σε πλήρως τοπική εκτέλεση.
- Κατά μέσο όρο 4.97 φορές κέρδη ταχύτητας και 81.85% λιγότερη ενέργεια ενάντια στην τροποποιημένη έκδοση του *Neurosurgeon - NS-nonOffloaded.*

Παρόμοια κέρδη παρουσιάζονται για και εκτελέσεις GPU, με τα κέρδη μας ενάντια στην τροποποιημένη έκδοση του *Neurosurgeon* να φτάνουν τις 35.74 φορές και 88.73% για latency και ενέργεια αντίστοιχα. Ο λόγος που παρουσιάζουμε αυτά τα speedup ένταντια στην τροποποιημένη έκδοση είναι ότι η δικιά μας λύση παρέχει μέχρι και δύο partition points, ενώ το *Neurosurgeon* μόλις μία, αναγκάζοντας το είτε να στείλει τα μεγάλα βάρη που βρίσκονται στο τέλος κάποιων νευρωνικών δικτύων είτε να συνεχίσει τοπικά την εκτέλεση έχοντας ωστόσο μειωμένη απόδοση. Η τροποποιημένη έκδοση της δικιάς μας λύσης ενάντια στο κλασσικό *Neurosurgeon* κερδίζει 54.09 φορές από άποψη latency και 58.06 φορές από άποψη ενέργειας.

Συνεχίζουμε την αξιολόγηση με Residual αρχιτεκτονικές Νευρωνικών Δικτύων (τις οποίες το *Neurosurgeon* δεν υποστηρίζει).

Για CPU εκτελέσεις παρατηρούμε ότι:

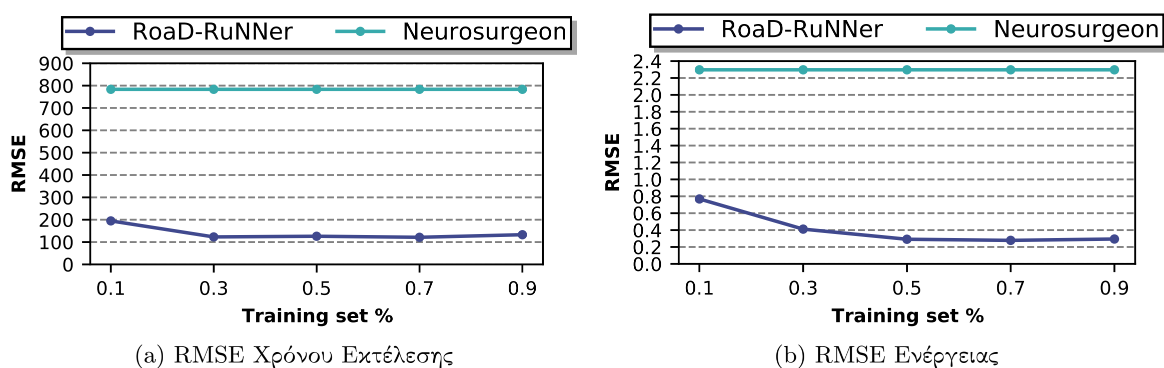
- Λαμβάνουμε μέχρι και 13.92 φορές καλύτερη επίδοση με 46.07 φορές λιγότερη κατανάλωση ενέργειας, ενάντια στην τοπική εκτέλεση.
- Απαιτούμε μέχρι και 45.01 φορές λιγότερο χρόνο και 24.05 φορές λιγότερη ενέργεια σε σχέση με την πλήρως offloaded εκτέλεση.

Για GPU εκτελέσεις:

- Η λύση μας είναι 1.84 φορές ταχύτερη με 1.34x λιγότερη κατανάλωση ενέργειας σε σχέση με την *Offload None* προσέγγιση.
- Η λύση μας είναι 112.98 φορές ταχύτερη και 16.59 φορές λιγότερο ενεργοβόρα σε σχέση με την *Offload All* τακτική.

Ακρίβεια Προβλέψεων

Η απόδοση της υλοποίησης εξαρτάται άμεσα από την ακρίβεια πρόβλεψης του μηχανισμού Collaborative Filtering. Αρχικά, αξιολογούμε την φάση εκπαίδευσης του CF αλγορίθμου. Όπως φαίνεται στα Σχήματα 1.4.2a και 1.4.2b οι προβλέψεις του CF αλγορίθμου συγκλίνουν στις αληθινές τιμές έχοντας δώσει μόλις το 30% των στρωμάτων ως σύνολο εκπαίδευσης. Για αυτό τον λόγο, θέτουμε τον ρυθμό δειγματοληψίας του *DNN Profiler* σε 0.3.

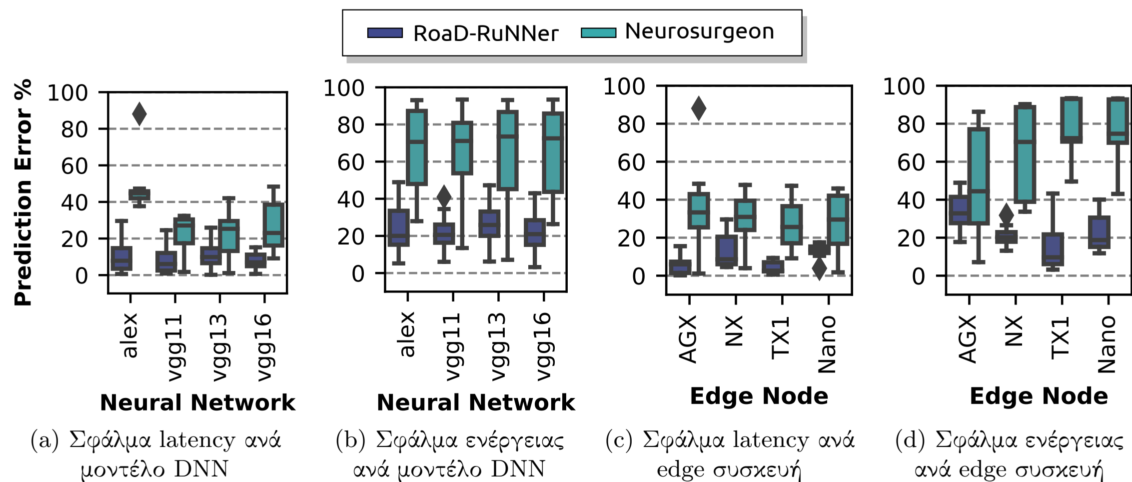


(a) RMSE Χρόνου Εκτέλεσης

(b) RMSE Ενέργειας

Figure 1.4.2: Root Mean Square Error(RMSE) of execution time and energy consumption over alternative Collaborative Filtering fill percentages.

Ακολούθως, αξιολογούμε την συνολική ακρίβεια πρόβλεψης για όλη την εκτέλεση του νευρωνικού, όπως φαίνεται στο Σχήμα 1.4.3.



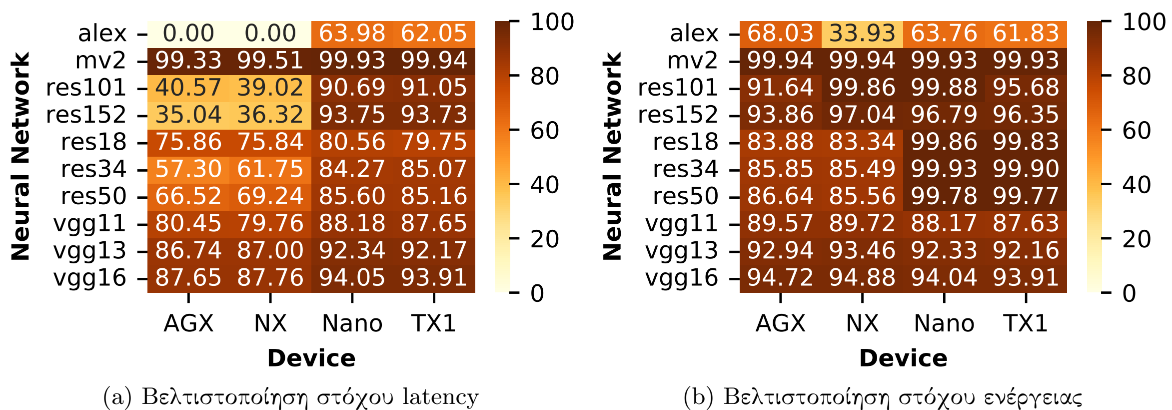
(a) Σφάλμα latency ανά μοντέλο DNN
(b) Σφάλμα ενέργειας ανά μοντέλο DNN
(c) Σφάλμα latency ανά edge συσκευή
(d) Σφάλμα ενέργειας ανά edge συσκευή

Figure 1.4.3: Ακρίβεια πρόβλεψης ενέργειας και χρόνου εκτέλεσης για διαφορετικά μοντέλα DNN και edge συσκευές.

Παρατηρούμε ότι η υλοποίηση μας πετυχαίνει κατά μέσο όρο 68.01% και 63.8% λιγότερο σφάλμα για τον χρόνο εκτέλεσης και την ενέργεια ανά Νευρωνικό αντίστοιχα, σε σχέση με το *Neurosurgeon* (Σχήματα 1.4.3a και 1.4.3b). Επίσης πετυχαίνει μέχρι και 69.6% λιγότερο σφάλμα πρόβλεψης latency και 34.9% λιγότερο σφάλμα στην πρόβλεψη της ενέργειας ανά edge συσκευή (Σχήματα 1.4.3c και 1.4.3d).

Ανάλυση Ποσοστού Offloading

Μία τελεταία ανάλυση που αξίζει να γίνει, έχει να κάνει με το πως αλλάζει το ποσοστό του offloading που πραγματοποιεί η λύση μας όταν μεταβάλλονται διάφορες παράμετροι του περιβάλλοντος εκτέλεσης. Η ανάλυση αυτή γίνεται στα heatmaps του Σχήματος 1.4.4.



(a) Βελτιστοποίηση στόχου latency
(b) Βελτιστοποίηση στόχου ενέργειας

Figure 1.4.4: Ποσοστό offloading για διάφορά DNNs πάνω από ένα ετερογενές σύνολο edge συσκευών.

Από τα δεδομένα του σχήματος μπορούν να βγουν δύο βασικά συμπεράσματα. Αρχικά, παρατηρούμε ότι οι πιο ισχυρές συσκευές όπως το AGX και το NX προτιμούν να κρατούν μεγαλύτερο κομμάτι της επεξεργασίας τοπικά σε σχέση με τις λιγότερο ισχυρές που είναι τα Jetson Nano και TX1 (58.7% και 87.1% ποσοστό offloading κατά μέσο όρο αντίστοιχα). Το δεύτερο συμπέρασμα είναι ότι κρατώντας αμετάβλητη την συσκευή και το Νευρωνικό Δίκτυο παρατηρούμε μεγαλύτερα ποσοστά εκφόρτωσης όταν βελτιστοποιούμε την ενέργεια σε σχέση με όταν βελτιστοποιούμε τον χρόνο εκτέλεσης (90.1% και 74.45% κατά μέσο όρο, αντίστοιχα). Αυτό συμβαίνει καθώς το χρονικό κόστος που εισάγει το δίκτυο για την μεταφορά δεδομένων είναι συχνά απογορευτικό όταν πρέπει να επιτευχθούν αυστηρότεροι στόχοι latency.

1.5 Συμπεράσματα και Μελλοντική δουλειά

Σε αυτή την διπλωματική εργασία παρουσιάζεται ένα πλαίσιο το οποίο πραγματοποιεί DNN partitioning και offloading σε ετερογενή edge computing συστήματα, παράγοντας ένα σύνολο Pareto βέλτιστων partitionings ανάμεσα σε ενέργεια και χρόνο εκτέλεσης. Η λύση που προτείνουμε χρησιμοποιεί τεχνικές Collaborative Filtering προκειμένου να κρατήσει χαμηλά τον απαιτούμενο χρόνο profiling των στρωμάτων Νευρωνικών Δικτύων, περιορίζοντας ταυτόχρονα τα σφάλματα προβλέψεων για χρόνο και ενέργεια έως και 3.19% και 0.18% αντίστοιχα. Σε σχέση με υπάρχουσες προσεγγίσεις, η υλοποίηση μας πετυχαίνει μία επιτάχυνση της τάξης των 9.58 φορών και 88.73% μείωση της ενέργειας κατά μέσο όρο.

Στο μέλλον θα μπορούσαμε να επεκτείνουμε την υπάρχουσα δουλειά με τους ακόλουθους τρόπους:

- Επέκταση του framework έτσι ώστε να λειτουργεί και σε άλλες αρχιτεκτονικές όπως FPGA.
- Offloading μεταξύ γειτονικών συσκευών στο cluster πέρα από τον server.
- Αλγοριθμικές βελτιστοποιήσεις που θα κρύβουν το κόστος μεταφοράς μέσω του δικτύου όπως caching βαρών και pipelining.
- Εξερεύνηση των Power Modes που παρέχουν οι συσκευές και δυναμική εναλλαγή μεταξύ τους.

Chapter 2

Introduction

During the last few years applications that utilize sophisticated Machine Learning (ML) techniques to make value out of complex data are experiencing rapid growth with that being only expected to increase further in the future. In this regard, Deep Neural Networks or DNNs have seen widespread use in a variety of domains ranging from driving [1], to biomedical and healthcare applications [2] and even Intelligent Personal Assistants (IPAs) [3], mostly due to their high prediction accuracy.

Such categories of applications can often be found at the edge of computing networks, close to the source of data, aiming to provide enhanced security and minimize the latency of transferring data to cloud servers [4]. As mentioned previously in the abstract, while DNNs offer exceedingly accurate results, they usually go hand in hand with high demands in computing power and memory size, limiting their ability to be effectively deployed on resource-contrained edge computing devices [5]. Furthermore, the depth (and therefore complexity) of Neural Networks only shows signs of increasing, with more hierarchical layers being integrated in the learned representation [6], thus making the aforementioned problem even more evident.

A commonplace approach to address increased computational demands is to offload the offending tasks to high-end cloud servers. Currently, when inference at the edge becomes computationally impossible, it is frequently offloaded to powerful servers hosted by cloud providers (e.g., Amazon Elastic Inference, Azure Machine Learning). The major drawback of this approach however is that it generates a vast amount of data being transmitted back and forth through the network, resulting in high network overheads and increased energy consumption. What is more, the large volume of computations being offloaded can saturate the throughput rate of the cloud, going as far as rendering it incapable to meet the always-increasing demand for resources [7]. With the aim of creating more efficient and energy proportional computing systems, hardware companies (e.g., Nvidia, Xilinx), can be seen introducing more capable and specialized edge devices that frequently integrate CPUs with on-board hardware accelerators and specialized units for DNN computations (e.g., Nvidia's Tensor cores) in a single System-on-Chip (SoC). And while such hardware may be capable of providing the necessary performance to match contemporary Neural Networks, it still comes at the expense of energy, with aforementioned accelerators and devices being extremely power hungry and often unsuitable for performing tasks under energy or battery constraints.

Attempting to find a viable trade-off between latency and energy for inference at the edge while simultaneously minimizing the bottleneck imposed by the transmission of data over the network, the Partitioning and Offloading of DNNs is being explored as a possible solution [8], [10].

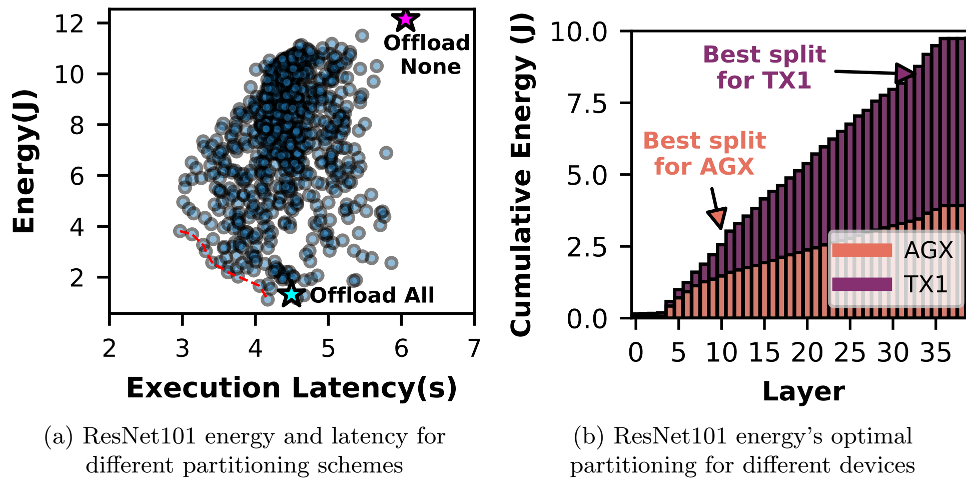
The main goal of DNN partitioning is to generate a layer-level splitting of the Neural Network, through which part of the computation is performed locally (on the edge device) and the rest is offloaded to the cloud. However, discovering such a split can be demanding as it depends on a set of user-specified parameters and the underlying characteristics of the application and hardware. More specifically, a

partitioning scheme must take into consideration:

- The deployed DNN architecture (e.g., latency/energy costs and data transmission time per layer)
- The nature of the application (e.g., latency or energy critical, possible deadlines)
- The performance and energy characteristics of the underlying hardware.

To further elaborate on this point, Fig. 2.0.1a demonstrates the latency and energy of all the possible partitionings for a ResNet101 architecture.

As can be seen, through partitioning of the model we can expect notably faster execution and energy savings when compared to fully local and fully offloaded execution. Moreover, the optimal partitioning scheme for each objective (energy/latency) is different, proving the existence of a space of possibly optimal splittings, a space that we intend to explore in this thesis. Fig. 2.0.1b also shows us that this split can vary for different devices, demonstrating the dependence of optimal splits to the hardware and showing that the problem of partitioning can be a complex one.



(a) ResNet101 energy and latency for different partitioning schemes

(b) ResNet101 energy's optimal partitioning for different devices

Figure 2.0.1: DNN optimal partitioning schemes w.r.t. *a)* different optimization goals and *b)* device heterogeneity.

Aiming to tackle the aforementioned challenges, several prior works have aimed to address the problem of DNN partitioning and offloading [8]–[10]. However, many of these solutions make fundamental assumptions about the problem, such as the having prior knowledge of the hardware or the deployed DNN architecture, allowing generation of the offloading scheme from extensive profiling. Given the fact that edge computing environments tend to be dynamic both in terms of devices joining the network and of deployed applications, the previously mentioned assumptions may not be applicable. This high dynamicity is a trend that is unlikely to change, especially with the introduction of emerging DNN architectures [4] and with the implementation of advanced techniques such as skip-layer connections, early exits and more.

2.1 Contributions

The contributions of this Thesis can be mainly located in two different areas. First we demonstrate how Collaborative Filtering can be used to generate predictions for the latency and energy of Neural Network layers in a cluster of heterogeneous edge devices. More specifically:

- We develop a Matrix Decomposition based Collaborative Filtering algorithm.
- We accelerate the algorithm through the use of C++ and OpenMP in order to attain fast training times.

- We train the algorithm on energy and latency data aggregated from a subset of all the Neural Network layers profiled on different devices.
- We show that the algorithm can accurately infer unknown values for layer and device combinations that it has not been trained on.

Secondly, we develop a Deep Neural Network partitioning and offloading framework that generates a set of Pareto optimal DNN splittings, trading-off between execution latency and energy. In order to achieve this goal:

- We select a number Neural Networks from established DNN architectures.
- In a cluster comprised of NVIDIA GPU-equipped edge devices and a powerful x86 offloading server, we profile a randomly selected subset of the layers from the previously mentioned Neural Networks for energy and latency.
- We utilize the Collaborative Filtering algorithm to generate predictions for every device and layer combination.
- We develop an $O(N^2)$ algorithm that explores different DNN partitioning points.
- We show how these points change for different Neural Networks, latency/energy objectives, device configurations and image sizes.
- We compare our framework to different approaches such as Fully Local or Fully Offloaded execution and a state-of-the-art Deep Neural Network offloading Framework named Neurosurgeon [8], showing that our method outperforms the Local-only and Fully offloaded approaches, and can outperform Neurosurgeon given that i) Neurosurgeon has not pre-offloaded the weights to the edge server or ii) that we also upload the weights to the server prior to execution.

2.2 Thesis Structure

This thesis is organised into 6 chapters. In the current section, we present an outline of the chapters that follow.

In Chapter 3 we recount the current state of DNN offloading and partitioning. We analyze the various approaches being explored in the field, their methodologies and their limitations.

Chapter 4 examines the theoretical background behind the concepts that underpin this thesis, which include Deep Neural Networks as well as the layers that comprise them, Network protocols and Collaborative Filtering.

Chapter 5 contains the core methodology behind out framework. More specifically, this chapter describes in depth the Collaborative Filtering based profiling, the partitioning algorithm and the offloading architecture that we propose.

In Chapter 6 we evaluate and compare our framework to other solutions and frameworks that perform inference at the edge of the network.

Finally in Chapter 7 we summarize and report our results while also proposing future research tangents stemming from this work.

Chapter 3

Related Work

This chapter covers the state-of-the-art Deep Neural Network Offloading approaches and the results that the yield.

There exists a plethora of works covering the topic of resource management on edge computing systems for Deep Neural Networks. As mentioned previously in Chapter 2, DNN partitioning and offloading solutions are becoming increasingly popular when it comes to deploying devices and inference applications to the edge of the network. Zhou et al. [20] present the various challenges that are entailed by deploying DNN models on the edge, both in the present and in the future. Similarly, authors of [21] outline the architectures, frameworks, and emerging technologies that are involved in the training/inference of deep learning models at the network edge.

With the intention of providing Deep Neural Network management solutions, the authors of [9] introduce a partitioning-based framework that divides the DNN into individual segments that are then uploaded to an edge server. In a similar manner, authors of [22] partition a DNN model in a dynamic manner, adapting to various changes in the computational resources and the state of the network. In [10] a workload partitioning algorithm generates DNN partitioning schemes in real-time, while authors of [12] create a local network of IoT devices that collaborate on reducing the latency of a subset of NN layers, mainly convolutional ones. Finally, in [8], the authors demonstrate the efficacy of DNN processing on the cloud, based on pre-offloading of layers. We will present the methodologies behind the approaches mentioned here in more detail in the following paragraphs.

Laskaridis et al in SPINN [11] demonstrate a variety of techniques to improve inference latency in edge devices. The approach put forward combines methods such as early exits, DNN partitioning and offloading, as well as the reduction in arithmetic precision for offloaded data, aiming to decrease network transfer latency. The framework is able to vary its execution strategy in many different ways, such as offloading but continuing execution until the next early exit node (possibly cancelling the offloading request if the confidence interval is satisfactory) or locally executing the DNNs through many exit nodes until a specified confidence threshold is reached. The selection of the execution methods is derived through a dynamic scheduling method that compiles its results from previously (offline) profiling data, while also taking into account fluctuations imposed by the network state and the server load. The feasible execution solutions are ranked legixographically with the user selecting the order of the optimization objectives. Possible optimization objectives include latency, throughput, server cost, device cost and accuracy.

Kang et al [8] propose a framework that splits DNN inference between the edge and the cloud, by executing at the edge up to a layer i and offloading the rest of the computations to the cloud. This is achieved via an algorithm that selects the optimal partition point for either energy efficiency or minimal inference latency based on linear or logarithmic regression performance and energy models for the layers as well as a linear model for the network's impact. The approach presented here also takes

into account the offloading server's load levels by parameterizing it in the regression models mentioned previously. One point that has to be mentioned is that the framework being proposed in [8] assumes that the weights and parameters of the DNNs have been a priori offloaded to the cloud, thus only requiring the transmission of intermediate results for further computations.

Authors of DeepThings [5] take a different approach compared to other frameworks. Rather than partitioning a Convolutional DNN horizontally - that is partitioning between layers - they split the network's input and convolutional layers vertically in a grid, effectively parallelizing the workload in a distributed cluster of edge devices. Dependencies between adjacent grid tiles are resolved through the use of a Fused Tile Partitioning algorithm that fuses successive convolutional layers into a single task that is then partitioned in a grid fashion. This approach also reduces the memory footprint, therefore allowing inference to be performed on heavily resource constrained edge devices. After the CNN is successfully partitioned through the method mentioned above, individual tiles are registered as available for offloading at the gateway which acts as a broker, inter-mediating work stealing requests from other devices in the cluster.

MoDNN which is described in [4], introduces a spectral clustering technique to group together non-zero weights of Fully Connected layers and perform sparsification on the weight matrix, thus allowing the framework to use efficient General Matrix multiplication for these clusters and less efficient - but with reduced data transmission cost - Sparce Matrix Multiplication for elements residing outside the clusters. The resulting dense clusters are then distributed across the worker nodes. Convolutional Layers are also partitioned along their greatest dimension with the resulting partitions being offloaded to other devices in the cluster. A similar method is explored in [10] where the partitioning problem is formulated as an Integer Linear Programming (ILP) with the network time and energy being added to the ILP formulation.

In [12] various new methods towards exposing and exploiting parallelism in convolutional and fully connected layers are explored. More specifically the authors experiment with distributing the channels (channel splitting), splitting the input tensor in the X and Y axis (spatial splitting) similarly to what was shown in [5] and filter splitting where both the convolutional filters and their corresponding input channels are distributed depth-wise. Fully connected layers are also distributed via input or output splitting, that is distributing the inputs and partially calculating the generated outputs or distributing the outputs and copying the input to all devices in the cluster. Through this approach, the authors aim to achieve speedups and to limit the usage of swap space in heavily resource constrained devices such as the Raspberry Pis used in the experimental evaluation of the framework.

The concept of partitioning and offloading workloads at the edge of the network has also been studied from the perspective of other sciences. More specifically, authors of [13] explore a market-based approach for the distribution of tasks in a cluster of edge devices. In greater detail, each edge device periodically receives an a specific amount of *Money* depending on its computational capabilities, the capabilities of the gateway and the number of connected devices in the cluster, with the efficacy of the system being based on standard economic concepts such as pricing and demand. The devices are then able to offload tasks to the edge server by *Buying* computational capacity through an auction. A device may choose to offload a workload that cannot be executed locally or can be executed with smaller latency at the server. Should a device be outbidded by the competition, it defers the execution/offloading for later. Game-theory can also be applied as a resource management strategy at the edge, as shown in [14]. In this context, the problem of optimally allocating tasks to gateways is solved through the use of an auctioning mechanism where initially the offloading or not of a task is decided through a *Stackelberg competition game*. Should the task be eligible for offloading, a Vickrey (second price) auction takes place, where gateways bid their estimated values for the task, with the winner of the auction receiving the task.

Straying from the topic of DNN partitioning, it must be mentioned that the inspiration behind using Collaborative Filtering as a method for generating estimates for performance and energy stems from [15] where Delimitrou et al. utilized Collaborative Filtering techniques, in order to find the most suitable server to execute a particular workload. This was achieved by profiling the application briefly in two randomly selected servers and then feeding the results to the CF algorithm, thus allowing

predictions for their fitness to be made for the entirety of the server cluster, demonstrating that exhaustive profiling can be substituted with minimal profiling and Machine Learning without loss of accuracy.

Despite the fact that resource management, partitioning and offloading of DNNs has been heavily studied by the research community, to the best of our knowledge, no study has leveraged Collaborative Filtering techniques to create an efficient and decentralized resource management solution for NN offloading that decides at the granularity of a single layer and targets heterogeneous CPU/GPU architectures.

Chapter 4

Background on DNN architectures and layers, Network protocols, Network profiling and Collaborative Filtering

This chapter describes in depth the types of Deep Neural Networks that are currently being utilized for inference at the edge and the different layer types that they contain. It also outlines Collaborative Filtering and how it can be used outside the field that it originates from to generate predictions for large amounts of data while only being trained on a fraction of the given dataset. We also present some of the Network protocols that will later be utilized by our framework as well as their performance and energy characteristics.

4.1 Neural Network Layers

In this section we analyze some of the most famous types of Neural Network Layers.

4.1.1 Convolutional Layers

Convolutional Layers perform a convolution with a kernel K over a given input Tensor T. More specifically, a layer of this type:

- Receives an input tensor of dimensions: $N \times C_{in} \times H_{in} \times W_{out}$
- Applies a convolution with the kernel K of dimensions $C_{out} \times L \times K_0 \times K_1$.
- Returns an output tensor of dimensions $N \times C_{out} \times H_{out} \times W_{out}$

The relationship between H_{out}, W_{out}, the dimensions of the input and of the convolution kernel K is the following, assuming the parameters $Padding, Stride, Dilation$ and $Groups$ are also specified

$$H_{out} = \lfloor \frac{H_{in} + 2 \times Padding_0 - Dilation_0 \times (K_0 - 1) - 1}{Stride_0} + 1 \rfloor$$

$$W_{out} = \lfloor \frac{W_{in} + 2 \times Padding_1 - Dilation_1 \times (K_1 - 1) - 1}{Stride_1} + 1 \rfloor$$

$$L = \frac{C_{in}}{Groups}$$

The main function of the Convolutional Layers is to extract features from an image such as edges and corners. These features can then be passed on to deeper layers where they are combined to understand details in the image.

4.1.2 Normalization Layers

Batch Normalization

Batch Normalization [23] is the process of normalizing the input before a layer by centering the distribution of values in the tensor to zero and their variance to one. This modification allows Neural Networks to train faster by minimizing oscillations[1] in the weights and allowing the learning rate to be increased accordingly. This is achieved through the element-wise application of the function:

$$BatchNorm(x) = \frac{x - \tilde{x}}{\sigma(x) + \epsilon},$$

where ϵ is an arbitrarily small value to ensure that the denominator is always larger than zero. Batch Normalization was first presented in [23] as a method to increase accuracy and decrease training times, and then integrated into existing and new NN architectures such as VGG [24] and ResNet [25].

4.1.3 Activation Layers

Activation layers are functions that are applied after Neurons in Neural Networks, returning a tensor of the same dimension as their input. The role of Activation Layers is to prevent the Neural Network from having a Linear behaviour, which is undesired, as linear behaviour can easily be achieved by a simple Linear layer. For this reason they are also called non-linearities. There are many popular activation functions with the most popular being the following.

ReLU

ReLU or Rectified Linear Unit is an activation function that applies the following function to every element of the input tensor.

$$ReLU(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

As can be inferred, the ReLU function sets all negative elements of the input to zero and leaves the rest unchanged. This forms one of the main advantages of ReLU, as it is computationally inexpensive. At the same time, the fact that there are no upper limits to the return value of ReLU solves the problem of *saturation*.[2] [26]. There also exists a variant of ReLU named ReLU6 that defines a maximum activation value of 6, with it being popularized in the MobileNet Neural Network architecture [27].

Softmax

Softmax is an activation layer defined by the following function, when given an input tensor x:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

As can be seen, by summing all the Softmax activations in a tensor, the result would be 1. This is because Softmax is frequently used to generate the probability distributions in the final layer of Neural Networks, which of course must sum to 1. A similar type of activation is Softmin, which is defined as:

[1] Fluctuations in the parameters of neural networks, due to loss rapidly shifting the weights in increasing and decreasing directions.

[2] Neural Network nodes frequently returning values close to an asymptote.

$$Softmin(x_i) = \frac{e^{-x_i}}{\sum_j e^{-x_j}}$$

Softmin once again has the property of summing to 1 for all the elements in a given tensor.

4.1.4 Pooling Layers

The role of pooling layers in a Neural Network is to decrease the number of parameters deeper in the Neural Network by reducing the dimensionality of their input. Pooling layers downsample the feature maps given to them as inputs, allowing some degree of invariability if features are relocated to a nearby section of the input. Some of the frequently used pooling layers are the following.

Maximum Pooling

Maximum Pooling creates a grid over the different channels of the input feature map, with each grid square being of dimensions equal to the kernel size of the Maximum Pooling layer, returning the maximum value in each grid square. More specifically, assuming a tensor X of dimensions $N \times C \times H_{in} \times W_{in}$ and a kernel of dimensions $kH \times kW$, Maximum Pooling returns an output tensor of dimensions $N \times C \times H_{out} \times W_{out}$, where:

$$H_{out} = \lfloor \frac{H_{in} + 2 \times Padding_0 - Dilation_0 \times (K_0 - 1) - 1}{Stride_0} + 1 \rfloor$$
$$W_{out} = \lfloor \frac{W_{in} + 2 \times Padding_1 - Dilation_1 \times (K_1 - 1) - 1}{Stride_1} + 1 \rfloor$$

The output of every channel is given by the function:

$$MaxPool(i, j, h, w) = max_{m=0,1,\ldots,kH-1} max_{n=0,1,\ldots,kW-1} X[i][j][stride_0 \times h + m][stride_1 \times w + n]$$

Average Pooling

Average Pooling behaves similarly to Maximum Pooling in terms of input and output dimensions, but instead of returning the maximum element in every grid it returns an average of all the elements in the grid. This input output relationship is described by the function:

$$AvgPool(i, j, h, w) = \frac{1}{kW \times kH} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} X[i][j][stride_0 \times h + m][stride_1 \times w + n]$$

4.1.5 Fully Connected Layers

Fully Connected or Linear layers are the first type of layers to be used in artificial Neural Networks, multiplying all the outputs of the previous layer with weights and adding biases. The input - output relationship of a Linear layer can be described as

$$y = x \cdot A^T + b,$$

where x is the input tensor, matrix A contains the weights of the Linear layer and matrix b the biases.

Linear layers display high computational complexity due to the matrix multiplication operation and occupy significant memory resources in order to store all the weights. As linear layers create outputs based on the entirety of the input, they are frequently used in the last layers of Neural Networks, where final predictions are made.

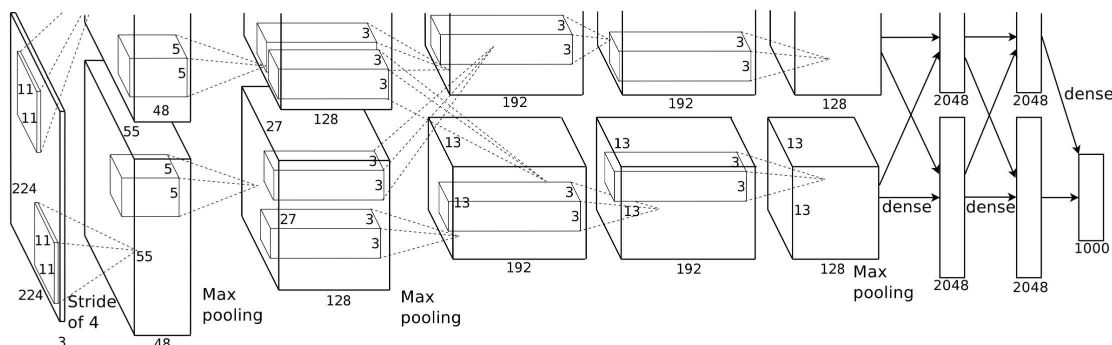


Figure 4.2.1: Alexnet Architecture deployed on two GPUs, from the original paper [26]

4.2 Neural Network Architectures

Deep Neural Networks form the backbone of many inference tasks in today's day and age, with Convolutional Neural Networks gaining particural traction after the introduction of AlexNet [26]. As the never-ending quest towards higher accuracy and lower inference time continues, novel Neural Network Architectures emerged with VGG [24] and ResNet [28] being standouts. At the same time, as inference tasks grew more common in the resource-constrained mobile and edge environment, the need to decrease the computational requirements became increasingly more apparent, with MobileNets [27] becoming a popular choice for such scenarios.

In this section we will analyze the architectures of the Neural Networks mentioned above and their unique characteristics. Furthermore we will present the characteristics of their layers along with their computational and energy requirements.

4.2.1 AlexNet

AlexNet, first presented in 2012 was the first mainstream Deep Convolutional Neural Network to win the ImageNet Large Scale Visual Recognition Challenge (LSVRC). While the idea of using Convolutional Neural Networks for Image Recognition had been presented much earlier [29], it was the utilization of GPU training that made such Deep Networks viable and ultimately enabled this breakthrough. The architecture and logic behind AlexNet is the following.

Architecture

In the general sense, AlexNet is a Convolutional Neural Network consisting of 8 Convolutional Layers and and 3 Fully Connected layers. Each Convolutional layer [3] is followed by a ReLU non-linearity. Also MaxPooling layers are placed between the first and second, second and third and between the fifth and sixth layers. The Fully Connected layers are located after the Convolutional ones, of which the first two are also followed by a ReLU layer, while the third and final one is followed by a SoftMax function that produces the probabilities for each possible class.

As can be seen in Fig. 4.2.2, the Computational and Energy requirements are high for the Convolutional and Linear layers, with the highest energy and latency being found in the first Linear layer. Size-wise, the largest layers are the Linear ones at around 100MB. This somewhat large size plays an important role in our offloading decisions as sending hundreds of MBs can be inefficient both for energy and latency. On the contrary, the Convolutional layers with their reduced size and comparable latency to the Linear ones can be more fitting candidates for offloaded execution, offering similar computational intensity without needing to transmit large amounts of data.

[3] in the inference configuration where Dropout layers are bypassed.

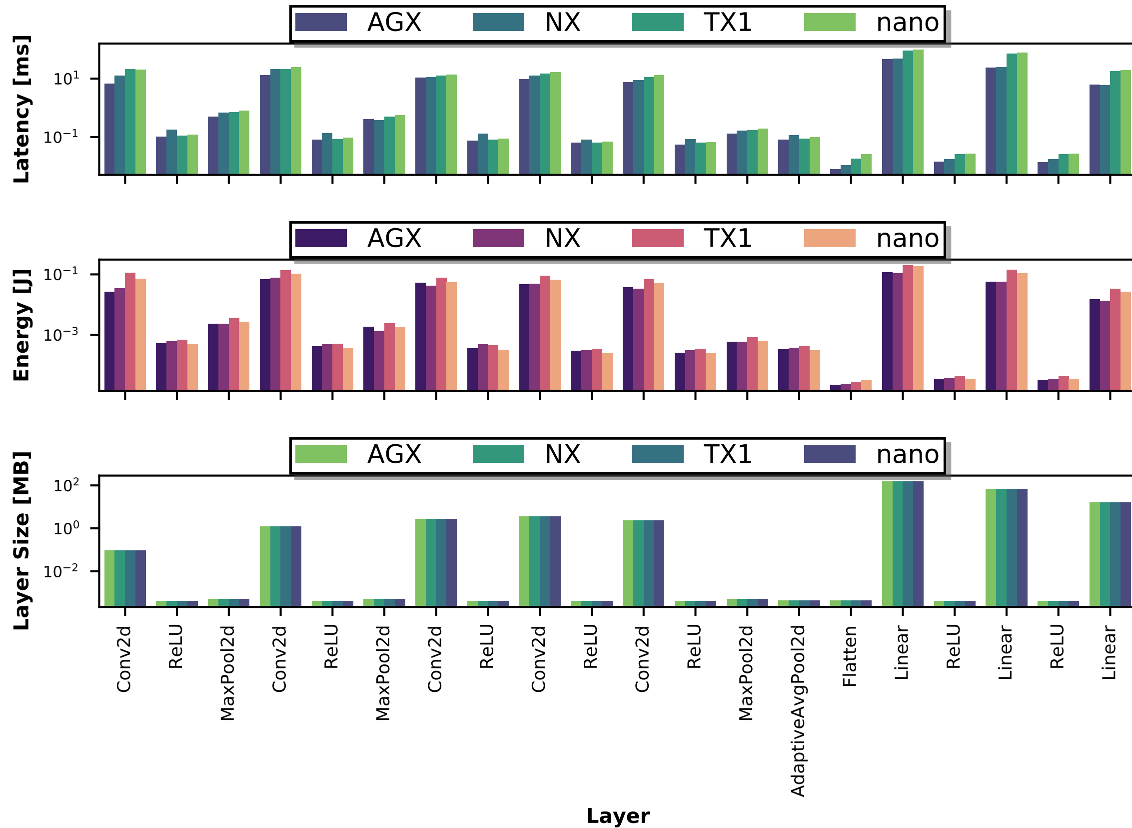


Figure 4.2.2: AlexNet Latency and Energy per layer, grouped by device

4.2.2 VGG

VGG, first presented in 2014 defines a set of Neural Network Architectures where the number of layers containing weights, that is Convolutional and Fully Connected layers varies from 11 to 13,16 and 19.The different configurations of VGG can be summarized as follows: VGG is comprised of 5 **blocks** of convolutional layers, and 3 Fully connected layers. The layers that reside in these blocks are configured based on the selected VGG architecture, as shown in the table of Figure 4.2.3.

Each Convolutional layer is followed by a ReLU non-linearity, with each block of layers passing its results through a Maximum Pooling function [4]. The Fully Connected layers are applied after the convolutional blocks with each one of them having 4096, 4096 and 1000 output features respectively (for the classification into 1000 classes). Finally, the classifier probabilities are calculated via the application of a SoftMax function at the output of the final Fully Connected layer. The architecture of VGG11 can be seen in Fig. 4.2.4 .

In Fig. 4.2.5 it can be seen that most of the time and energy in VGG11 is spent while calculating the middle Convolutional layers - the ones that are contained inside the blocks - and the final Linear layers, with the latter ones having large layer sizes, as was the case with AlexNet. In the same fashion as previously, the algorithm that we describe later on takes these parameters into account and usually avoids sending these large layers, as to not incur the high Network penalties. Similar behaviour is also demonstrated by the other members of the VGG family of networks, with partitioning being performed in a comparable manner.

[4]It is also common in some non-standard implementations to include Batch Normalization between each convolution and ReLU, as Batch Normalization was discovered after VGG was initially released

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 4.2.3: VGG Network Configuration Table, as proposed in the original paper [24]

4.2.3 ResNet

Architecture

ResNet, winner of the LSVRC competition in 2015, popularised the concept of Residual Neural Networks by introducing skip-connection in layer blocks, effectively applying functions of the form $F(x)+x$ on a given block of layers F with an input tensor x. The resulting Neural Networks perform accurately on image classification and segmentation tasks, while demonstrating reduced computational complexity compared to the previously mentioned VGG. At the time of its inception, ResNet was 8× deeper than VGG, standing at 152 weighted (Convolutional and Fully connected) layers deep in its most complex configuration, with other possible configurations featuring 18,34,50 and 101 weighted layers respectively.

The profiling results for ResNet18 (shown in Fig 4.2.7) display a different behaviour to that shown by AlexNet and VGG. Here, the vast majority of the computational burden is located in the ResNet Basic Blocks with the configurations of these blocks, being displayed in Fig. 4.2.6. As opposed to the architectures seen before, the layer weights are comparatively small, making offloading viable even for the Linear layer at the end of the network. As will later be seen in the evaluation of our framework in Fig. 6.3.4, ResNets exhibit some of the highest offloaded execution percentages from the architectures that we examined. This behaviour is shared across the board for all ResNet architectures including the ResNet18 one that is shown in the figure.

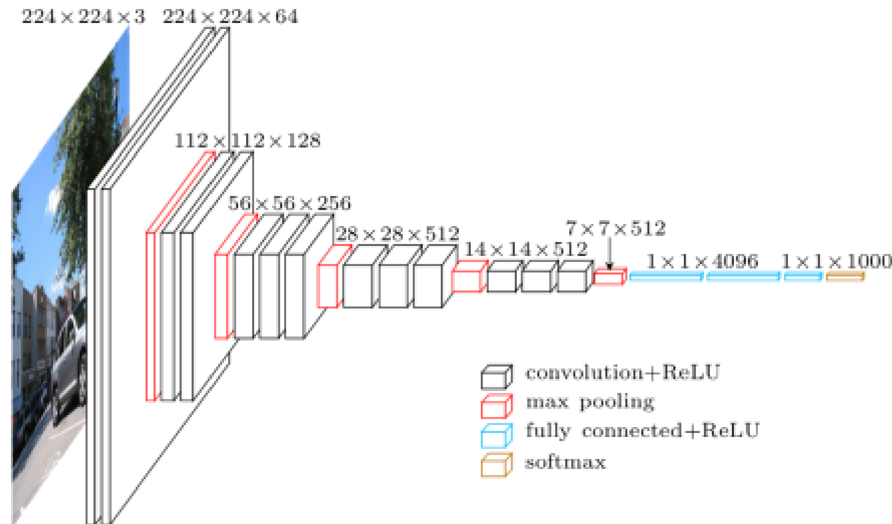


Figure 4.2.4: VGG Architecture, figure by Davi Frossard

4.2.4 MobileNetV2

As mentioned previously the demand to perform inference at the edge of the network gave rise to architectures with reduced computational complexity, with one of the proposed architectures being described in the MobileNets paper [27]. The proposed Neural Networks are tuned through two hyperparameters, α which reduces the computational effort at every layer level and ρ which defines the resolution of the input images, thus allowing the deployment of inference applications on environments with varying computational capabilities and latency requirements. The proposed architecture is a Residual Neural Network, implementing a class of layers known as Inverted Residual, that reduce computational complexity as compared to the typical Residual blocks found in other architectures.

Similarly to ResNet, most of the latency and energy consumption is aggregated in the Residual Blocks of the MobileNet architecture. Layer sizes are particularly small as seen in Fig. 4.2.8 being usually under the size of a MB and reaching sizes of up to a few MBs. This makes offloading particularly effective and contributes to the very high offloading percentages seen in 6.3.4, well over 99% for all cases.

4.2.5 Comments on Residual Architectures

As far as offloading and partitioning is concerned, Residual Architectures pose an interesting challenge due to the possibility of having more than one layer inputs, introducing dependencies that must be resolved when offloading to another device.

This can be seen in Figure 4.2.9 as the input X is required for offloading from Layer 2 and onwards due to the existence of the + operator. A sophisticated approach to this issue would be to create a Directed Acyclic Graph modelling the dependencies between different Neural Network operations, which can then be used to dynamically detect the dependencies and transferring the required data accordingly. Such an approach was utilized by the SPINN framework [11] that was mentioned earlier. Another approach, which is the one that we utilize, is to limit the granularity of the partitioning, essentially maintaining the illusion of having only one input and one output per layer. Essentially, residual blocks can be marked as indivisible, with their entire payload of layers being offloaded together, hiding in this way the aforementioned dependencies.

Figure 4.2.5: VGG11 Latency and Energy per layer, grouped by device

4.3 Collaborative Filtering

In this section we will discuss the motivation behind the development of Collaborative Filtering methods and their advantages compared to other methods.

4.3.1 Brief Description

Most machine learning methods make decisions based on a set of features from the input. Collaborative Filtering changes this by not examining features, but rather extracting information about a set of `users` and `items`. As a result, Collaborative Filtering can be a powerful tool in recommender systems, being the go-to algorithm for the Netflix Prize [5] due to its ability to make item recommendations to users given only a set of ratings from other users. The ability of Collaborative Filtering to provide accurate recommendations was famously used for assigning workloads to servers by Delimitrou and Kozyrakis in PARAGON [15], thus proving its efficacy in the domain of Resource Management for Computing applications.

4.3.2 Algorithms

Collaborative Filtering encompasses three main types of algorithms:

- Memory-based approaches: Predictions are generated by finding similarities between users or data and proposing the best-suited items.

[5] 1,000,100 $ prize for the best algorithm predicting user ratings for films, without information about the users or films.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×8
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23	[1×1, 256 3×3, 256 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 4.2.6: Different Configurations of ResNet architectures, from the original paper [25]

- Model-based approaches: Known ratings are used to train a model and then predict unknown ratings.
- Hybrid approaches: Combinations between the Memory-based and Model-based methods.

Memory-based Collaborative Filtering

The Memory based approach can further be broken down into to main subcategories, **user-based** where similarities between users guide the recommendation and **item-based** where the recommendations are driven by the similarities between items. In essence, the user-based method generates a vector containing the known values for items, while the item-based methods create vectors containing values from all the users that have *seen* the item. Similarities between vectors can the be found given some vector distance metric. Famous such metrics as mentioned in [30] are:

- Vector Cosine Similarity: $\frac{\vec{A}\cdot\vec{B}}{||\vec{A}||\cdot||\vec{B}||}$
- Pearson Correlation: $\frac{\sum_i (r_{ai}-\tilde{r_a})(r_{bi}-\tilde{r_b})}{\sqrt{\sum_i (r_{ai}-\tilde{r_a})^2 \cdot \sum_i (r_{bi}-\tilde{r_b})^2}}$
- Conditional Probability-based similarity

In the user-based approach, the k unseen items are recommended as the top unseen items from the k most similar users. In the item similarity approach, after finding the similarities between items, recommendations are made to the users based on the items that are most similar to the ones that have already been seen by them.

The main advantages of the Memory-based methods is that it is an intuitive approach - similar users usually like similar items and similar items might interest the same user - and their ease of development, due to the fact that the only thing that is needed is a distance metric. However, this simplicity can be a weakness when the data is too sparse as is usually the case with the algorithms providing low accuracy recommendations. It also fails to satisfy our use case, which is predicting ratings (or energy/latency values in our case).

Model-based Collaborative Filtering

As mentioned previously, in the Model based approach, the known ratings-values are used to train a model that can the be used to provide recommendations. Such models include Bayesian Belief Networks, Clustering and methods based on Matrix Factorization namely Singular Value Decomposition

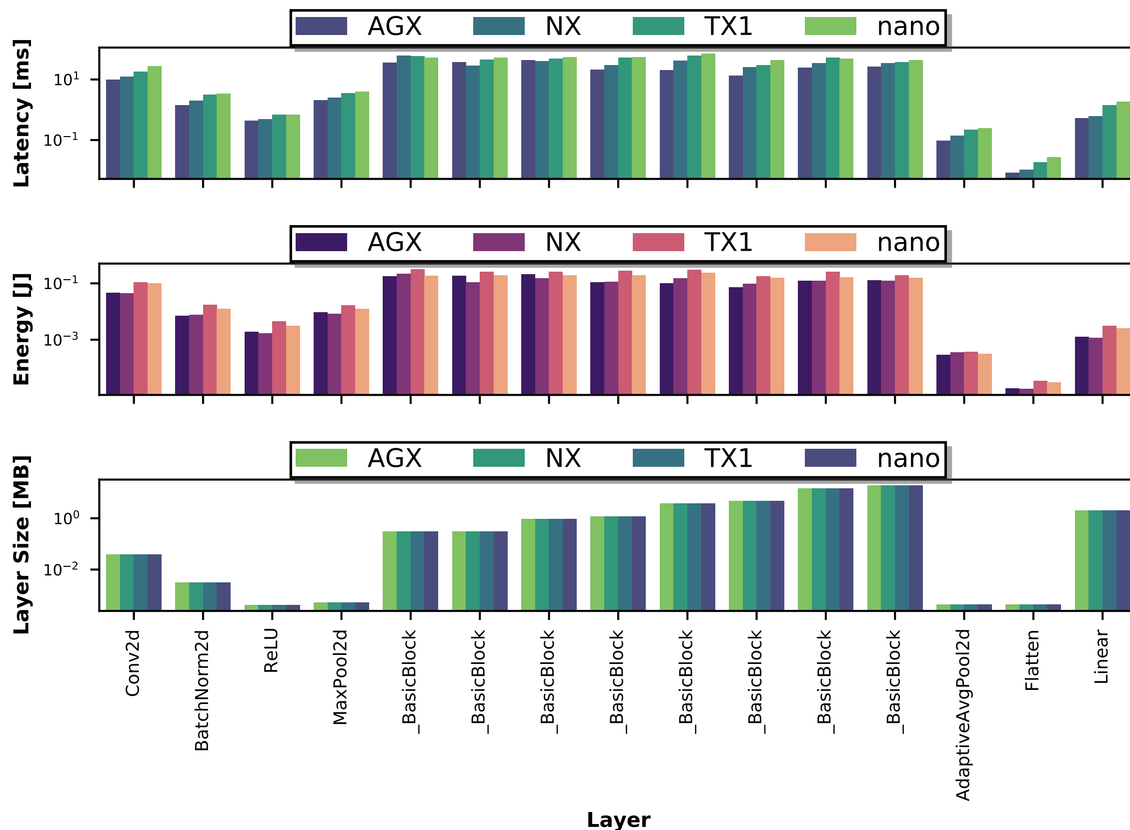


Figure 4.2.7: ResNet18 Latency and Energy per layer, grouped by device

(SVD) and its variants, with the latter being used in Paragon. The main attribute of Model Based Collaborative Filtering is that instead of predicting items, it predicts ratings of items, that can then be used to generate recommendations. The advantage of such methods is that they deal with the main drawback of Memory based approaches, as they can provide accurate recommendations even in sparse datasets, also allowing for more configurability. They do however lack the intuitive sense of Memory based methods, relying on hidden features/embeddings (low dimensionality interpretations of our original high-dimensional input data).

4.3.3 Matrix Factorization Collaborative Filtering for Predicting Latency and Energy

As seen in subsection 4.3.2, model based Collaborative Filtering methods can be used to predict user ratings for different items. We can use this ability to predict values other than ratings if the problem can be formulated as a user-item relationship. In our approach, we would like to predict the latency and energy consumption of Neural Network layers on different devices. Different devices have different computational capabilities and as such they have different energy consumption, and performance, in the same manner that different users have different preferences. Similarly, different Neural Network layers have different computational requirements (i.e., high memory requirements, computational performance requirements etc.) that impact their performance on different devices, behaving like items in the Collaborative Filtering model of thought.

Having established the user-item relationship between devices and layers, the only parameter missing before selecting and training a model is the rating. This of course, is any value that we have partial

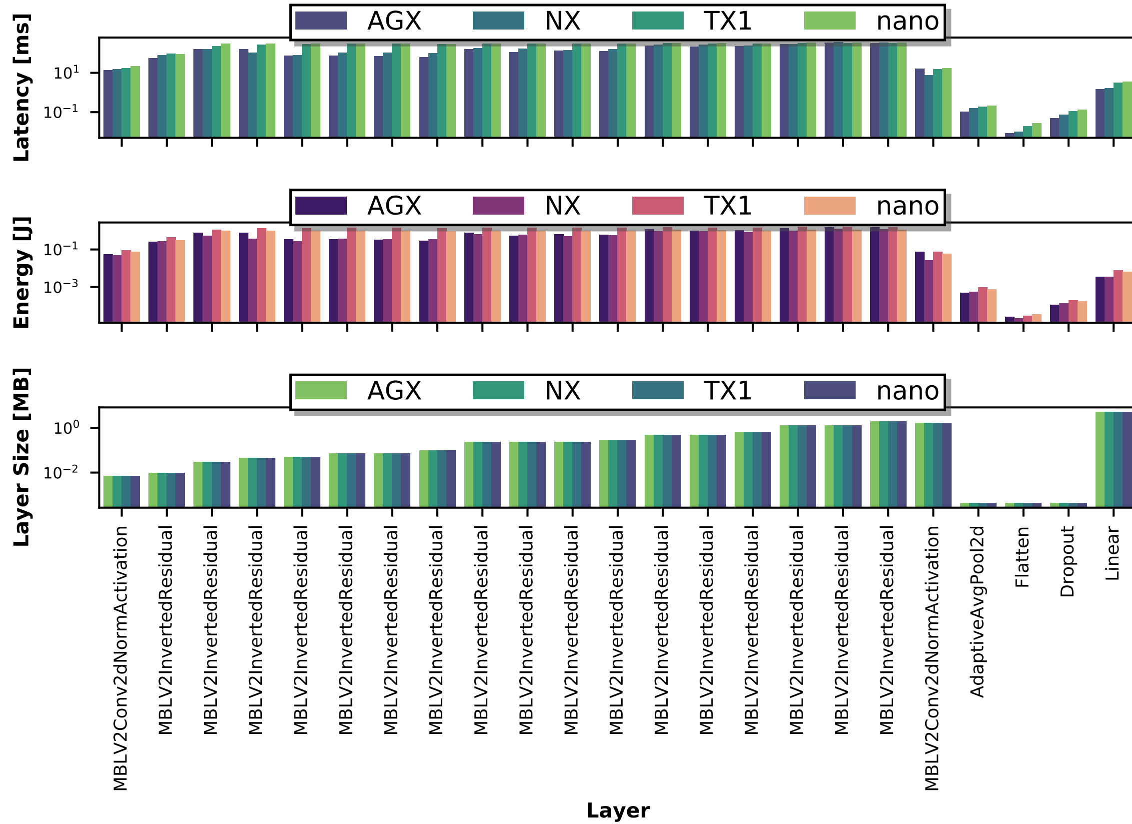


Figure 4.2.8: MobileNetV2 Latency and Energy per layer, grouped by device

knowledge of and that we would like to predict. Thus we train two different CF models, the one having the latency of a layer as the rating parameter and the other having the energy of the layer. We select as our algorithm a simple Gradient Descent based Matrix Factorization, a Model based approach. There are two main reasons behind this approach. Firstly, as shown in [15], Matrix Factorization approaches can achieve high accuracy in very sparse datasets (as low as 1% of the Rating Matrix being filled). Secondly, they can be programmed to achieve high computational performance and low latency [15], something that is critical for the online part of our algorithm.

4.4 Network Communication Protocols

This section gives an overview of the two protocols that are utilized by our framework in order communicate and transfer data between the edge devices and server. These protocols are ZeroMQ and FTP respectively.

4.4.1 ZeroMQ Networking Library

ZeroMQ is a lighweight messaging library supporting different protocols and communication patterns that can be interfaced with through most famous programming languages such as C/C++, Java, Python, Haskell and others. In its core, ZeroMQ presents a socket-based API through which the different communication patterns are supported. These patterns or models each introduce their own socket types that can be used to implement the needed communication schemes.

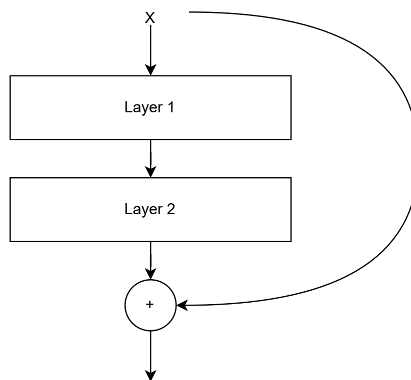


Figure 4.2.9: Example of a Residual Block

Request - Reply model

In the synchronous Request - Reply model, a REQ (request) socket operated by the client connects to a REP (reply) socket which is listened to by a server, with the server then responding to the client's request. The same mode of communication can be established in an asynchronous manner, meaning that there is no need for a preset communication pattern (both the client and server can send messages to each other at any time without restrictions). In this context the client-side socket is named DEALER and the server-side ROUTER.

Publish - Subscribe model

The Publish - Subscribe model establishes a network in which a set of Subscriber devices listen to and receive messages from a set of Publisher devices. The subscriber socket can be of type SUB or XSUB, with the difference between the two being that SUB listens to all messages sent by a publisher (PUB) socket, whereas XSUB sends subscription messages to XPUB sockets in order to subscribe to them. The Publish - Subscribe paradigm is useful in many-to-one situations where a single device must transmit the same message to many other devices.

Pipeline model

The Pipeline model is characterized by two socket types, PUSH, being a send-only socket and PULL being a receive-only one. This mode of communication is practical in applications dealing with the distribution of tasks in a cluster of worker nodes, with the PUSH sockets transmitting data to anonymous worker nodes in a Round Robin manner to ensure fairness. The anonymity of the worker nodes also allows new nodes to join, thus ensuring the scalability of the cluster at runtime.

Exclusive Pair model

The Exclusive Pair communication pattern is a special type of socket with the name PAIR, facilitating the communication between devices with a common architecture. As a result, it is typically used for communication between threads in a single application and frequently utilizes the `inproc` transport protocol.

4.4.2 FTP Protocol

The File Transfer Protocol (FTP), described in RFC 959 [31], is an Internet Protocol outlining the transfer of files between computers and a server. More specifically, a computer acts as an FTP server where files can be uploaded and retrieved from by users connecting to it (also known as clients). This connection can be either authenticated, though unsecure due to the passwords being transmitted in

plaintext [6], or anonymous, meaning that any user can login. FTP establishes a set of commands that can be used to connect, upload and retrieve data to and from the server, as well as performing many more complex functions. Some of the basic FTP commands are described here.

FTP open

`open` connects the current FTP session to a server with the command `open <addr><port>`, where `addr` is the address of the server that we wish to connect to and port is the port on said server. The `port` parameter can be left unspecified, in which case it is assumed to be the *well-known* port 21. After typing the `open` command the user will be asked for identification, logging in with password authentication, or as an anonymous user by typing `anonymous`.

FTP quit

`quit` finishes an FTP session, by simply typing the command to the FTP terminal.

FTP stor

The FTP `stor` command uploads a file from the local machine to the server by typing `stor <filename>`. The file is then uploaded to the current working directory on the server, which can be changed by the `cwd` command.

FTP retr

In a manner similar to the `stor` command, FTP `retr` retrieves a given file from the current working directory on the server with the syntax `retr <filename >`.

FTP ascii and binary

FTP supports transfering files with different encodings with the most common being ASCII for text files and Binary which transfers each byte in a file. These encoding modes can be selected by using the `ascii` and `binary` commands respectively. Other modes are also supported but are not widely used (i.e., support for machines that have byte sizes different than 8 bits).

4.5 Network Profiling

In order to make accurate partitioning and offloading decisions we need to make accurate predictions about the necessary time to send and receive data through the network. As will be later described, we have created a Network profiling mechanism, which is accompanied by a Network latency and energy prediction mechanism that makes the actual predictions for the cost of transmitting data over the network. We present the data points accumulated by the profiling tool that can be used to gauge the latency and energy for transmitting data over FTP. The datapoints from Fig. 4.5.1 will later be used to generate the models for the prediction mechanism.

As can be seen from the figure mentioned previously, the profiled devices display similar behaviour in general. First of all, there is minimal latency and energy for transmitting and receiving data less than a MB in size. In fact, the time to perform such transmissions is so trivial that our profiling for energy returned 0 Joules, even after averaging for many runs [7]. An interesting phenomenon occurs for the TX1 and Nano devices - both boards display increased energy consumption for the same volume of data sent when compared to the others (AGX and NX)- . This is due to the fact that these devices are equipped with a relatively small 4GB of RAM [8] as opposed to 32GB for the AGX and 8GB for the NX. As a result when sending and receiving large sizes of data, swap space must be used to hold

[6] This can be addressed by using SFTP or other secure File Transfer Protocols if security is an issue.

[7] If the duration of an event is small, measuring energy for it can be difficult as the energy tool can only work accurately at a frequency lower than 10 Hz

[8] Detailed specifications for the devices can be found in 6.1

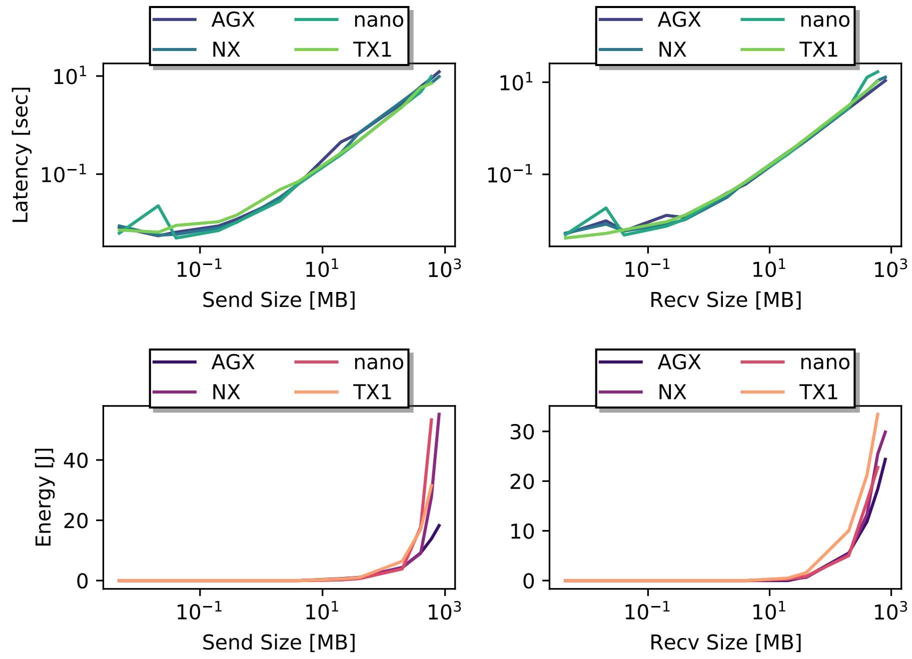


Figure 4.5.1: Network performance and energy metrics per device

temporarily some of the other pages in memory. The small onboard memory can also be a problem when trying to transmit very large segments of data, e.g. attempting to send 4GB of data on these devices causes an out of memory error. This is evident in Fig. 4.5.1, as the last two datapoints are missing for the TX1 and Nano devices in all of the plots.

Chapter 5

DNN Partitioning and Offloading

This chapter details our efforts toward developing a framework that generates a set of Pareto optimal Deep Neural Network partitionings, trading-off between latency and energy. This is accomplished with minimal profiling through the use of Collaborative Filtering and the implementation of an $O(N^2)$ Design Space Exploration algorithm that takes into account the estimated energy and latency of DNN layers and the network.

5.1 Problem Description

As shown previously in 2.0.1a, there exist many different ways to partition a Neural Network in order to achieve a specific performance-energy target. Of these many ways, some designate a Pareto Frontier where a one of these two targets cannot be further improved without sacrificing the other. We aim to discover and explore this set of solutions.

5.2 Proposed Methodology for DNN partitioning/offloading

In order to make Pareto Optimal partitioning and offloading decisions we need to gather metrics for the following parameters:

1. The processing time and energy for the computation of different DNN layers on the edge.
2. The latency and energy cost of transmitting data over the network.
3. The performance of inference calculations on the offloading machine.

Of the steps mentioned above, aggregating information about the energy of layers on a machine is the step that requires the most time. As such, we limit the number of layers profiled to a small percentage of the total layers, opting instead to use Machine Learning techniques, namely Collaborative Filtering, to infer the rest of the results. Steps 2 and 3 are trivial as far as profiling time is concerned and do not require the implementation of such solutions.

Once the required data are aggregated, they are fed into an algorithm that explores the different layer partitioning options. This algorithm explores all available partitioning options where DNN layers are first executed locally up to a layer i, calculation of layers $i+1$ to j is offloaded to the edge server and finally layers $j+1$ are calculated once again locally. The resulting configurations are analyzed for Pareto optimality in relation to energy and latency. Fully local and fully offloaded execution remain as options if they are part of the generated Pareto set.

This approach effectively splits our framework into two segments. One that runs offline and an online part. The offline part is responsible for the profiling of the layers and the network, while the online

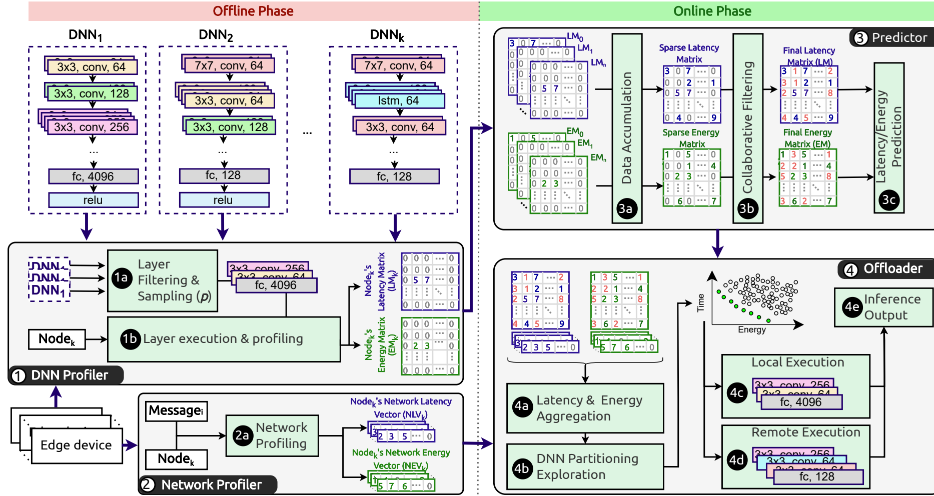


Figure 5.2.1: Overview of Online and Offline RoaD RuNNer Architecture.

is responsible for initially gathering, executing the Collaborative Filter and scattering the generated data. Afterwards, it generates the Pareto optimal set and executes the DNN partitioning/offloading based on schemes from the Pareto set and specified energy/latency targets.

5.2.1 Offline Phase

The *Offline Phase* is comprised of two distinct submechanisms: (i) the *DNN Profiler* (❶) and (ii) the *Network Profiler* (❷), that are responsible for generating the necessary data and information to be inputed to the Online part of our framework. As input to the *Offline Phase*, we provide the possible DNN architectures. This is done for every node in our edge computing network.

DNN Profiler

Taking advantage of the inherently distributed nature of our edge computing cluster, and utilizing the heterogeneity of devices and DNN models to our advantage, we design a Collaborative Filtering based mechanism [15], [16], that enables our framework to accurately predict the per layer execution latency and energy of DNN layers on every device. The first stage of this process of offline decision making, incorporates a *Layer Filtering and Sampling* (1a) component, that filters and samples the layers of the alternative DNN models that are given as input to our framework . In greater detail, we randomly select a small percentage p of the input layers in each model for in-depth profiling on the edge device. By selecting only a small percentage we can make sure that the profiling times do not become prohibitively large when the number of layers and DNNs increases. The selection percentage p is derived from experimentation.

Following this stage, the selected layers the are fed as input to the *Layer Execution & Profiling* (1b) step.

In order to gather accurate data about the energy consumption of each profiled layer we utilize the `tegrastats` utility [32] that provides information about power consumption, CPU and GPU temperatures, memory utilization and clock frequencies of different components across the device (Example log given in Listing 5.1) [1]. We set the `tegrastats` refresh rate to $100ms$ as values smaller that that cause noticeable interference and degradation of the generated results. Due to the relatively short duration that many layers take for their computation (i.e ReLU, BatchNorm etc.) we repeatedly run each layer until a set time limit has been exceeded. We set this limit to $300ms$ so that there exist at

[1] We define as power consumption the dynamic power consumption, that is $Power - Power_{idle}$, where $Power_{idle}$ is measured with the device idling with the GPU turned off

least 3 `tegrastats` log entries per profiled layer. At the same time, we measure the total time it took to profile the layers as well as the number of runs that were performed during the profiling. From the data gathered we can:

1. Measure the average time taken to calculate a layer by dividing the total duration with the number of runs performed.
2. Calculate the energy consumption of the layer by integrating the reported power from `tegrastats` with the total time and dividing by the number of runs.

```
RAM 703/7773MB (lfb 1398x4MB) SWAP 0/3887MB (cached 0MB)
CPU [1%@1907,0%@1907,0%@1907,0%@1907,off,off] EMC_FREQ 0%@1600 GR3D_FREQ 0%@510
APE 150 MTS fg 0% bg 0% AO@41C GPU@41C PMIC@50C AUX@41.5C CPU@43C thermal@41.8C
VDD_IN 2772/2772 VDD_CPU_GPU_CV 285/285 VDD_SOC 1059/1059

RAM 703/7773MB (lfb 1398x4MB) SWAP 0/3887MB (cached 0MB)
CPU [0%@1907,0%@1907,0%@1907,0%@1907,off,off] EMC_FREQ 0%@1600 GR3D_FREQ 0%@510
APE 150 MTS fg 0% bg 0% AO@41C GPU@41C PMIC@50C AUX@41.5C CPU@42.5C thermal@41.8C
VDD_IN 2772/2772 VDD_CPU_GPU_CV 285/285 VDD_SOC 1059/1059

RAM 703/7773MB (lfb 1398x4MB) SWAP 0/3887MB (cached 0MB)
CPU [1%@1906,0%@1907,0%@1907,0%@1907,off,off] EMC_FREQ 0%@1600 GR3D_FREQ 0%@510
APE 150 MTS fg 0% bg 0% AO@41C GPU@41C PMIC@50C AUX@41.5C CPU@42.5C thermal@41.8C
VDD_IN 2812/2785 VDD_CPU_GPU_CV 326/298 VDD_SOC 1059/1059

RAM 703/7773MB (lfb 1398x4MB) SWAP 0/3887MB (cached 0MB)
CPU [2%@1907,1%@1907,2%@1907,1%@1907,off,off] EMC_FREQ 0%@1600 GR3D_FREQ 0%@510
APE 150 MTS fg 0% bg 0% AO@41C GPU@41C PMIC@50C AUX@41.5C CPU@43C thermal@41.8C
VDD_IN 2935/2822 VDD_CPU_GPU_CV 448/336 VDD_SOC 1059/1059

RAM 703/7773MB (lfb 1398x4MB) SWAP 0/3887MB (cached 0MB)
CPU [3%@1907,0%@1907,0%@1907,0%@1907,off,off] EMC_FREQ 0%@1600 GR3D_FREQ 0%@510
APE 150 MTS fg 0% bg 0% AO@41C GPU@41C PMIC@50C AUX@41.5C CPU@43C thermal@41.65C
VDD_IN 2853/2828 VDD_CPU_GPU_CV 366/342 VDD_SOC 1059/1059

RAM 703/7773MB (lfb 1398x4MB) SWAP 0/3887MB (cached 0MB)
CPU [1%@1907,0%@1907,0%@1907,0%@1907,off,off] EMC_FREQ 0%@1600 GR3D_FREQ 0%@510
APE 150 MTS fg 0% bg 0% AO@41C GPU@41C PMIC@50C AUX@41.5C CPU@43C thermal@41.65C
VDD_IN 2812/2826 VDD_CPU_GPU_CV 326/339 VDD_SOC 1059/1059
```

Listing 5.1: Tegrastats Log Example for Jetson NX

The set of sampled layers are profiled locally on each node, based on the process analyzed above. The resulting profiling information for an edge node k is the Node's Latency Matrix (LM_k) and Node's Energy Matrix (EM_k). Both of these matrices are forwarded to the *Predictor* mechanism that is described in detail in the *Online* part of our approach (Section 5.2.2), where they will be further processed.

Network Profiler

A common aspect of edge computing systems is their ability to operate utilizing the existing network infrastructure. When offloading decisions must be made, as discussed previously both in Chapter 2 and the beginning of Section 5.2, it is vital to accurately estimate the overhead imposed by the transmission of data over the network. As a result, we perform profiling of the Network for energy and latency (both for the transmission and the reception of data) utilizing an extended *Network Profiling* mechanism (2a). Our mechanism profiles the transmission and reception of data ranging from a few KB to a few GB in accordance to what was described in 4.5. The resulting datapoints of this profiling can be seen in Fig. 4.5.1.

When the profiling step has been completed, we generate a set of two vectors. The The Network Latency Vector (NLV_k) contains datapoints correlating the size of a transmitted or received message to the latency of the transmission, while the Network Energy Vector (NEV_k) represents the energy required for the transmission (or reception) of alternative message sizes. These two vectors are used

to generate fourth-order polynomial curves for the associated data, with the polynomials later being used during the *Online Phase* for estimation of the network's overhead. Each device creates its own set of Polynomial parameters, tailored to its unique network connection characteristics.

5.2.2 Online Phase

The high dynamicity of the edge computing environment creates the need for our framework to make decisions in a run-time manner. Addressing this issue are the following two components that make up the core of our methodology.

- *Predictor* (❸): Responsible for dynamically predicting latency and energy per layer.
- *Offloader* (❹): Responsible for network and Collaborative Filtering data aggregation, dynamic DNN partitioning and offloading.

Predictor

The Node's Latency Matrix (LM_k) and Node's Energy Matrix (EM_k) that were produced during execution of the *DNN Profiler* (❶) for every node k are aggregated on the cloud server (3a). The transmission of the matrices is performed over FTP. By combining all of the LM_ks into a single matrix and all of the EM_ks into another one, we generate two new matrices that span the entirety of our edge devices, *Sparse Latency Matrix* and the *Sparse Energy Matrix*, respectively. As their name would suggest, these matrices are sparse due to the fact that their fill percentage is regulated by the p parameter, that was defined in the *DNN Profiler*. Each time the sparse matrices are generated, a run of the Collaborative Filtering algorithm is performed (3b), generating estimates for the unknown values and creating the *Final Latency Matrix(LM)* and the *Final Energy Matrix(EM)*, containing the energies and latencies for all the DNN layers on all the edge devices. Once these matrices are generated, the edge devices are notified by the server to retrieve their individual results, containing accurate estimates for the latency and energy of the DNN layers for that specific edge device. Contrary to prior works, which depend on extensive profiling of the DNN models [9], [11], our approach is adaptable to dynamic scenarios. New nodes joining the network are integrated in the existing latency and energy matrices and the sparse matrices are updated, triggering re-runs of the mechanism. Furthermore, new incoming Neural Network models can benefit from layers already in the CF matrices.

5.3 Collaborative Filtering Mechanism

As shown in subsection 4.3.3, Collaborative Filtering can be used to infer unknown values in a sparse user-item style Matrix. Originally the algorithm was implemented in Python with the help of NumPy, achieving the required accuracy for our application. However, the comparatively long time required for training (approximately 30 minutes) was unacceptable for its usage in the Online Mechanism shown earlier. To counteract this issue, the critical path of the algorithm was rewritten in `C++` at first and then accelerated/parallelized with the usage of `OpenMP` [18] to take advantage of our multicore edge server. These modifications reduced the training time to only 4 seconds, which is low enough to be used in the Online scenario.

Our collaborative filtering is based on matrix decomposition, i.e. based on the factorization of each matrix into a product of matrices. At first the sparse matrix R is filled with the known values, that is the few layers that were profiled on a specific device. Element $R_{i,j}$ contains the value (energy or latency depending on what we want to infer) of device i for layer type j. For the values that have not been derived through profiling, the corresponding cells in the matrix are filled with zero. From the above, it can be inferred that matrix R is of dimensions $M \times N$, where M is total number of devices and N the different types of layers encountered in the different Neural Networks that we have implemented. We introduce two new matrices, P and Q with dimensions $M \times K$ and $K \times N$ respectively, where K is the number of features in our Collaborative Filter, derived through experimentation. Matrices P and Q

Algorithm 1: DNN Partitioning Algorithm

```
1  Partition(network):
2      for layer in (0,...,N-1) do
3          /* predict local for layers 0 to layer */
4          l1=predictLocal(0, layer)
5          for j in (layer+1,...,N-1) do
6              /* predict network,cloud for layers i+1 to j */
7              n=predictNetwork(layer+1, j)
8              c=predictCloud(layer+1, j)
9              /* predict network,cloud for layers j+1 to N-1 */
10             l2=predictLocal(j+1, N-1)
11             totalPredictions=accumulatePredictions(l1,n,c,l2)
12     /* find Pareto Optimal */
13     paretoPoints = DesignSpaceExploration(totalPredictions)
14     return paretoPoints
```

are initially filled with random data. The expected value of an element can be derived by multiplying the i row of the P matrix with the j column of the Q matrix.

$$\begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1N} \\ R_{21} & R_{22} & \cdots & R_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ R_{m1} & R_{m2} & \cdots & R_{MN} \end{bmatrix} \approx \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1K} \\ P_{21} & P_{22} & \cdots & P_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ P_{M1} & P_{M2} & \cdots & P_{MK} \end{bmatrix} \times \begin{bmatrix} Q_{11} & Q_{12} & \cdots & Q_{1N} \\ Q_{21} & Q_{22} & \cdots & Q_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{K1} & Q_{K2} & \cdots & Q_{KN} \end{bmatrix}$$

Figure 5.3.1: Decomposition of R matrix into $P \times Q$

Training of the Collaborative Filter relies on gradient descent. More specifically, training loss calculated by comparing the estimated value and the actual value from the known element multiplied by a preset training rate, updates the features in the P and Q matrices, so that a loss function is minimized, in this case Mean Squared Error - (MSE). This process is repeated for a large number of times so that the values are able to converge. Finally, all the the elements in the R matrix, including the once unknown values can be inferred by multiplying the P and Q matrices.

5.3.1 Offloader

The last component in the *Online Phase* aggrerates the network polynomials and CF data and based on these, performs the actual partitioning and offloading of the DNNs. On each edge node k that wishes to perform an inference, the polynomials as well as the corresponding rows of the LM and EM matrices are gathered in order to generate energy and latency estimates for the different partitionings of a Deep Neural Network. The *DNN Partitioning Exploration* (4b) utilizes the above data by feeding them into the DNN partitioning exploration algorithm that is illustrated in Algorithm 1. This algorithm explores all the possible partitioning solutions and then keeps only the ones that correspond to the Pareto Optimal set defined by the points that have the lowest energy and/or latency. Each pareto point in this set is a different DNN splitting, with a subset of the layers executed on the edge device (4c) and the rest being offloaded to the cloud server (4d). Our possible solutions are characterized by a tuple of two indexes (i, j) that define the following offloading scheme.

- Layers 0 to i are executed on the edge device.
- Layers $i + 1$ to j are executed on the cloud server.
- Layers from $j + 1$ and up to the end of the network are executed once again on the edge device.

Edge cases such as fully local and completely offloaded executions are also evaluated and remain valid execution schemes if their performance/energy estimates are part of the Pareto set. A differentiation

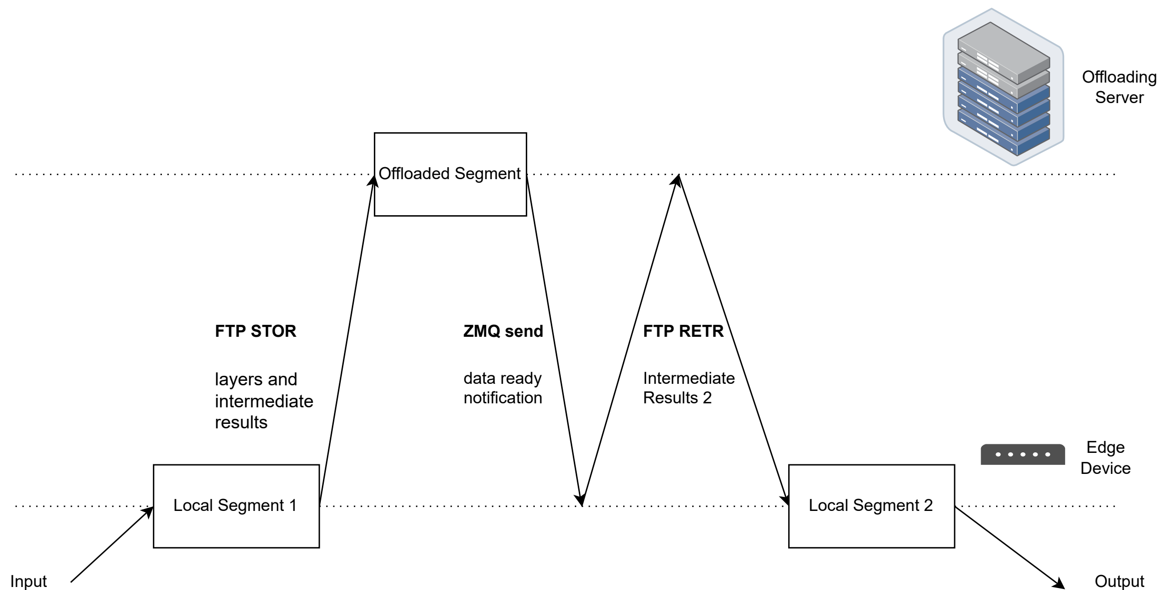


Figure 5.3.2: Offloading Request and Response

point of our framework compare to existing approaches [8] is the ability to offload when skip-layer connections are present, a problem described and solved in subsection 4.2.5.

5.4 Lifetime of an Inference Request

In this section we will discuss the individual components that an allow our framework to facilitate inference requests, elaborating on the steps that make up the calculation of layers, more specifically stages 4c and 4d. The individual components that comprise this pipeline are the following:

- `layerLib`: A module containing the framework's implementation of various Neural Network layers.
- `nnLib`: A library that contains the definitions for various DNN architectures, written to support our framework.
- `offloadingServicesClient`: The module responsible for offloading part of the computation to the edge server.
- `offloadingManager`: The orchestrator for the execution of an inference request, managing both the local and remote parts of the execution.

All of the components mentioned previously reside in the edge device, however there exists one more component. `OffloadingServicesServer` is responsible for the server-side computation of an inference request. In the following segment we will describe the inner workings of the components mentioned above as well as the interactions between them.

5.4.1 layerLib

`layerLib` serves as the interface between our framework and its basis, PyTorch. In essence, it re-implements the definitions of Neural Network Layers by creating wrapper class around the PyTorch layer. This wrapper class maintains full compatibility with PyTorch with one key difference: the `__call__` method is overloaded so that instead of computing the output of a layer it creates a copy of it, registering it with the `offloadingManager` as will be seen later. It also serves the function of managing the inter-layer dependencies found in Residual Neural Networks, a problem that was

mentioned previously in 4.2.5. In this case, we create a new `layerLib` layer type that confines the dependency within it. For example, the implementation of a Bottleneck sequence of layers from ResNet registers a new class of layer, containing all the Conv, BatchNorm, Downsampling and addition operations within the sequence. This new Bottleneck layer is considered atomic, meaning that it can either be offloaded in its entirety, or not offloaded at all.

5.4.2 nnLib

The purpose of `nnLib` is to contain the definitions for the Neural Network Architectures that we have selected. These definitions utilize the layer implementations from `layerLib` so that the Neural Network can later be partitioned and offloaded to the edge server.

5.4.3 offloadingServicesClient

`offloadingServicesClient` provides the core functionality of the offloading mechanism. It achieves this through the use of two different protocols, ZeroMQ and FTP. The offloading mechanism works by first pickling the selected layers along with their input. It then establishes a connection with an FTP server that listens on the offloading device. Before transmitting the data, `offloadingServicesClient` sets up a ZeroMQ socket that listens for messages from the offloading server. The functionality provided by this function will become apparent later. Once the ZMQ socket is initiated, data is transmitted over FTP (`FTP STOR command`). The edge device now blocks, until a ZMQ message is received, notifying the device that the offloaded data has been processed by the server and the results can be received via the `FTP RETR command`. This approach allows us to avoid constantly polling the server for the state of our data. The received results are finally unpickled and all active sockets are closed, with the output provided by the offloading action being fed to the remaining layers of the Neural Network.

5.4.4 offloadingManager

All of the components mentioned previously provide different types of funtionality to our framework. However, it is the job of `offloadingManager` to orchestrate the partitioning and distribution of data. First, of all, the `offloadingManager` component contains a a list with all the layers from the Neural Network. This is achieved through the overloading of the `__call__` method as was mentioned in the presentation of the `layerLib` component. Given this list and the partitioning points i, j from the **DNN Partitioning Exploration**, it creates three different `nn.Sequential` class objects:

1. `Local_1`, containing all the layers up to $i-1$. This object receives the input image and is executed locally on the embedded CPU or GPU depending on the selected configuration. The output of this sequence of layers is the first intermediate result.

2. `Offloaded`, containing layers i through j. This object along with the first intermediate result is offloaded to the edge server with the help of the `offloadingServicesClient` module. Once its results are retrieved, they form the second intermediate result.

3. `Local_2`, containing the rest of the layers, $j+1$ and up to the last layer of the Neural Network. These are computed locally with their output forming the output of the Neural Network.

5.4.5 offloadingServicesServer

In contrast to the previous components, `OffloadingServicesServer` is located on the edge server, launching the FTP server that data is sent to by the `OffloadingServicesClient` component. The implementation is based on `pyftpdlib` by creating a custom `on_receipt` method that is called whenever a file is uploaded to the FTP server. More specifically, instead of behaving like a typical FTP server storing the data over the network, the `on_receipt` method unpickles the received data, computing output of the offloaded `nn.Sequential` class object from the given input. It then sends the ZeroMQ message that notifies the edge device for completion of the computation, as was mentioned in

`OffloadingServicesClient`. Once the data are retrieved, an overload on the `on_file_sent` method deletes the temporary files.

Chapter 6

Experimental Evaluation

6.1 NVIDIA Jetson Family of devices

In this section we will present the NVIDIA Jetson family of edge devices, which are specialized for providing inference solutions at the edge of the network. In accordance with the trend presented in 2, these edge devices incorporate GPUs to accelerate Neural Network inference requests.

We set up all of our edge devices to run the same NVIDIA provided image, namely `Jetson Linux R32.7.1` [33]. Our framework was set up to run on the same PyTorch Docker image `(l4t-pytorch:r32.7.1-pth1.10-py3)` [34], which provided by the manufacturer.

6.1.1 Device Specifications

In order to be fair in our evaluation, we have selected devices with different computational capabilities, ranging from powerful edge devices such as the Jetson AGX and Jetson NX, to less capable ones such as the Jetson Nano and Jetson Tegra TX1 boards.

Jetson AGX

Jetson AGX is the top-of-the line edge device produced by NVIDIA, containing an 8-core ARM processor, 32GB of RAM and a 512-core GPU Volta GPU with Tensor cores. It also contains hardware accelerators for Computer Vision and Deep Learning applications, with the latter supporting fixed point arithmetic. The device can be configured to run in a variety of Power Modes that adjust the Clock frequencies and active cores in its various processing systems, in order to stay under a given power target.

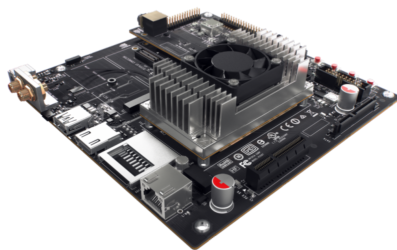
Jetson NX

Jetson NX contains a 6-core CPU and a 384-core Volta-GPU with Tensor cores, as well as 8GB of RAM[1] In general Jetson NX offers increased performance compared to the Jetson TX1 and Jetson Nano device, but reduced performance compared to the AGX. As was the case with the latter, Jetson NX also includes specialized hardware for the acceleration of Deep Learning and Computer Vision applications, as well as the selection of power modes, 5 different modes by default, that enforce power constraints.

[1] There also exists a 16GB model.

(a) NVIDIA Jetson AGX

(b) NVIDIA Jetson NX



(c) NVIDIA Tegra TX1

(d) NVIDIA Jetson Nano

Figure 6.1.1: Devices from the NVIDIA Jetson family

Jetson Nano

Jetson Nano is the smallest, least powerful device currently offered by NVIDIA. It features a Quad core ARM processor, alongside a 128-core Maxwell GPU[2] and 4 GB of RAM [3] This device does not feature the dedicated hardware accelerators mentioned on the previous devices, it does however support two different power modes, setting the maximum power draw to 5 and 10W respectively.

Jetson TX1

The TX1 is a previous generation NVIDIA edge device with a quad core processor, a 256-core Maxwell GPU and 4GB of RAM. Similarly to the Nano, the Jetson TX1 does not feature the hardware accelerators offered on the newer, more powerful devices and it also does not support power mode configuration that the rest of the presented devices offer.

[2] As opposed to the Volta GPU architecture of Jetson NX and AGX this does not support tensor cores.
[3] A 2 GB model also exists.

Table 6.1: Technical characteristics of heterogeneous Edge nodes and Cloud Server [35]

Device	CPU	L2	L3	DRAM	GPU
Jetson AGX	8×Carmel@2.2GHz ARMv8.2 64-bit	8MB	4MB	32GB	512×Volta@1.4GHz
Jetson NX	6×Carmel@1.4GHz ARMv8.2 64-bit	6MB	4MB	8GB	384×Volta@1.1GHz
Jetson Nano	4×Cortex-A57@1.4GHz ARMv8 64-bit	2MB	-	4GB	128×Maxwell@0.9GHz
Jetson TX1	4×Cortex-A57@1.7GHz ARMv8 64-bit	2.5MB	-	4GB	256×Maxwell@1.0GHz

Table 6.2: Configurations by Power Mode table for all devices, highlighted entries are the ones utilized during the experiment [36]

Device	Mode ID	CPU configuration	GPU Freq	Max Power
Jetson AGX	0	8 @ 2.265GHz	1377 MHz	unlimited
Jetson AGX	1	2 @ 1.2GHz	520 MHz	10W
Jetson AGX	2	4 @ 1.2GHz	670 MHz	15W
Jetson AGX	3	8 @ 1.2GHz	900 MHz	30W
Jetson AGX	4	6 @ 1.45GHz	900 MHz	30W
Jetson AGX	5	4 @ 1.78GHz	900 MHz	30W
Jetson AGX	6	2 @ 2.1GHz	900 MHz	30W
Jetson AGX	7	4 @ 2.19GHz	670 MHz	15W
Jetson NX	0	2 @ 1.9GHz	1100 MHz	15W
Jetson NX	1	4 @ 1.4GHz	1100 MHz	15W
Jetson NX	2	6 @ 1.4GHz	1100 MHz	15W
Jetson NX	3	2 @ 1.5GHz	800 MHz	10W
Jetson NX	4	4 @ 1.2GHz	800 MHz	10W
Jetson NX	5	4 @ 1.9GHz	510 MHz	10W
Jetson NX	6	2 @ 1.9GHz	1100 MHz	20W
Jetson NX	7	4 @ 1.4GHz	1100 MHz	20W
Jetson NX	8	6 @ 1.4GHz	1100 MHz	20W
Jetson Nano	0	4 @ 1.48GHz	912 MHz	10W
Jetson Nano	1	2 @ 0.9Hz	640 MHz	5W
Jetson TX1	0	4 @ 1.73GHz	964 MHz	unlimited

6.2 Experimental Setup

6.2.1 Hardware Infrastructure

In order to evaluate our framework we deploy a setup comprised of heterogeneous NVIDIA CPU/GPU devices, as described in 6.1. The role of the offloading/cloud server is covered by a capable x86 server featuring $2 \times 20 - core$ Intel Xeon Gold 5281R processors and an NVIDIA Volta-V100 GPU, forming a setup typical in both edge and cloud scenarios [17].

6.2.2 Technical Implementation

Our framework is mainly written in the Python programming language. The device cluster is connected internally and to the outside world via an 80MB/s wireless network. Control messages are sent via the ZeroMQ protocol and FTP is the connection protocol responsible or the transmission of the actual data and layers. As to overcome the differences in architectures, operating systems and Python package versions, the framework is deployed in a dockerized container. The GPU acceleration for the NN layers is developed in CUDA. Our Collaborating Filtering component also makes use of `C++` and `OpenMP` for acceleration of the critical path.

6.2.3 Examined DNN models

During evaluation we test for a variery of established DNN architectures and their variants, i.e. AlexNet (alex), MobileNetV2 (mv2), Resnet18 (res18), Resnet34 (res34), Resnet50 (res50), Resnet101 (res101), Resnet152 (res152), VGG11, VGG13 and VGG16. These architectures see widespread use in object detection and image classification tasks in edge computing scenarios [37], [38]. We also vary the input sizes, evaluating for images with dimensions of 224×224, 512×512 and 768×768. The DNN models are ported from PyTorch [19] and are integrated to our framework.

6.2.4 Reference Baselines

The evaluation of our approach covers three key metrics.

1. Performance (Latency)
2. Energy Consumption
3. Prediction accuracy of our Collaborating Filtering approach.

We compare ourselves against different execution strategies such as:

- *Offload None*, which executes the Neural Network on the edge device only.
- *Offload All*, performing a full offloading of the DNN to the cloud server.
- *Neurosurgeon*(NS) [8], a state-of-the-art resource management algorithm for DNN offloading.

In addition, since Neurosurgeon pre-offloads the layers to the cloud and thus does not need to transmit any layers (only intermediate results), we level the playing field by implementing a version of it that offloads layers at the time of execution, named *NS-nonOffloaded*. We also implement a pre-offloaded implementation of our framework that works similarly to Neurosurgeon, with the layers offloaded a priori to the cloud.

6.3 Evaluation

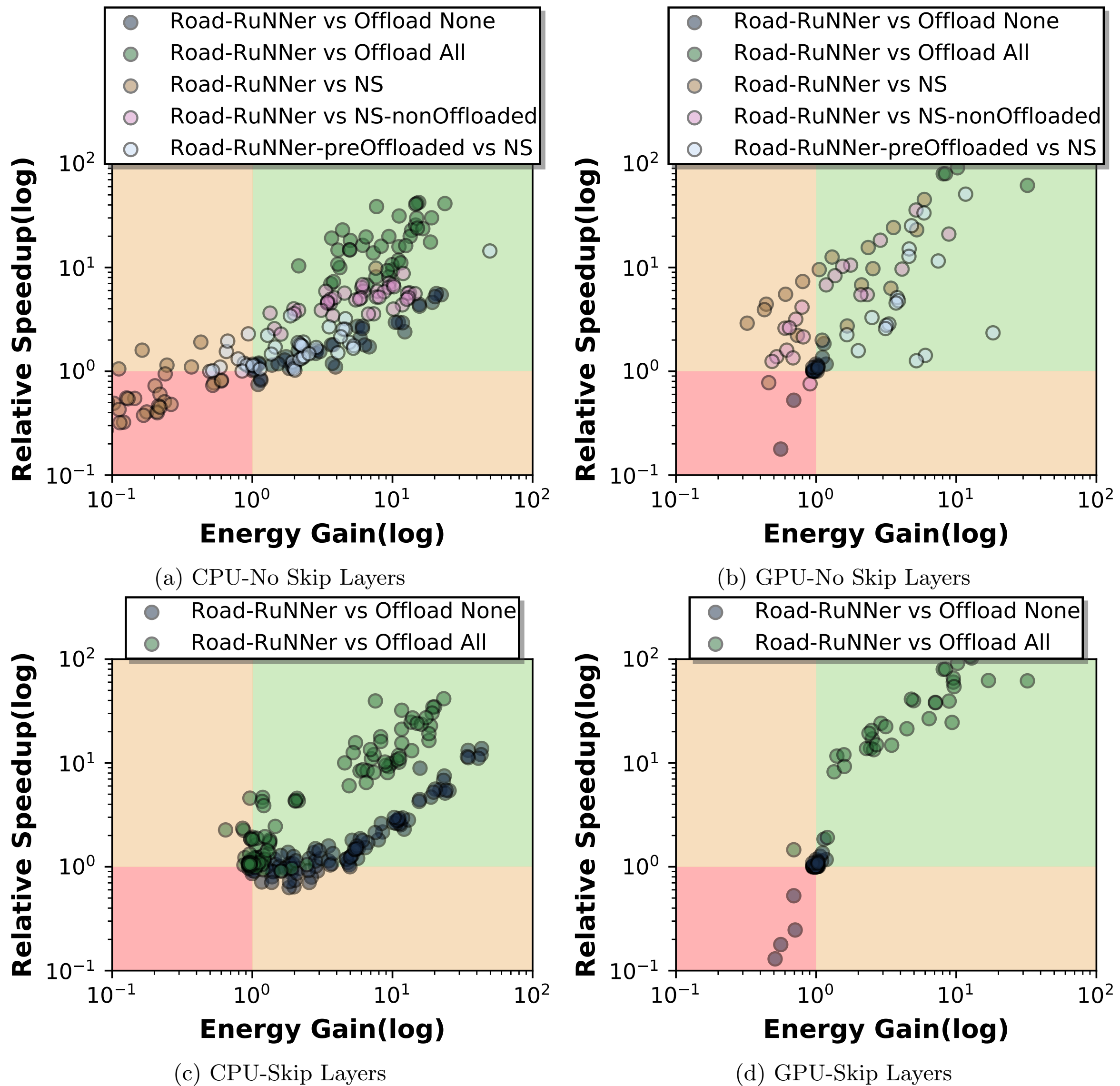
6.3.1 Performance and Energy Evaluation

For our first comparison, we evaluate our framework for performance and energy consumption against the approaches mentioned earlier in Section 6.2, illustrated in Fig. 6.3.1. We limit this comparison to DNNs without skip-layer connections (taking ResNet and MobileNet out of the evaluation pool for now), as Neurosurgeon is not designed to operate with Residual DNN architectures. Thus the displayed data come from experiments on the VGG11,13,16 and AlexNet models only. In all of the plots of Fig. 6.3.1, the X axis demonstrates energy gain (or loss) of our framework against the compared approach and the Y axis similarly demonstrate the relative speedup (or slowdown). Fig. 6.3.1a displays results for CPU execution only, while in Fig. 6.3.1b our devices were configured to perform inference on the GPU. As can be seen, each plot can be divided into four distinct sectors:

- One green sector, where our framework wins the comparison in both speedup and energy
- Two orange ones, demonstrating that our framework performs faster but with higher energy consumption, or slower at reduced energy expenditure against the compared approach.
- One red sector, where our framework performs worse both in terms of performance and energy.

For Fig. 6.3.1a (CPU execution) our framework outperforms the *Offload All* approach by providing up to a 45.41× speedup and 95.84% energy reduction. Similarly, it outperforms the *Offload None* approach with up to 6.61× performance increase and 95.87% energy savings. Similar results can be seen for GPU execution (shown in Fig. 6.3.1b). When comparing ourselves to *Neurosurgeon*, more often than not *NS* wins the comparison, placing our framework in the red quadrant, due to the fact that it already has the DNN weights on the cloud, an insurmountable advantage, as we endure much larger

penalties for transmiting data over the network. Nevertheless, when compared to the online version of *Neurosurgeon* (*NS-nonOffloaded*) our framework can provide a 4.97× mean speedup and a 81.85% mean energy saving on the CPU, with the GPU evaluation giving up to 35.74× better performance and 88.73% energy gain against *NS-nonOffloaded*. This is due to the fact that all *NS* approaches define a single breakpoint, after which all execution is offloaded. That means that if an offloading point is selected, *NS-nonOffloaded* must transmit all the layers beyond that point, which often entails transmitting the large Linear Layers at the end of Neural Networks (seen many times in Section 4.2). As a result, it often has to make a difficult choice between offloading with high network costs and executing locally on relatively slow hardware. Our approach solves this issue by selecting two partition points, allowing offloading to start and stop at any point in the network, thus providing us with the ability to offload large parts of the computation while keeping cumbersome layers locally. Furthermore, if we a priori offload the layers in our framework, we can expect up to 54.09× improvement in terms of performance and up to 58.06× improvement in terms of energy when compared to *Neurosurgeon*.



(a) CPU-No Skip Layers

(b) GPU-No Skip Layers

(c) CPU-Skip Layers

(d) GPU-Skip Layers

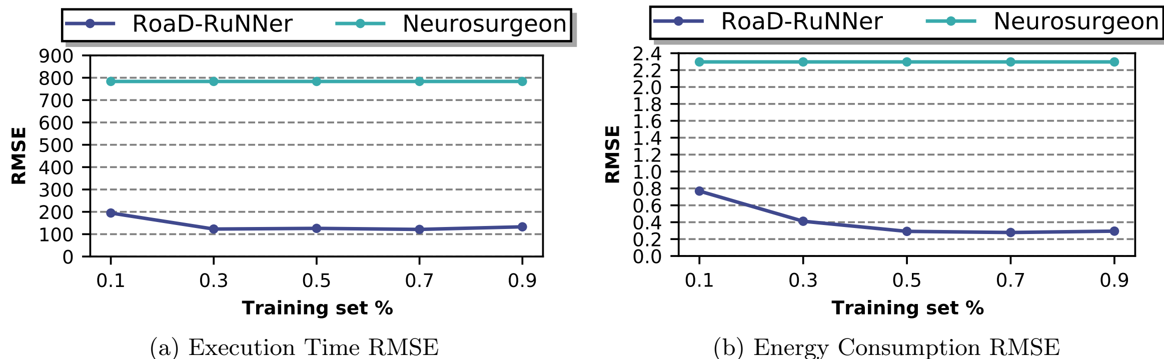
Figure 6.3.1: Performance and Energy Comparison of RoaD-RuNNer framework against other approaches for CPU and GPU nodes for alternative DNN workloads.

Our framework, in contrast to *Neurosurgeon* can handle skip-layer connections in DNNs. As a result we continue our evaluation by adding the previously excluded Resnet18, Resnet34, Resnet50, Resnet101, Resnet152 and MobileNetV2 models to the benchmark suite. The new results are depicted in plots Fig. 6.3.1c and Fig. 6.3.1d for CPU and GPU execution, respectively. When it comes to CPU execution, our framework can outperform the *Offload None* approach by offering up to 13.92× optimized performance and 46.07× less energy consumption. *Offload All* is also outperformed, up to 45.41× in terms of performance and 24.05× in terms of energy. Finally, evaluating our findings for GPU executions reveals an 1.85× speedup (maximum) and 1.34× energy saving (maximum) compared to *Offload None* and 112.98× faster execution on average, with 16.59× smaller energy consumption (also on average) in comparison to *Offload All*.

6.3.2 Prediction Accuracy

The ability of our framework to perform is correlated to the accuracy of its prediction (Collaborative Filtering) mechanism. The first step in its evaluation is a comparison of the Root Mean Square Error (RMSE) of the learning phase in our approach, to the RMSE of Neurosurgeon's prediction algorithm (Linear and Log regression). As seen in Fig. 6.3.2, after only 30% of the input of the training set being given, the latency and energy consumption error has converged. We can also see much better performance when compared to *Neurosurgeon*, up to 6.45× and 8.48× less error, for execution latency and energy, respectively. It must be mentioned that *Neurosurgeon* displays linear behavior, due to the fact that it utilizes the entirety of the dataset during the training phase.

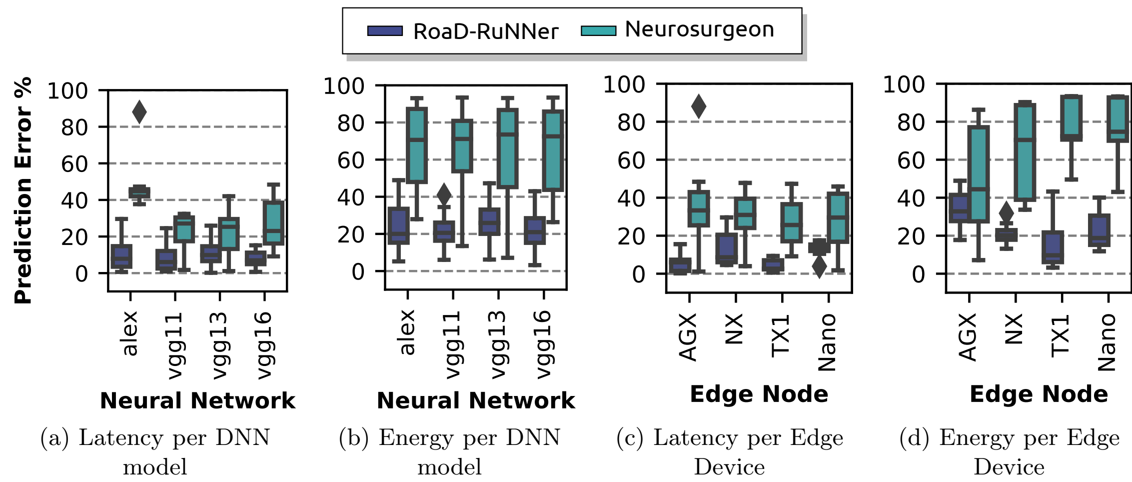
Fig. 6.3.3 compares the accuracy of our execution time and energy predictions for our approach and Neurosurgeon. This is repeated for every examined DNN architecture and available edge device. We outperform Neurosurgeon by having 68.01% less execution time prediction error and 63.8% less energy prediction error per DNN on average (Fig. 6.3.3a, 6.3.3b), while also achieving up to 69.6% less execution time prediction error and up to 34.9% less energy prediction error per edge device (Fig. 6.3.3c, 6.3.3d). The Linear and Log regression prediction models of *Neurosurgeon* are unable to capture complex non-linear behaviours in the system, in contrast to our CF algorithm, thus giving us the advantage in terms of RMSE.



(a) Execution Time RMSE

(b) Energy Consumption RMSE

Figure 6.3.2: Root Mean Square Error(RMSE) of execution time and energy consumption over alternative Collaborative Filtering fill percentages.

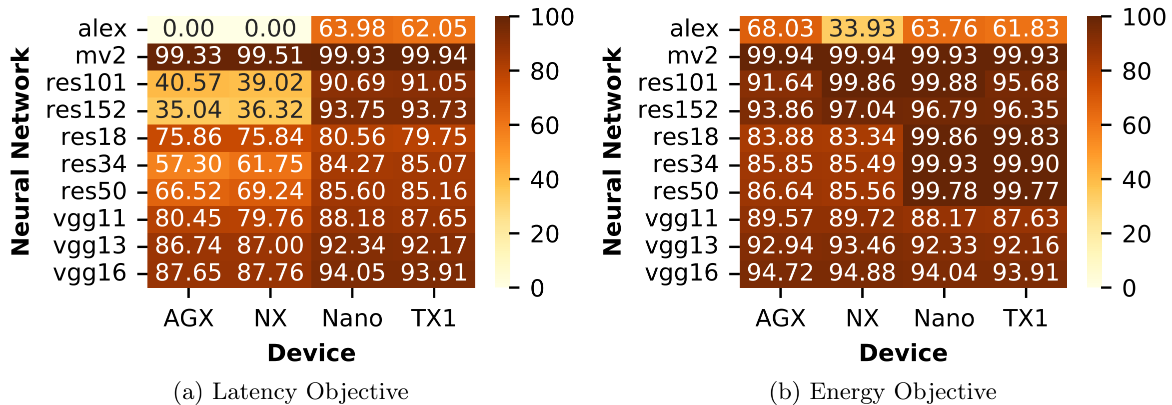


(a) Latency per DNN model (b) Energy per DNN model (c) Latency per Edge Device (d) Energy per Edge Device

Figure 6.3.3: Execution Latency and Energy Consumption Prediction Accuracy for alternative DNN workloads and Edge nodes.

6.3.3 DNN Offload Analysis

More can be said about the decisions made by our framework in order to achieve its execution latency and energy optimization targets. Fig. 6.3.4 depicts the layer offloading percentage (the number of offloaded layers per model and device), for each optimization target. In Fig. 6.3.4a, where the minimization of latency is the goal, we see that the more capable devices such as the Jetson AGX and Jetson NX offload less layers for remote execution, compared to the less powerful devices (Jetson Nano and TX1). Similar behaviour can be seen when trying to optimize for energy consumption (Fig. 6.3.4b). In general, optimizing for energy leads to higher offloading percentages (90.1% on average) than optimizing for latency (74.45% on average). We attribute this fact to the high latency penalties imposed by the network, making large transmission of data and layers prohibitively expensive when trying to reduce the latency.



(a) Latency Objective (b) Energy Objective

Figure 6.3.4: DNN percentage offloading over heterogeneous Edge nodes for latency and energy optimization objectives.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis presents a novel resource management framework for partitioning and offloading of Deep Neural Networks over heterogeneous CPU/GPU edge architectures. Our framework utilizes Collaborative Filtering techniques to estimate performance and energy requirements of individual DNN layers over heterogeneous devices. By aggregating this information and passing it on to a partitioning algorithm, it specifies a set of Pareto optimal DNN partitioning schemes that trade-off between inference latency and energy consumption. Our approach outperforms existing state-of-the-art approaches by providing 9.58× speedup on average and up to 88.73% less energy consumption, while offering high prediction accuracy, limiting the prediction error down to 3.19% and 0.18% for latency and energy, respectively.

7.2 Future Work

Various optimizations and variations on our work can be applied in the future. During the formulation of our methodology and subsequent experimentation we discussed the following possibilities as future tangents to our work. Through these we aim to further increase the performance of our framework and expand it to more heterogeneous platforms.

- Optimizing and pipelining the communication stage of our algorithm so that communication and computation overlap, at least partially.
- Dynamically switching between the available power modes in the edge devices in order to meet inference deadlines or minimize energy.
- Implementing the more detailed graph solving approach for managing dependencies in Residual Neural Networks.
- Caching of offloaded layers at the edge server so that previously executed layers will not need to be re-transmitted.
- Offloading to neighbouring edge devices (as opposed to the current edge server only approach) to increase throughput.
- Creating more dynamic partitioning and offloading algorithms that modify the offloading scheme at inference time in order to meet deadlines.
- Expanding our framework to other architectures such as FPGAs to further exploit heterogeneity.

Bibliography

[1] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[2] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: Review, opportunities and challenges," *Briefings in bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 2018.

[3] Y. Liang, D. O'Keeffe, and N. Sastry, "Paige: Towards a hybrid-edge design for privacy-preserving intelligent personal assistants," in *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, 2020, pp. 55–60.

[4] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[5] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[6] 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.

[7] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE communications surveys & tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[8] Y. Kang, J. Hauswald, C. Gao, *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[9] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 401–411.

[10] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.

[11] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: Synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th annual international conference on mobile computing and networking*, 2020, pp. 1–15.

[12] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Toward collaborative inferencing of deep neural networks on internet-of-things devices," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, 2020.

[13] M. Katsaragakis, D. Masouros, V. Tsoutsouras, *et al.*, "Dmrm: Distributed market-based resource management of edge computing systems," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, pp. 1391–1396.

[14] A. Karteris, M. Katsaragakis, D. Masouros, and D. Soudris, "Sgrm: Stackelberg game-based resource management for edge computing systems," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 1203–1208.

[15] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[16] E.-I. Christoforidis, S. Xydis, and D. Soudris, “Cf-tune: Collaborative filtering auto-tuning for energy efficient many-core processors,” *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 25–28, 2017.
[17] *Edge computing: Gaining the digital edge*, Accessed: 10-09-2022.
[18] L. Dagum and R. Menon, “Openmp: An industry-standard api for shared-memory programming,” *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998, ISSN: 1070-9924. DOI: 10.1109/99.660313. [Online]. Available:
[19] A. Paszke, S. Gross, F. Massa, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
[20] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, “Edge intelligence: Paving the last mile of artificial intelligence with edge computing,” *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.
[21] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, “Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–37, 2020.
[22] L. Zhou, H. Wen, R. Teodorescu, and D. H. Du, “Distributing deep neural networks with containerized partitions at the edge,” in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.
[23] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available:
[24] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
[25] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, 2012.
[27] A. G. Howard, M. Zhu, B. Chen, *et al.*, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.
[28] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available:
[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.
[30] X. Su and T. M. Khoshgoftaar, “A survey of collaborative filtering techniques,” *Adv. in Artif. Intell.*, vol. 2009, Jan. 2009, ISSN: 1687-7470. DOI: 10.1155/2009/421425. [Online]. Available:
[31] *File Transfer Protocol*, RFC 959, Oct. 1985. DOI: 10.17487/RFC0959. [Online]. Available:
[32] *Tegrastats: Memory and processor usage for tegra-based devices*, Accessed: 17-10-2022.
[33] *Jetson linux*, Accessed: 17-10-2022.
[34] *Nvidia docker images: Pytorch*, Accessed: 17-10-2022.
[35] *Nvidia jetson edge devices*, Accessed: 17-10-2022.
[36] *Nvidia jetson power modes*, Accessed: 17-10-2022.
[37] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, “A survey of convolutional neural networks: Analysis, applications, and prospects,” *IEEE transactions on neural networks and learning systems*, 2021.
[38] E. Arnold, O. Y. Al-Jarrah, M. Dianati, S. Fallah, D. Oxtoby, and A. Mouzakitis, “A survey on 3d object detection methods for autonomous driving applications,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 10, pp. 3782–3795, 2019.