Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

# Απεικόνιση αλγορίθμων Βαθιάς Μάθησης σε πλατφόρμες υλικού Graphcore IPU και NVIDIA GPU

Διπλωματικη Εργασια

**Απόστολος Γερακάρης**

**Επιβλέπων:** Διονύσιος Πνευματικάτος
Καθηγητής

Αθήνα, Απρίλιος 2023

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

# Απεικόνιση αλγορίθμων Βαθιάς Μάθησης σε πλατφόρμες υλικού Graphcore IPU και NVIDIA GPU

Διπλωματική Εργασία

**Απόστολος Γερακάρης**

**Επιβλέπων:** Διονύσιος Πνευματικάτος
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10 Απριλίου 2023.

| *(Υπογραφή)* | *(Υπογραφή)* | *(Υπογραφή)* |
|---|---|---|
| ............................ | ...................... | ............................ |
| Διονύσιος Πνευματικάτος<br>Καθηγητής | Νεκτάριος Κοζύρης<br>Καθηγητής | Γεώργιος Γκούμας<br>Αναπληρωτής Καθηγητής |

Αθήνα, Απρίλιος 2023

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Copyright © – All rights reserved. Με την επιφύλαξη παντός δικαιώματος.
Απόστολος Γερακάρης, 2023.

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.

Το περιεχόμενο αυτής της εργασίας δεν απηχεί απαραίτητα τις απόψεις του Τμήματος, του Επιβλέποντα, ή της επιτροπής που την ενέκρινε.

**ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ**

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

*(Υπογραφή)*

.............................
Απόστολος Γερακάρης

10 Απριλίου 2023

# Abstract

This thesis investigates the performance of hardware accelerators, namely GPUs and IPUs, for Machine Learning and Deep Learning applications. Specifically, we focus on two tasks: automating and accelerating eyeblink-response detection from video and training an image-based CNN face detection model. For the former, we explore and compare different algorithms and optimization techniques to achieve real-time processing speed. In the latter, we optimize the training pipeline by leveraging both CPUs and device accelerators. The Eyeblink Conditioning experiment is a widely used experiment in the field of neuroscience to study learning and memory processes in the brain. In the past, researchers have used potentiometers or electromyography (EMG) to monitor the movement of the eyelid during an experiment. In recent years, the use of computer vision and image processing has greatly reduced the need for these methods, as they they need human intervention and do not allow real-time processing. In order to fully automate eyelid tracking, we chose a combination of face and landmark-detection algorithms and accelerated them to create a fast and accurate implementation. Various different algorithms from the fields of Deep Learning and Machine Learning are analyzed and compared for face detection and landmark detection (eyelid detection). Based on this study, two algorithms are identified as most suitable for our use case: the Ensemble of Regression Trees (ERT) approach for landmark detection and the BlazeFace CNN-based model for face detection. The BlazeFace model was accelerated on three different hardware accelerators: V100 Tesla GPU, MK1 IPU and MK2 IPU. The ERT algorithm was accelerated using multi-core CPUs. A combined implementation is successfully deployed for a real neuroscientific use-case: eyeblink response detection, achieving an overall runtime of 0.642 ms per frame with Tesla V100 GPU and 32 CPU processes, 0.7116 ms per frame with MK2-IPU and 64 CPU processes and 0.761 ms per frame with MK1-IPU and 32 CPU processes. Furthermore, an experimental open-source training implementation of the BlazeFace face detector was built from scratch to benchmark the performance of IPU and GPU hardware accelerators. Our results show that IPU-based systems have superior performance compared to the GPU-based systems in training the CNN-based face detector, especially for small batch sizes.

## Keywords

Face Detection, BlazeFace, Landmark Detection, ERT landmark detector, DLib, Mediapipe, BioID, FDDB, Tesnorflow, Keras, IPU, GPU, Eyeblink Conditioning, Artificial Intelligence, Deep Learning, Machine Learning, Convolutional Neural Networks

# Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω θερμά τον καθηγητή κ. Διονύσιο Πνευματικάτο για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να την εκπονήσω στο εργαστήριο CSLab. Επίσης, ευχαριστώ ιδιαίτερα τον Δρ. Χρήστο Στρύδη, επικεφαλή του Εργαστηρίου Νευροϋπολογιστών του Erasmus MC, για την καθοδήγησή του και την εξαιρετική συνεργασία που είχαμε. Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου και τους φίλους μου για την καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν σε αυτό το ταξίδι.

Αθήνα, Απρίλιος 2023

*Απόστολος Γερακάρης*

# Περιεχόμενα

**Abstract** **1**

**Ευχαριστίες** **3**

**I Ελληνικό Κείμενο** **9**

**1 Εισαγωγή** **11**

**2 Θεωρητικό Υπόβαθρο** **15**
2.1 Eye-blink Conditioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Όραση υπολογιστών και ανίχνευση αντικειμένων . . . . . . . . . . . . . . . . 15
2.2.1 Έννοιες και ορισμοί της Ανίχνευσης Αντικειμένων . . . . . . . . . . . . 16
2.3 Συνελικτικά Νευρωνικά Δίκτυα . . . . . . . . . . . . . . . . . . . . . . . . 18

**3 Πλατφόρμες υλικού** **19**
3.1 Κεντρική Μονάδα Επεξεργαστή (CPU) και Μονάδα Επεξεργαστές Γραφικών (GPU) 19
3.2 Graphcore Intelligence Processing Unit (IPU) . . . . . . . . . . . . . . . . . 20
3.2.1 Προγραμματιστικό μοντέλο IPU . . . . . . . . . . . . . . . . . . . . . 21
3.2.2 Μοντέλο εκτέλεσης εργασιών μιας μονάδας IPU . . . . . . . . . . . . . 22

**4 Επιλογή αλγορίθμων** **25**
4.1 Προδιαγραφές δεδομένων . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Ανίχνευση Προσώπων . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2.1 Επιλεγμένοι αλγόριθμοι . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.3 Αξιολόγηση Μεθόδων Ανίχνευνσης Προσώπου . . . . . . . . . . . . . . . . . . 28
4.3.1 Σύνολα Δεδομένων . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.3.2 Επίδοση ανιχνευτών στο υποσύνολο δεδομένων AWFL . . . . . . . . . . 29
4.3.3 Επίδοση ανιχνευτών στο σύνολο δεδομένων BioID . . . . . . . . . . . . 30
4.3.4 Συμπεράσματα στην αξιολόγηση ανιχνευτών προσώπου . . . . . . . . . . 31
4.4 Eyelid-closure detection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.4.1 Ανίχνευση οροσήμων προσώπου - Landmark detection . . . . . . . . . . 33
4.4.2 Επιλογή ανιχνευτή οροσήμων . . . . . . . . . . . . . . . . . . . . . . . 33

**5 Σχεδιασμός και Υλοποίηση** **35**
5.1 Φόρτωση εικόνων και προ-επεξεργασία . . . . . . . . . . . . . . . . . . . . . 35
5.2 Ανίχνευση Προσώπων- BlazeFace . . . . . . . . . . . . . . . . . . . . . . . 36

5.2.1 Inference σε πατφόρμες υλικού MK1 και MK2 IPU . . . . . . . . . . . . 36
5.2.2 Inference στη Tesla V100 GPU . . . . . . . . . . . . . . . . . . . . . . . . 43
5.3 Ανίχνευση Προσώπων Post-processing . . . . . . . . . . . . . . . . . . . . . 44
5.4 Ανίχνευση Οροσήμων . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.4.1 Αξιολόγηση ανιχνευτών στα συνολα δεδομένων IBUG-300W και BioID datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.4.2 Παράλληλη εκτέλεση διεργασιών σε πολυπύρηνες ῝ΠΥ . . . . . . . . . 47
5.5 Post-processing & Eye-blink Detection . . . . . . . . . . . . . . . . . . . . 47

**6 Αξιολόγηση 49**
6.1 Περιβάλλον Διεξαγωγής Πειραμάτων . . . . . . . . . . . . . . . . . . . . . 49
6.2 Eye-Blink Response Detection . . . . . . . . . . . . . . . . . . . . . . . . . 50
6.2.1 Φόρτωση εικόνων και προ-επεξεργασία . . . . . . . . . . . . . . . . . . 50
6.2.2 Ανίχνευση Προσώπου σε πλατφόρμες υλικού IPU & GPU . . . . . . . . 50
6.2.3 Ανίχνευση οροσήμων σε πολυπύρηνες CPU . . . . . . . . . . . . . . . . 51
6.2.4 Αποτελέσματα ολοκληρωμένης υλοποίησης για Eye-Blink Response Detection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
6.2.5 Επίδοση υλικού για Low-Latency αποκρίσεις . . . . . . . . . . . . . . 53
6.3 Hardware Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.3.1 Εκτέλεση του μοντέλου BlazeFace σε πολλαπλές μονάδες IPU . . . . . 55
6.3.2 Ελάχιστοι απαιτούμενοι πόροι υλικού που πληρούν τις προδιαγραφές . 58

**7 Analyzing Performance of IPU and GPU Platforms for CNN-Based Model Training 59**
7.1 Υλοποίηση . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
7.1.1 Δεδομένα . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
7.1.2 Λογισμικό . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
7.1.3 Αξιολόγηση αποτελεσμάτων . . . . . . . . . . . . . . . . . . . . . . . . 60

**II English Text 65**

**1 Introduction 67**
1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
1.2 Thesis Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
1.3 Contribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
1.4 Thesis organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

**2 Background 73**
2.1 Eye-blink Conditioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
2.2 Object Recognition and Detection . . . . . . . . . . . . . . . . . . . . . . . 73
2.2.1 Introductory Object Detection Concepts . . . . . . . . . . . . . . . . . 74
2.3 Convolutional Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . 76

**3 Hardware Platform** **81**
3.1 General Purpose Processors and GPUs . . . 81
3.1.1 Central Processing Unit (CPU) . . . 81
3.1.2 Graphics Processing Units (GPU) . . . 82
3.1.3 Programming tools . . . 84
3.2 Graphcore Intelligence Processing Unit (IPU) . . . 84
3.2.1 IPU Architecture . . . 85
3.2.2 Programming model . . . 87
3.2.3 Parallel Execution . . . 88
3.2.4 Supported programming tools . . . 89
3.3 Discussion . . . 89

**4 Algorithm selection** **91**
4.1 Requirments for eyeblink-response algorithms . . . 91
4.2 Face-detection . . . 91
4.2.1 Selected Algorithms . . . 94
4.3 Face-detection algorithm comparison . . . 96
4.3.1 Dataset selection . . . 96
4.3.2 Performance on constrained AFLW subset . . . 97
4.3.3 Perfomance on BioID database . . . 100
4.3.4 Conclusion on face-detection testing . . . 101
4.4 Eyelid-closure detection . . . 101
4.4.1 Landmark detection . . . 103
4.4.2 Selected algorithm . . . 104
4.5 Details of the selected algorithms . . . 105
4.5.1 Face detector . . . 105
4.5.2 Landmark detector . . . 107

**5 Implementation** **111**
5.1 Image Loading & Pre-processing . . . 112
5.2 Face Detection - BlazeFace . . . 114
5.2.1 Hardware Specifications . . . 114
5.2.2 Inference on MK1 & MK2 IPU chips . . . 114
5.2.3 Inference on Nvidia Tesla V100 . . . 124
5.2.4 Summary of optimizations for accelerating BlazeFace implementation 126
5.3 Face detection Post-processing . . . 127
5.4 Landmark Detection . . . 127
5.4.1 Training an experimental custom Dlib landmark detector . . . 128
5.4.2 Evaluating on IBUG-300W and BioID datasets . . . 130
5.4.3 Work-sharing with multiple processes . . . 132
5.5 Landmark detection Post-processing & Eye-blink detection . . . 132

**6 Evaluation** **135**
6.1 Experimental set-up . . . 135
6.2 Eye-Blink Response Detection Acceleration results . . . 136
6.2.1 Image Loading and pre-processing . . . 136
6.2.2 Face Detection on IPU & GPU . . . 136
6.2.3 Landmark Detection on multi-core CPU . . . 137
6.2.4 Combined implementation for eyeblink-response detection . . . 138
6.2.5 Low-Latency Hardware Performance . . . 139
6.3 Hardware Scalability . . . 141
6.3.1 Face detection on multiple IPUs . . . 141
6.3.2 Minimum hardware to meet the requirements . . . 143

**7 Analyzing Performance of IPU and GPU Platforms for CNN-Based Model Training** **145**
7.1 Implementation . . . 145
7.1.1 BlazeFace architecture . . . 145
7.1.2 Loss Function and Optimizer . . . 146
7.1.3 Training Datasets . . . 148
7.1.4 Training BlazeFace on IPU and GPU hardware platforms . . . 151
7.2 Training Evaluation of BlazeFace . . . 153
7.2.1 Training Resutls . . . 153

**8 Conclusions** **157**
8.1 Contribution . . . 157
8.2 Discussion . . . 159
8.3 Future work . . . 162

**Παραρτήματα** **165**

**A Appendix** **167**

**Bibliography** **177**

# Μέρος I

# Ελληνικό Κείμενο

## Κεφάλαιο 1

# Εισαγωγή

Η κλασική εξαρτημένη μάθηση (Classical conditioning) είναι μια μορφή συνειρμικής μάθησης, όπου ένας οργανισμός μαθαίνει να συνδέει δύο ερεθίσματα μεταξύ τους. Υπάρχουν δύο βασικά ερεθίσματα: το μη εξαρτημένο ερέθισμα (UCS) και το εξαρτημένο ερέθισμα (CS). Το μη εξαρτημένο ερέθισμα είναι ένα φυσικό ερέθισμα που προκαλεί μια συγκεκριμένη αντίδραση, η οποία ονομάζεται ανεξάρτητη απόκριση. Το εξαρτημένο ερέθισμα, από την άλλη πλευρά, είναι αρχικά ένα ουδέτερο ερέθισμα που δεν προκαλεί την αντίδραση που μας ενδιαφέρει, αλλά μέσω της επαναλαμβανόμενης αντιστοίχισης με το μη εξαρτημένο ερέθισμα, συσχετίζεται με αυτό και τελικά έρχεται να προκαλέσει την ίδια αντίδραση. Η εμφάνιση παρόμοιας αντίδρασης ενός οργανισμού ονομάζεται εξαρτημένη απόκριση. Ένα από τα πιο διάσημα παραδείγματα κλασικής εξαρτημένης μάθησης είναι το πείραμα του Pavlov, στο οποίο συνέδεε επανειλημμένα τον ήχο ενός κουδουνιού με την παρουσίαση τροφής σε σκύλους. Με την πάροδο του χρόνου, τα σκυλιά έμαθαν να συνδέουν τον ήχο του κουδουνιού με την άφιξη της τροφής με αποτέλεσμα να τρέχουν τα σάλια τους μόνο με τον ήχο του κουδουνιού, ακόμη και όταν δεν παρουσιαζόταν τροφή. Αυτό έδειξε πώς ένα μη εξαρτημένο ερέθισμα (το κουδούνι) μπορεί να συνδεθεί με μια αντανακλαστική αντίδραση (σιελόρροια) μέσω επαναλαμβανόμενης αντιστοίχισης με ένα εξαρτημένο ερέθισμα (τροφή) και παρείχε σημαντικές πληροφορίες για τους μηχανισμούς μάθησης και συμπεριφοράς.

Το τμήμα Νευροεπιστήμης του πανεπηστημίου Erasmus MC διεξάγει ένα πείραμα κλασικής εξαρτημένης μάθησης, το οποίο ονομάζεται Eyeblink Conditioning (EBC), και εφαρμόζεται σε ανθρώπους. Σε αυτό το πείραμα, ένας τόνος (το εξαρτημένο ερέθισμα) συνδυάζεται με ένα φύσημα αέρα στο μάτι (το μη εξαρτημένο ερέθισμα), το οποίο προκαλεί ένα κλείσιμο των ματιών (η μη εξαρτημένη απόκριση). Κατά τη διάρκεια του πειράματος, ο ήχος και το φύσημα αέρα συνδυάζονται επανειλημμένα μέχρι ο συμμετέχων να μάθει να συσχετίζει τα δύο ερεθίσματα. Ο ήχος παρουσιάζεται λίγο πριν από το φύσημα αέρα, έτσι ώστε ο συμμετέχων να αρχίσει να αναμένει το φύσημα όταν ακούει τον ήχο. Με την πάροδο του χρόνου, ο τόνος μόνος του έρχεται να προκαλέσει την αντίδραση του κλεισίματος των ματιών (την εξαρτημένη απόκριση). Έτσι, μετά από μια περίοδο επαναλαμβανόμενων συχετίσεων, οι ερευνητές μπορούν να μελετήσουν τα χαρακτηριστικά της εξαρτημένης απόκρισης των ματιών παρουσιάζοντας μόνο τον ήχο.

Το Eyeblink conditioning (EBC) είναι μια καλά μελετημένη μορφή κλασικής εξαρτη-

μένης μάθησης που χρησιμοποιείται από τους επιστήμονες για την εξαγωγή πολύτιμων πληροφοριών σχετικά με τις νευρικές δομές και τους μηχανισμούς που διέπουν τη μάθηση και τη μνήμη. Πρόσφατες μελέτες που κάνουν χρήση της συγκεκριμένης μεθόδου είναι οι [1], όπου μελετάται τι συμβαίνει στους ανθρώπινους εγκεφάλους όταν μαθαίνονται νέες κινητικές δεξιότητες, [2], όπου μελετάται η επίδραση διαφόρων διαταραχών από το φάσμα του αυτισμού στις αποκρίσεις του ματιού (conditioned responses) και [3], όπου η μέθοδος EBC χρησιμοποιείται για τη διερεύνηση της παρεγκεφαλιδικής δυσλειτουργίας σε διαταραχές σχιζοφρένειας.

Μια πρώιμη προσέγγιση για τη μέτρηση της εξαρτημένης απόκρισης των ματιών ήταν η ανάλυση του κλεισίματος των βλεφάρων σε ένα βίντεο, καρέ-καρέ. Οι επιστήμονες έπρεπε να επιλέξουν χειροκίνητα το μισό πρόσωπο στο πρώτο καρέ και στη συνέχεια να επιλέξουν χειροκίνητα το μάτι που τους ενδιέφερε. Στη συνέχεια, ακολουθούσε μια διαδικασία αντιστοίχισης (template matching) [4] για να περικοπεί η περιοχή του ματιού σε κάθε καρέ/εικόνα και τελικά να υπολογιστεί το κλείσιμο των βλεφάρων. Τα μειονεκτήματα αυτής της προσέγγισης ήταν η ανάγκη χειροκίνητης παρέμβασης (επιλογή του προσώπου και του ματιού για κάθε νέα δοκιμή) και η αναγκαιότητα αποθήκευσης πολλών δεδομένων, καθώς το πραγματικό βίντεο πρέπει να προβληθεί και να υποστεί επεξεργασία από έναν άνθρωπο.

Η δημιουργία μιας αυτόματης διαδικασίας ανίχνευσης προσώπου και ματιών, με ταχύτητα επεξεργασίας ικανή να συμβαδίζει με το ρυθμό καρέ του βίντεο, αποτελεί σημαντικό βήμα προς την κατεύθυνση μιας on-line εφαρμογής για την ανάλυση των αποτελεσμάτων. Αυτό θα επέτρεπε στους ερευνητές να αναλύουν τα αποτελέσματα του πειράματος σε πραγματικό χρόνο, γεγονός που με τη σειρά του θα μείωνε την ανάγκη ανθρώπινης παρέμβασης και αποθήκευσης των δεδομένων.

Τα τελευταία χρόνια, η έρευνα και η ανάπτυξη της Μηχανικής Μάθησης (Machine Learning), έχει αυξηθεί ραγδαία και χρησιμοποιείται σε διάφορα επιστημονικά πεδία. Η μηχανική μάθηση αποτελεί μέρος του κλάδου της πληροφορικής που παρέχει τη δυνατότητα στον υπολογιστή μέσα από τη συλλογή δεδομένων ή τις περιπτώσεις που έχει αντιμετωπίσει, να μαθαίνει για το εκάστοτε πρόβλημα χωρίς να χρειάζεται περαιτέρω και πιο ειδικός προγραμματισμός. Η Βαθιά Μάθηση [5] είναι ένα υποπεδίο της Μηχανικής Μάθησης και προσπαθεί να μιμηθεί την ανθρώπινη νοημοσύνη μέσα από το συνδυασμό δεδομένων, βαρών (weights) και της προκατάληψης (bias). Η Βαθιά Μάθηση διαφοροποιείται από τη Μηχανική Μάθηση βάσει του τύπου δεδομένων που επεξεργάζεται όσο και βάσει των μεθόδων που χρησιμοποιεί για να μάθει. Χρησιμοποιεί δίκτυα πολλαπλών επιπέδων ώστε σταδιακά να εξάγει χαρακτηριστικά υψηλότερου επιπέδου, χωρίς την παρέμβαση του ανθρώπινου παράγοντα, από ανεπεξέργαστα μη δομημένα δεδομένα (όπως φωτογραφίες, κείμενα, βίντεο κ.α.).

Τα δίκτυα αυτά ονομάζονται Βαθιά Νευρωνικά Δίκτυα (Deep Neural Networks - DNNs) και βρίσκουν εφαρμογή σε πληθώρα ερευνητικά θέματα όπως η όραση υπολογιστών, η επεξεργασία φυσικής γλώσσας, η αναγνώριση ομιλίας. Μεταξύ αυτών των θεμάτων, παραδείγματα της όρασης υπολογιστών είναι αλγόριθμοι για ταξινόμηση εικόνων (image classification) [6], ανίχνευση αντικειμένων (object detection) [7] [8] και κατάτμηση εικόνων (image segmenta-

tion) [9] οι οποίοι έχουν προσελκύσει αυξανόμενο ερευνητικό ενδιαφέρον λόγω των δυνατοτήτων τους σε ένα ευρύ φάσμα εφαρμογών του πραγματικού κόσμου, όπως η αυτόνομη οδήγηση, η αλληλεπίδραση ανθρώπου-μηχανής και η ανάλυση ιατρικών εικόνων.

Ειδικότερα, τα Συνελικτικά Νευρωνικά Δίκτυα (CNN) έχουν χρησιμοποιηθεί ευρέως και έχουν επιδείξει κορυφαίες επιδόσεις σε εφαρμογές επεξεργασίας εικόνας. Με την εξέλιξη των βαθιών νευρωνικών δικτύων, εισήχθησαν μεγαλύτερα και βαθύτερα μοντέλα για την αντιμετώπιση δυσκολότερων και πιο σύνθετων προβλημάτων, και ως εκ τούτου αυτές οι προσεγγίσεις που βασίζονται σε CNN εχουν αυξημένες απαιτήσεις σε αποθηκευτικό χώρο (storage), μνήμη χρόνου εκτέλεσης (runtime memory), καθώς και υπολογιστική ισχύ τόσο κατά την εκπαίδευση (training) όσο και κατά την εξαγωγή συμπερασμάτων (inference). Η διαδικασία εξαγωγής συμπερασμάτων χρησιμοποιείται για την ταξινόμηση ή την εξαγωγή προβλέψεων από τα δεδομένα εισόδου σε εφαρμογές του πραγματικού κόσμου, μετά την εκπαίδευση του νευρωνικού δικτύου.

Οι προσεγγίσεις βαθιάς μάθησης, και πιο συγκεκριμένα τα CNN, απαιτούν συνήθως τη χρήση ενός επιταχυντή υλικού για την επιτάχυνση των υπολογισμών. Όπως αναφέρθηκε προηγουμένως, η εξαγωγή συμπερασμάτων εκτελείται σε εφαρμογές του πραγματικού κόσμου, αφού εκπαιδευτεί ένα μοντέλο, και ως εκ τούτου είναι σημαντικό να διασφαλιστεί ότι το υλικό, στο οποίο αναπτύσσεται η εφαρμογή, είναι ικανό να εκτελεί αποτελεσματικούς και γρήγορους υπολογισμούς. Επιπλέον, η εκπαίδευση των DNN είναι απαιτητική σε δεδομένα, απαιτητική σε πόρους και χρονοβόρα. Περιλαμβάνει την ολιστική χρήση όλων των πόρων σε έναν διακομιστή, από την αποθήκευση και τη CPU για την άντληση και την προεπεξεργασία του συνόλου δεδομένων έως το εξειδικευμένο υλικό (GPU, IPU) που εκτελεί υπολογισμούς στα μετασχηματισμένα δεδομένα.

Η παρούσα εργασία επικεντρώνεται στην αξιολόγηση των επιδόσεων των πλατφορμών υλικού GPU και IPU με την ανάπτυξη αλγορίθμων Μηχανικής Μάθησης και μοντέλων Βαθιάς Μάθησης τόσο σε εφαρμογές εξαγωγής συμπερασμάτων (Inference) όσο και σε εργασίες εκπαίδευσης (Training). Αρχικά επικεντρώνεται στον σχεδιασμό και ανάπτυξη μιας εφαρμογής για την εξαγωγή συμπερασμάτων στις διαθέσιμες πλατφόρμες υλικού. Η εφαρμογή αυτή έχει στόχο την αυτοματοποίηση και επιτάχυνση της διαδικασίας ανίχνευσης της απόκρισης των ματιών από καρέ βίντεο, προκειμένου να επιτευχθεί ταχύτητα επεξεργασίας σε πραγματικό χρόνο. Εξετάζουμε και χρησιμοποιούμε λύσεις από τα πεδία της μηχανικής και βαθιάς μάθησης για την κατασκευή μιας on-line εφαρμογής, η οποία μας απαλλάσσει από την ανάγκη ύπαρξης μεγάλου αποθηκευτικού χώρου (off-line storage) για τα καταγεγραμμένα βίντεο καθως επίσης επιτρέπει τη προσαρμογή του πειράματος σε συνθήκες πραγματικού χρόνου εκτέλεσης. Χρησιμοποιούμε δύο διαφορετικούς επιταχυντές υλικού (IPU, GPU) για την επιτάχυνση της διαδικασίας ανίχνευσης και συγκρίνουμε τη συνολική τους απόδοση (όσον αφορά την καθυστέρηση απόκρισης - latency, την κατανάλωση ενέργειας και την αποδοτικότητα των υπολογισμών - computational efficiency) καθώς και τη καταλληλότητά τους για εφαρμογές επεξεργασίας εικόνας σε συνθήκες πραγματικού χρόνου.

Στη συνέχεια, η παρούσα εργασία επικεντρώνεται στην αξιολόγηση των επιδόσεων κάθε

πλατφόρμας υλικού και στη διαδικασία εκπαίδευσης ενός Συνελικτικού Νευρωνικού Δικτύου (CNN), με βάση την πρωτότυπη εργασία του ανιχνευτή προσώπων BlazeFace [10]. Η διαδικασία εκπαίδευσης είναι πολύπλοκη και απαιτεί αποτελεσματική αξιοποίηση τόσο των CPU όσο και των επιταχυντών συσκευών, όπως οι GPU και οι IPU. Θα διερευνήσουμε και θα εφαρμόσουμε βελτιστοποιήσεις σε διάφορα βήματα του αγωγού εκπαίδευσης για την επίτευξη βέλτιστης απόδοσης.

Η αρχιτεκτονική των IPU ακολουθούν διαφορετική προσέγγιση σε σύγκριση με τις GPU και κάθε συσκευή έχει τα πλεονεκτήματα και τα μειονεκτήματά της. Στο πλαίσιο αυτό, η αξιολόγηση των επιδόσεων των διαθέσιμων επιταχυντών υλικού είναι ζωτικής σημασίας για την ανάπτυξη αποτελεσματικών και αποδοτικών εφαρμογών μηχανικής μάθησης και βαθιάς μάθησης που μπορούν να αντεπεξέλθουν στις υπολογιστικές απαιτήσεις του μέλλοντος.

Τέλος είναι σημαντικό να αναφερθεί οτι η παρούσα εργασία αποτελεί μέρος μιας ευρύτερης συνεργασίας μεταξύ του CSLab του Εθνικού Μετσόβιου Πολυτεχνείου και του τμήματος Νευροεπιστήμης του Erasmus MC και αποσκοπεί στην αξιοποίηση και αξιολόγηση σύγχρονων επιτεχυντών υλικού χρησιμοποιόντας λύσεις που παρέχονται από τα πεδία της Μηχανικής Μάθησης και της Βαθιάς Μάθησης.

Κεφάλαιο 2

# Θεωρητικό Υπόβαθρο

Στο παρών κεφάλαιο θα καλυφθεί το θεωρητικό υπόβαθρο που κρίνεται απαραίτητο για την κατανόηση της διπλωματικής εργασίας.

## 2.1 Eye-blink Conditioning

Στην παρούσα εργασία πραγματευόμαστε το πείραμα Eyeblink Conditioning σε ανθρώπινους οργανισμούς. Γίνεται χρήση μιας μιας κάμερας υψηλής ταχύτητας για την καταγραφή των δεδομένων και κάθε βίντεο έχει διάρκεια 2 δευτερολέπτων. Ο ρυθμός καταγραφής είναι 333 καρέ ανά δευτερόλεπτο (FPS), και επομένως κάθε βίντεο περιέχει 666 καρέ. Κατα την διάρκεια των πειραμάτων είναι αναγκαίο να καταγραφεί μια ευρεία περιοχή, καθώς ο άνθρωπος κινείται σχετικά ελεύθερα κατά τη διάρκεια της δοκιμής και επομένως δεν εξασφαλίζεται οτι η θέση του κεφαλιού θα παραμείνει σταθερή. Η κάμερα τοποθετείται περίπου ένα μέτρο μακριά απο τον άνθρωπο για την καταγραφή του προσώπου. Η θέση και η στάση σώματος του υποκειμένου έχουν σημαντικό ρόλο στη διαδικασία καταγραφής, καθώς απότομες κινήσεις και περιστροφές του προσώπου έχουν μεγάλο αντίκτυπο στα καταγεγραμμένα δεδομένα τα οποία μπορούν να καταστούν ακατάλληλα. Για τον λόγο αυτό μια οθόνη που προβάλει μια ταινία τοποθετείται δίπλα απο την κάμερα για να βοηθήσει το υποκείμενο να κρατήσει όσο τον δυνατόν πιο σταθερή τη θέση του.

## 2.2 Όραση υπολογιστών και ανίχνευση αντικειμένων

Όραση Υπολογιστών (Computer Vision): Είναι το πεδίο της Τεχνητής Νοημοσύνης που επιτρέπει σε υπολογιστές και συστήματα να αντλούν σημαντικές πληροφορίες από ψηφιακές εικόνες ή βίντεο και να προβαίνουν σε ενέργειες ή να κάνουν προτάσεις με βάση τις εξαγόμενες πληροφορίες. Το Computer Vision έχει αρχίσει και αποκτά ιδιαίτερο ενδιαφέρον καθώς έχει πολλές εφαρμογές σε διάφορα πεδιά της καθημερινής ζωής. Κάποια παραδείγματα είναι η διάγνωση ενός ασθενούς μέσω επεξεργασίας και ανάλυσης ιατρικών εικόνων, η ρομποτική (αυτόνομα οχήματα, κινητά ρομπότ) και η αλληλεπίδραση υπολογιστήανθρώπου.

Η Αναγνώριση Αντικειμένου (Object Detection) είναι ένα υποπεδίο της Όρασης Υπολογιστών (Computer Vision) που μας επιτρέπει να αναγνωρίζουμε και να εντοπίζουμε αν-

τικείμενα μέσα από μια φωτογραφία ή ένα βίντεο. Το ειδικό χαρακτηριστικό σχετικά με την Αναγνώριση Αντικειμένου είναι ότι προσδιορίζει την κλάση (άνθρωπος, σκύλος, γάτα κλπ) του αντικειμένου και τις ακριβείς συντεταγμένες του στη δεδομένη εικόνα ή βίντεο. Το πρόβλημα αυτό περιγράφεται ως εξής: Με δεδομένη μία εικόνα εισόδου, πρέπει να προβλεφθεί η τοποθεσία και η έκταση των αντικειμένων που ανήκουν σε ένα σύνολο προκαθορισμένων κλάσεων, και να αποδοθεί η σωστή κλάση στο κάθε ένα. Η πληροφορία για τη θέση ενός αντικειμένου δίνεται ως συντεταγμένες ενός πλαισίου οριοθέτησης (bounding box)το οποίο σχεδιάζεται γύρω απο το αντικείμενο.

Οι μέθοδοι για την επίλυση του προβλήματος Ανίχνευσης Αντικειμένων μπορούν να χωριστούν σε προσεγγίσεις Μηχανικής Μάθησης βασισμένες και σε προσεγγίσεις Βαθιάς Μάθησης βασισμένες σε Νευρωνικά Δίκτυα (Neural Networks). Για αυτές που βασίζονται στη μηχανική μάθηση αρχικά πρέπει να καθοριστούν προσεκτικά τα χαρακτηριστικά και κατόπιν χρησιμοποιούνται ταξινομητές Support Vector Machines (SVM), για να γίνει η κατηγοριοποίηση [11][12]. Απο την άλλη, οι μέθοδοι που στηρίζονται σε Νευρωνικά Δίκτυα είναι ικανές να κάνουν μία end-to-end Ανίχνευση Αντικειμένων, χωρίς να χρειάζεται ή να απαιτείται ο προσδιορισμός των χαρακτηριστικών και συνήθως χρησιμοποιούν Συνελικτικά Νευρωνικά Δίκτυα (CNN). Οι σύγχρονες προσεγγίσεις που βασίζονται στη Βαθιά Μάθηση [13][14][15] έχουν επιτύχει σημαντική βελτίωση των επιδόσεων σε σύγκριση με τις μεθόδους Μηχανικής Μάθησης και χρησιμοποιούνται σε ένα ευρύ φάσμα εφαρμογών του πραγματικού κόσμου.

### 2.2.1 Έννοιες και ορισμοί της Ανίχνευσης Αντικειμένων

#### Πλαίσιο Οριοθέτησης (Bounding Box)

Το Bounding Box ή αλλιώς Πλαίσιο Οριοθέτησης είναι ένα από τα πιο αναγνωρισμένα και συνήθως χρησιμοποιούμενα εργαλεία στην Ανίχνευση Αντικειμένων. Σε μια εικόνα ένα Πλαίσιο Οριοθέτησης είναι ένα νοητό ορθογώνιο, το οποίο περικλείει ένα ολόκληρο αντικείμενο. Εκτός από τη θέση του αντικειμένου μέσα σε μια εικόνα μπορεί να χρησιμοποιηθεί για τον καθορισμό πρόσθετων χαρακτηριστικών ενός αντικειμένου, όπως η κλάση (π.χ. πρόσωπο, μη-πρόσωπο) και εμπιστοσύνη (πόσο πιθανό είναι το αντικείμενο να βρίσκεται σε αυτή τη θέση). Η αναπαράσταση ενός Πλαίσιου Οριοθέτησης γίνεται συνήθως με έναν από τους εξής τρόπους:

- Δίνοντας τις συντεταγμένες δύο σημείων του ορθογωνίου, δηλαδή $(x_{min}, y_{min})$ και $(x_{max}, y_{max})$. Όπου $(x_{min}, y_{min})$ συντεταγμένες της κάτω αριστερής γωνίας και $(x_{max}, y_{max})$ συντεταγμένες της άνω δεξιάς γωνίας.
- Δίνοντας τις συντεταγμένες του σημείου του κέντρου του Πλαισίου Οριοθέτησης $(x_c, y_c)$ και έπειτα του ύψους και του πλάτους (w, h).

#### Non Maximum Suppression

Η Non Maximum Suppression [16] είναι μια μέθοδος υπολογιστικής όρασης που επιλέγει μια μοναδική οντότητα από πολλές επικαλυπτόμενες οντότητες. Το κριτήριο είναι συνήθως

η απόρριψη οντοτήτων που βρίσκονται κάτω από ένα δεδομένο όριο πιθανότητας. Ένα πρόβλημα που εμφανίζεται συχνά στην Ανίχνευση Αντικειμένων είναι η ύπαρξη πολλαπλών πλαισίων οριοθέτησης που περιγράφουν το ίδιο αντικείμενο. Ο αλγόριθμος NMS θα επιλέξει το καλύτερο Πλαίσιο Οριοθέτησης με τη μέγιστη εμπιστοσύνη και θα αποκλείσει όλα τα άλλα πλαίσια με επικάλυψη μεγαλύτερη από 50% σε σχέση με το επιλεγμένο. Η διαδικασία επαναλαμβάνεται έως ότου τελικά επιλέγεται ένα πλαίσιο.

#### Λόγος Τομής προς Ένωση (Intersection over Union)

Για την υλοποίηση οποιουδήποτε συστήματος ανίχνευσης αντικειμένων είναι απαραίτητος ο ορισμός μίας μετρικής της ομοιότητας μεταξύ δύο αντικειμένων. Η μετρική αυτή χρησιμοποιείται τόσο για τη σύγκριση της πρόβλεψης με το αληθινό αντικείμενο (ground truth) για την αξιολόγησή της, όσο και για τη σύγκριση διαφορετικών προβλέψεων μεταξύ τους με σκοπό την απαλοιφή υπερβολικά όμοιων προβλέψεων. Το πιο ευρέως χρησιμοποιούμενο μέγεθος για το σκοπό αυτό είναι ο λόγος της τιμής προς την ένωση (Intersection over Union - IoU).

Στην περίπτωση της σύγκρισης αντικειμένων που περιγράφονται από πλαίσια οριοθέτησης, το IoU ορίζεται ως το εμβαδόν της τομής των δύο πλαισίων, δηλαδή της περιοχής που ανήκει τόσο στο ένα πλαίσιο όσο και στο άλλο, προς το εμβαδόν της ένωσής τους, δηλαδή της συνολικής περιοχής που καλύπτουν και τα δύο πλαίσια, όπως φαίνεται στο Σχήμα 2.1.

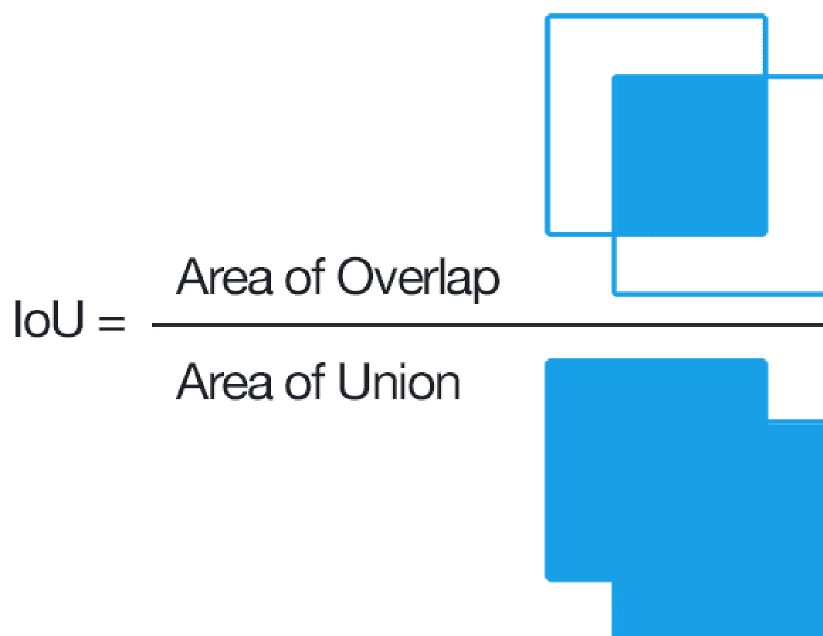



**Figure 2.1.** *IOU definition. Image from:* https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

## 2.3 Συνελικτικά Νευρωνικά Δίκτυα

Τα συνελικτικά νευρωνικά δίκτυα (CNN) είναι ένα είδος νευρωνικών δικτύων στη βαθιά μάθηση, τα οποία χρησιμοποιούνται ευρέως σε προβλήματα οπτικής αναγνώρισης. Σε σύγκριση με άλλες μεθόδους οπτικής αναγνώρισης, όπως οι ταξινομητές SVM, το CNN μπορεί να χειριστεί την εικόνα εισόδου στο επιθυμητό αποτέλεσμα από άκρο σε άκρο χωρίς κανένα χαρακτηριστικό σχεδιασμένο από τον άνθρωπο.

Τα συνελικτικά νευρωνικά δίκτυα (CNN) ανήκουν στην κλάση των προς τα εμπρός τροφοδοτούμενων δικτύων. Συγκεκριμένα αποτελούνται συνήθως από ένα στρώμα εισόδου (Input Layer), πολλαπλά κρυφά στρώματα (Hidden Layers) και ένα στρώμα εξόδου (Output Layer). Τα κρυφά στρώματα αποτελούνται από συνελικτικά (cnn), συγκεντρωτικά (pooling layer) και πλήρως συνδεδεμένα στρώματα (fully connected layer). Τα περισσότερα συστήματα αναγνώρισης εικόνων χρησιμοποιούν CNN καθώς αυτά βασίζονται στο γεγονός ότι τα στατιστικά των εικόνων είναι μεταφραστικά αμετάβλητα και συνεπώς μπορεί να γίνει εκμάθηση και αναπαράσταση πολλών χαρακτηριστικών μέσω φίλτρων που υπάρχουν στα συνελικτικά στρώματα. Τα φίλτρα αυτά εφαρμόζονται στα εικονοστοιχεία της εικόνας με σκοπό να εξαχθούν τοπικά χαρακτηριστικά, όπου με την επανάληψη της διαδικασίας αυτής αποκτώνται χαρακτηριστικά υψηλότερου επιπέδου. Ένα φίλτρο είναι στην ουσία ένας πίνακας $n \times n$ που περιέχει ορισμένα βάρη w, τα οποία βάρη πολλαπλασιάζονται με τα pixels της εικόνας στοιχείο προς στοιχείο (element wise operation). Εφαρμόζοντας λοιπόν το φίλτρο σε μία εικόνα προκύπτει μία νέα αναπαράσταση αυτής, η οποία ενδέχεται να υποστεί περεταίρω επεξεργασία. Το μέγεθος του φίλτρου είναι συνήθως $3 \times 3, 5 \times 5$ ή $7 \times 7$ αλλά γενικά μπορεί να πάρει οποιαδήποτε τιμή έτσι ώστε να μπορεί να εξάγει περισσότερα ή λιγότερα χαρακτηριστικά κάθε φορά.

Οι λεπτομέρειες της αρχιτεκτονικής ενός CNN δεν ακολουθούν κάποιον συγκεκριμένο κανόνα ο οποίος θα προσφέρει σίγουρα αποτελέσματα για ένα πρόβλημα και συνεπώς η επιλογή αυτών γίνεται εμπειρικά/πειραματικά. Κάποιες από αυτές τις λεπτομέρειες είναι το είδος των στρωμάτων και το πλήθος του κάθε είδους, η σειρά των στρωμάτων, οι διαστάσεις των φίλτρων οι συναρτήσεις απώλειας κ.α. Η επιλογή της δομής ενός μοντέλου επηρεάζεται σημαντικά από τα σύνολα δεδομένων στα οποία εκπαιδεύονται και ελέγχονται. Μια μέθοδος που ακολουθείται συχνά σχετικά με το μέγεθος του δικτύου είναι αρχικά να δοκιμαστεί με σχετικά μικρό πλήθος στρωμάτων και σταδιακά να αυξάνεται όσο παρατηρείται βελτίωση απόδοσης.

## Κεφάλαιο 3

# Πλατφόρμες υλικού

### 3.1 Κεντρική Μονάδα Επεξεργαστή (CPU) και Μονάδα Επεξεργαστές Γραφικών (GPU)

Οι σύγχρονες CPU έχουν κατασκευαστεί ως επεξεργαστές γενικής χρήσης, στους οποίους έχουν προστεθεί διάφορα χαρακτηριστικά ώστε να μπορούν να υποστηρίξουν ένα ευρύ φάσμα εφαρμογών. Οι CPU ανήκουν στην κατηγορία των χωρικών αρχιτεκτονικών (spatial architectures) όπου η υπολογιστική δομή αποτελείται από πολλαπλές επεξεργαστικές μονάδες επεξεργασίας. Αυτές οι μονάδες μπορούν να έχουν εσωτερικό έλεγχο (internal control), αρχεία καταχωρητών (register files - RF) για την αποθήκευση δεδομένων και να είναι διασυνδεδεμένες μεταξύ τους για την ανταλλαγή δεδομένων. Οι διανυσματικές-CPU (Vector CPUs) διαθέτουν πολλαπλές μονάδες ALU που μπορούν να επεξεργάζονται παράλληλα πολλαπλά δεδομένα. Οι περισσότερες από αυτές υιοθετούν το μοντέλο εκτέλεσης Single-Instruction Multiple-Data (SIMD), το οποίο εφαρμόζει μεμονωμένα ρεύματα εντολών σε πολλάπλα στοιχεία δεδομένων ταυτόχρονα. Ειδιαίτερο ενδιαφέρον έχει η μελέτη του [17], στην οποία εξετάζονται διάφορες τεχνικές για τη βελτιστοποίηση εφαρμογών βαθιάς μάθησης σε κινητά, server και cluster πολλαπλών CPU.

Για την επίτευξη μιας γρήγορης και αποτελεσματικής εξαγωγής συμπερασμάτων (inference) η/και εκπαίδευσης μοντέλων Βαθιάς Μάθησης χρησιμοποιείται ενας συνδιασμός των διαθέσιμων τεχνολογιών οπου η CPU συνεργάζεται με κάποιο επιταχυντή γενικού η ειδικού σκοπού (π.χ GPU, IPU). Η CPU μπορεί να θεωρηθεί ως ο βασικός ορχηστρωτής των εργασιών ολόκληρου του συστήματος, συντονίζοντας ένα ευρύ φάσμα υπολογιστικών εργασιών γενικής χρήσης, ενώ η GPU εκτελεί ένα στενότερο φάσμα πιο εξειδικευμένων εργασιών. Οι μονάδες GPU διαφέρουν απο τους παραδοσιακούς επεξεργαστές μηχανικά και δομικά. Στο σχήμα 3.1, παρουζιάζονται σχηματικά οι επεξεργαστικές μονάδες μιας πολυπύρηνης CPU και μιας GPU. Μία GPU μπορεί να παριέχει χιλιάδες εξειδικευμένους πυρήνες επεξεργασίας δεδομένων και αρκετά υψηλή μνήμη που φτάνει τα 64 η 128 GB. Η υψηλή πυκνότητα των πυρήνων καθιστά τις GPU ιδανικές για εκτέλεση Νευρωνικών Δικτύων (Neural Networks) στα οποία μπορούν να υπολογιστούν παράλληλα πολλοί νευρώνες καθώς μεταφράζονται υπολογιστικά σε πολλαπλασιασμό και προσθήκη διανυσμάτων.

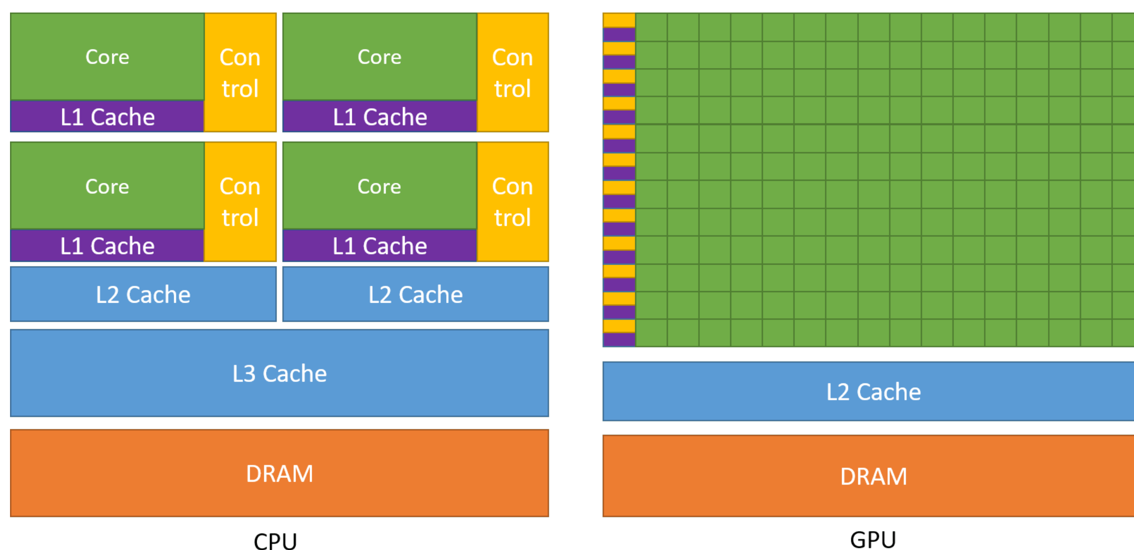



**Figure 3.1.** *Schematic comparison between the chip layout of a multi-core CPU and GPU [18]*

## 3.2 Graphcore Intelligence Processing Unit (IPU)

Η μονάδα IPU αποτελεί ένα νέο είδος επιταχυντή ειδικού σκοπού που αναπτύχθηκε απο την Graphcore για εφαρμογές Τεχνητής Νοημοσύνης και Βαθιάς Μάθησης. Η IPU είναι ένας τύπος επεξεργαστή με διαφορετική αρχιτεκτονική από αυτή των GPU. Ένα σύστημα IPU αποτελείται απο τέσσερα βασικά δομικά στοιχεία: IPU-tile, IPU-exchange, IPU-links και διεπαφές PCIe.

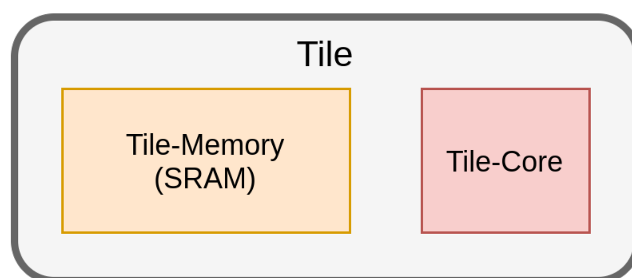



**Figure 3.2.** *IPU-tile*

Το βασικό δομικό στοιχείο ενός συστήματος IPU ειναι το πλακίδιο-IPU (IPU-tile). Κάθε μονάδα IPU διαθέτει πολλαπλά πλακίδια-tiles όπου το κάθε tile αποτελείται απο εναν multi-threaded IPU-επεξεργαστή (ipu-core) και την τοπική μνήμη (Σχήμα 3.2). Το IPU προσφέρει μικρές και κατανεμημένες μνήμες (SRAM) που είναι τοπικά συνδεδεμένες μεταξύ τους μέσω ενός πολύ γρήγορου all-to-all δικτύου επικοινωνίας που ονομάζεται IPU-exchange. Πρόκειτε για ένα on-chip δίκτυο που επιτρέπει την αποδοτική επικοινωνία σε ζώνες υψηλού εύρους (high-bandwidth) μεταξύ των tiles. Επιπλέον, κάθε μονάδα IPU χρησιμοποιεί δύο διεπαφές PCIe για την ανταλλαγή δεδομένων με τον κεντρικό επεξεργαστή CPU του συστήματος. Τέλος,

κάθε IPU περιέχει δέκα διασυνδέσεις IPU-Links που επιτρέπουν την απευθείας ανταλλαγή δεδομένων μεταξύ πολλαπλών IPU, χωρίς αυτά να περνούν από τον κεντρικό επεξεργαστή (CPU) ή τη κεντρική μνήμη. Ένα σύστημα μπορεί να διαθέτει πολλαπλές μονάδες IPU. Μια Multi-IPU είναι μια εικονική συσκευή IPU που αποτελείται από πολλαπλές φυσικές IPU και προσφέρει όλους τους πόρους μνήμης και υπολογισμού τους σαν να ανήκαν σε μια ενιαία συσκευή. Στο Σχήμα 3.3 παρουσιάζεται ένα απλουστευμένο διάγραμμα της αρχιτεκτονική ενός επιταχυντή IPU.

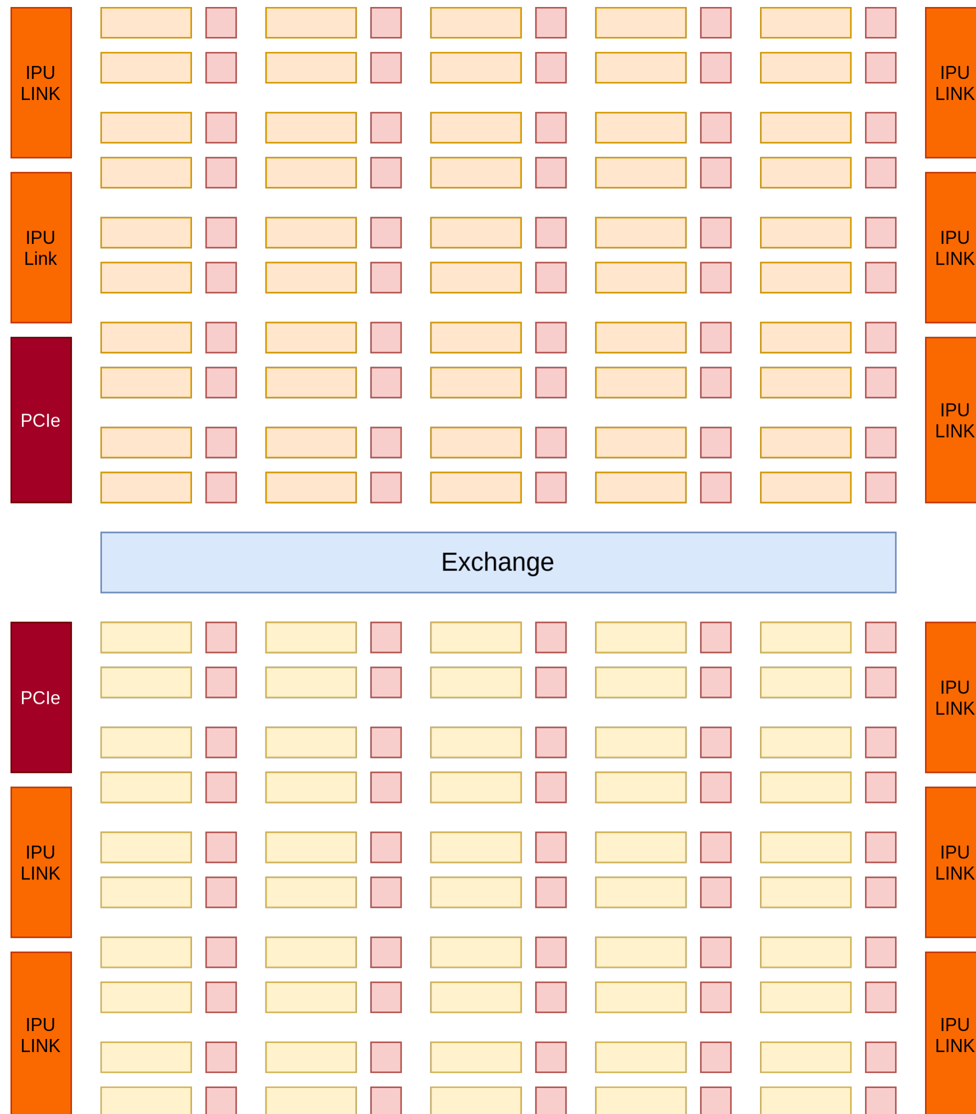



**Figure 3.3.** *Απλουστευμένο διάγραμμα της αρχιτεκτονική ενός επιταχυντή IPU.*

### 3.2.1 Προγραμματιστικό μοντέλο IPU

Ο χρήστης θα πρέπει αρχικά να καθορίσει το σύνολο τον φυσικών IPU που θα εκτελέσουν το πρόγραμμα-IPU και το σύνολο αυτό δεν μπορεί να αλλάξει κατά τη διάρκεια της εκτέλεσης του προγράμματος. Έπειτα, το πρόγραμμα μεταγλωτίζεται σε ένα σύνολο εντολών

που καταλαβαίνει ο επεξεργαστής IPU. Το παραγόμενο πρόγραμμα-IPU ακολουθεί μια διαδρομή καθορισμένης ροής ελέγχου και εκτελεί υπολογισμούς σε μεγάλους πολυδιάστατους τυποποιημένους πίνακες δεδομένων, σταθερού μεγέθους, που ονομάζονται Τανυστές (Tensors). Στη γενική περίπτωση, οι τανυστές είναι δομές δεδομένων που χρησιμοποιούνται για την περιγραφή βαθμωτών, διανυσμάτων και πινάκων.

Υπάρχουν δύο βασικές εντολές που χρησιμοποιεί ένα εκτελούμενο πρόγραμμα-IPU για να χειριστεί τις μεταβλητές τανυστών: αντιγραφή δεδομένων και εκτέλεση υπολογιστικών συνόλων (compute sets). Κάθε υπολογιστικό σύνολο αποτελείται από πολλές κορυφές (vertices) οι οποίες περιγράφουν τις υπολογιστικές εργασίες ως προς εκτέλεση. Οι κορυφές καθορίζουν τον τρόπο με τον οποίο ένα υπολογιστικό σύνολο χωρίζει τις εργασίες του σε μικρότερα κομμάτια υπολογισμών τα οποία μπορούν να εκτελεστούν παράλληλα απο πολλαπλά IPU-tiles. Κάθε κορυφή συνδέεται με ένα συγκεκριμένο πλακίδιο της IPU και εκτελεί ένα μικρό κομμάτι κώδικα που επεξεργάζεται μόνο τη δική του είσοδο και έξοδο δεδομένων. Ο συνδυασμός όλων των κορυφών από τα πολλαπλά υπολογιστικά σύνολα ενός προγράμματος σχηματίζει τον υπολογιστικό γράφο (Σχήμα 3.4).

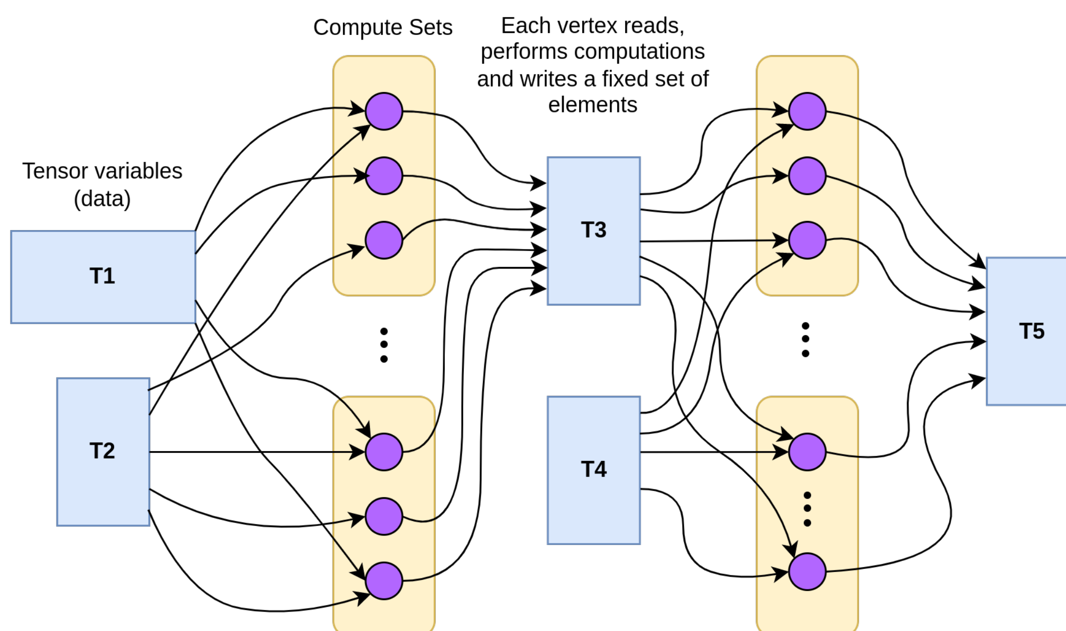



**Figure 3.4.** *Graph representation of variables and processing*

.

### 3.2.2 Μοντέλο εκτέλεσης εργασιών μιας μονάδας IPU

Η IPU ακολουθεί το μοντέλο εκτέλεσης Bulk-synchronous Parallel (BSP) [19] στο οποίο η εκτέλεση εργασιών πανώ στα tiles οργανώνεται σε πολλαπλά διαδοχικά βήματα. Το κάθε βήμα αποτελείται απο τρία στάδια:

- Τοπικος υπολογισμός (local computation): Στο στάδιο αυτό όλα τα tiles εκτελούν υπολογισμούς παράλληλα αποκλειστικά στα δεδομένα που βρίσκονται στην τοπική τους μνήνη SRAM.
- Συγχρονισμός (Synchronisation): Το στάδιο αυτό εξασφαλίζει ότι όλα τα tiles έχουν ολοκληρώσει τους τοπικούς υπολογισμούς τους.
- Ανταλλαγή Δεδομένων (Data-Exchnage): Το τελικό στάδιο είναι η ανταλλαγή απαραίτητων δεδομένων εφόσον όλα τα tiles εχουν εισέλθει στο στάδιο συγχρονισμού.

Η συνολική διαδικασία επαναλαμβάνεται καθώς όλα τα tiles εισέρχονται εκ νέου στη φάση τοπικού υπολογισμού. Κάθε ένα από αυτά τα βήματα πραγματοποιείται παράλληλα σε όλα τα tiles και η μονάδα IPU μπορεί να θεωρηθεί ότι εκτελεί μια ακολουθία αυτών των βημάτων (Σχήμα 3.5). Η σειρά των προς εκτέλεση βημάτων καθορίζεται από ένα πρόγραμμα ελέγχου που φορτώνεται σε κάθε tile από την CPU για τον έλεγχο της εκτέλεσης υπολογισμών και ανταλλαγής δεδομένων.

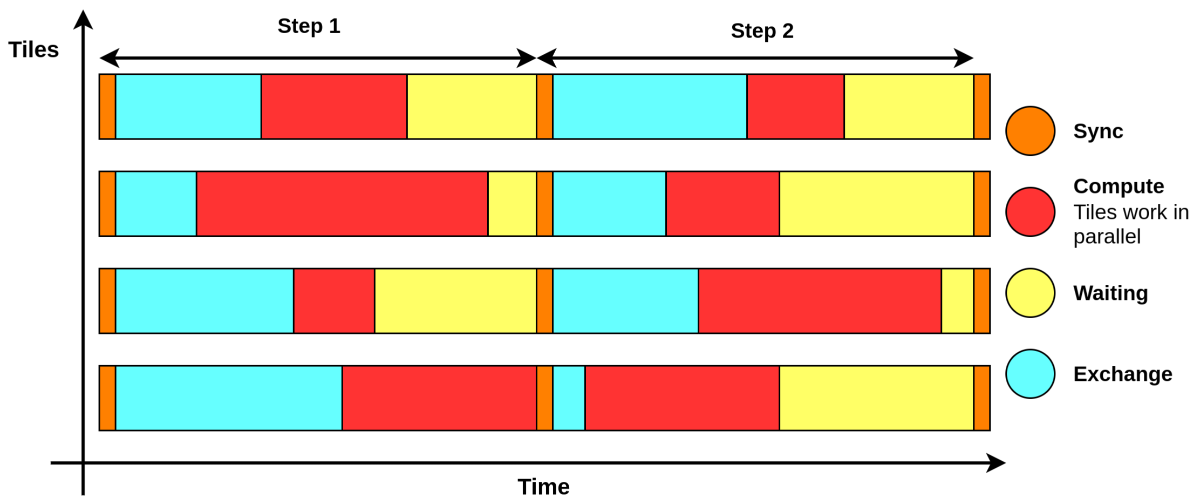



**Figure 3.5.** *Bulk-synchronous parallel execution model across IPU tiles*

.

Κεφάλαιο 4

# Επιλογή αλγορίθμων

Ένα σημαντικό κομμάτι της εργασίας μου ήταν η επιλογή κατάλληλων μεθόδων ανίχνευσης προσώπου (face detection) και χαρακτηριστικών προσώπου (landmark detection). Οι επιλεγμένοι αλγόριθμοι θα πρέπει να είναι σε θέση να ανιχνεύσουν με ακρίβεια την απόκριση των βλεφάρων στις ρυθμίσεις καταγραφής μας, αλλά και να πληρούν πρόσθετες απαιτήσεις ως προς την ταχύτητα ανίχνευσης. Σε αυτό το κεφάλαιο, εξετάζουμε και αξιολογούμε εναλλακτικές μεθόδους τόσο για την ανίχνευση προσώπου όσο και για την ανίχνευση οροσήμων. Οι μετρικές αξιολόγησης που χρησιμοποιήθηκαν είναι ο χρόνος εκτέλεσης για την εξαγωγή μιας ανίχνευσης (detection speed) αλλά και η ακρίβεια/ορθότητα αυτής (accuracy). Επιπρόσθετα, η διαθεσιμότητα προ-εκπαιδευμένων μοντέλων και υλοποιήσεις ανοιχτού κώδικα (open-source) αποτελούν εξίσου σημαντικά κριτήρια για την επιλογή μας.

## 4.1 Προδιαγραφές δεδομένων

Τα δεδομένα στα οποία τελικά θα εκτελεστεί η εφαρμογή μας ακολουθούν μια σειρά προδιαγραφών σχετικά με το περιβάλλον καταγραφής αλλά και με τη θέση-στάση του ανθρώπου κατα τη διάρκεια αυτής. Η περιστροφή στις τρεις χωρικές διαστάσεις ορίζονται ως yaw, roll και pitch και δεδομένου οτι ο άνθρωπος έχει τη προσοχή του εστιασμένη στην κάμερα, οι τιμές τους περιορίζονται σε $\pm 20°$, $\pm 25°$και $\pm 40°$αντίστοιχα. Το περιβάλλον στο οποίο καταγράφονται τα βίντεο είναι καλά φωτισμένο και το υποκείμενο κάθεται αρκετά κοντά στην κάμερα ώστε το πρόσωπο του να καλύπτει τουλάχιστον το 20% της εικόνας. Τέλος, ο ρυθμός καταγραφής (Frames per Second - FPS) της κάμερας ορίζεται στα 500 FPS το οποίο σημαίνει ότι ο μέγιστος χρόνος επεξεργασίας για κάθε καρέ είναι 2 ms.

## 4.2 Ανίχνευση Προσώπων

Η ανίχνευση προσώπου αποτελεί ενα υποπεδίο της ανίχνευσης αντικειμένου που χρησιμοποιείται για την αναγνώριση ανθρώπινων προσώπων σε εικόνες ή βίντεο. Στόχος της ανίχνευσης προσώπου είναι ο εντοπισμός των προσώπων σε μια εικόνα και η σχεδίαση πλαισίων οριοθέτησης (bounding boxes) γύρω από αυτά.

Οι πρώτες προσπάθειες ανίχνευσης προσώπων βασίζονταν κυρίως σε προσεκτικά σχεδιασμένα χαρακτηριστικά απο ερευνητές, τα οποία εξάγονται από μια εικόνα και στη συνέχεια

τροφοδοτούνται σε έναν ταξινομητή (classifier) για την ανίχνευση πιθανών περιοχών που περιέχουν ένα πρόσωπο. Δύο κλασικοί ανιχνευτές προσώπου είναι ο ανιχνευτής προσώπου Haar-cascade των Viola και Jones [12] και Histogram of Oriented Gradients (HOG) σε συνδιασμό με εναν ταξινομητή SVM [11]. Ενώ οι μεθόδοι αυτοί παρουσίασαν εξαιρετικές επιδόσεις, η δυσκολία κατασκευής και κυρίως η ανάγκη για βελτίωση της επίδοσης των συστημάτων αναγνώρισης προσώπου οδήγησαν στην ανάπτυξη νέων μεθόδων που προσφέρουν πολύ καλύτερα αποτελέσματα.

Η χρήση μηχανικής μάθησης για το πρόβλημα της αναγνώρισης προσώπου γενικότερα έχει οδηγήσει σε τεράστια βελτίωση επιδόσεων των συστημάτων αναγνώρισης. Οι τεχνικές μηχανικής μάθησης χρησιμοποιούν νευρωνικά δίκτυα για την εξαγωγή αρκετά μεγάλου αριθμού χαρακτηριστικών ενός προσώπου, αποσπώντας λεπτομερείς πληροφορίες για ένα πρόσωπο μέσω μιας εικόνας. Η εξαγωγή αυτή πραγματοποιείται μέσω μίας διαδικασίας εκπαίδευσης των δικτύων, παρέχοντας σε αυτά την αυτονομία να εντοπίσουν και να καθορίσουν τα σημαντικότερα χαρακτηριστικά της εικόνας. Τα νευρωνικά δίκτυα, και πιο συγκεκριμένα τα Συνελικτικά Νευρωνικά Δίκτυα (CNN) έχουν επιτύχει αξιοσημείωτες επιδόσεις σε διάφορα πεδία της υπολογιστικής όρασης, όπως η ταξινόμηση εικόνων [20] και η αναγνώριση προσώπων [21], [22]. Τα τελευταία χρόνια έχει προταθεί μεγάλος αριθμός εξειδικευμένων στην ανίχνευση προσώπων μοντέλων [23], τα οποία χωρίζονται σε δύο βασικές κατηγορίες ως προς τη δομή τους:

- Τα **μοντέλα ενός σταδίου** (one-step models), όπως τα S3FD[24], RetinaFace[25] and BlazeFace[10], χρησιμοποιούν ένα feed forward CNN [13] για να προσδιορίσουν την κατηγορία (class) των αντικειμένων ενδιαφέροντος καθώς και την ακριβή τοποθεσία τους στην εικόνα (πλαίσιο οριοθέτησης).

- Τα **μοντέλα δύο σταδίων** (two-step models ή region-based models), βασίζονται στην λειτουργεία των Regional CNN (R-CNN) [26]. Τα μοντέλα αυτά χρησιμοποιούν, σαν πρώτο βήμα, έναν αλγόριθμο αναζήτησης (π.χ. Selective Search [27]) ή ένα μοντέλο (συνήθως ένα region-based CNN) το οπόιο δέχεται μια εικόνα σαν είσοδο και προτείνει διαφορετικές πιθανές περιοχές ενδιαφέροντος (Regions of Interest - RoI). Στην συνέχεια (δεύτερο βήμα), χρησιμοποιείται ένα Συνελικτικό Νευρωνικό Δίκτυο ως feature extractor ώστε να υπολογίσει χαρακτηριστικά απο αυτές τις προτάσεις και να εντοπίσει την ακριβή θέση των προσώπων (πλαίσια οριοθέτησης). Παραδείγματα τέτοιων μοντέλων είναι τα CMS-RCNN[28], R-FCN[29] και "Face Detection Using Improved Faster RCNN" [30].

Το γεγονός ότι στη κατηγορία μοντέλων ενός σταδίου δεν πραγματοποιούνται region proposals τα καθιστά απλούστερα και ταχύτερα, αλλά η απόδοσή τους είναι μειωμένη σε σύγκριση με τα μοντέλα δύο σταδίων. Δεδομένου ότι το έργο μας στοχεύει τόσο στην ακρίβεια των αποτελεσμάτων όσο και στην ταχύτητα επεξεργασίας, αποφασίσαμε να μην διερευνήσουμε τους ανιχνευτές δύο σταδίων καθώς δεν έχουν σχεδιαστεί για real-time εφαρμογές. Αντίθετα, οι ανιχνευτές ενός σταδίου έχουν επιτυχώς χρησιμοποιηθεί σε real-time εφαρμογές [31] [32].

### 4.2.1 Επιλεγμένοι αλγόριθμοι

Για τους σκοπούς της παρούσας Πτυχιακής Εργασίας εξερευνήθηκαν τρείς διαφορετικοί ανιχνευτές.

#### HOG - (Histogram of Oriented Gradients)

Ο πρώτος αλγόριθμος που επιλέχθηκε είναι ο αλγόριθμος Histogram of Oriented Gradients (HOG). Μια προ-εκπαιδευμένη υλοποίηση του ανιχνευτή είναι διαθέσιμη μέσω της βιβλιοθήκης ανοιχτού κώδικα Dlib [33]. Ο ανιχνευτής HOG αρχικά εξάγει χαρακτηριστικά από τις εικόνες εισόδου, τα οποία στη συνέχεια τροφοδοτούνται σε έναν ταξινομητή SVM. Ο ταξινομητής SVM χρησιμοποιεί μια τεχνική ολισθαίνοντος παραθύρου πάνω στην εικόνα για την ανίχνευση προσώπων σε μια περιοχή 80x80 εικονοστοιχείων. Ο αλγόριθμος HOG έχει χρησιμοποιηθεί ευρέως στην όραση υπολογιστών για την ανίχνευση προσώπων και μπορεί να είναι αποτελεσματικός υπό διαφορετικές συνθήκες φωτισμού και πόζας.

#### MTCNN - (Multi-task Cascaded Convolutional Network)

Ο δεύτερος αλγόριθμος ονομάζεται Multi-Task Cascaded Convolutional Neural Networks ή MTCNN, ένα συνελικτικό νευρωνικό δίκτυο (CNN) το οποίο δημοσιεύθηκε το 2016 από τους Zhang et al.[34]. Το μοντέλο αποτελείται από τρια νευρωνικά υπο-δίκτυα τα οποία εκτελούνται σειριακά για την ανίχνευση και ανάλυση προσώπων σε μια εικόνα.

Το πρώτο υποδίκτυο ονομάζεται P-Net και είναι ένα μικρό συνελικτικό νευρωνικό δίκτυο. Το δίκτυο αυτό λαμβάνει την εικόνα εισόδου και παράγει ένα σύνολο οριοθετημένων πλαισίων που είναι πιθανό να περιέχουν ένα πρόσωπο. Το δεύτερο υποδίκτυο ονομάζεται R-Net και είναι ένα ελαφρώς μεγαλύτερο CNN που λαμβάνει την έξοδο του P-Net και βελτιώνει τις παραγόμενες θέσεις των πλαισίων οριοθέτησης στην εικόνα. Το τρίτο και τελευταίο υποδίκτυο, που ονομάζεται O-Net, είναι ένα μεγαλύτερο CNN που χρησιμοποιείται για τον εντοπισμό ορόσημων του προσώπου. Το O-Net λαμβάνει την έξοδο του R-Net και παράγει ένα σύνολο από ορόσημα που αντιστοιχούν σε βασικά σημεία του προσώπου, όπως οι γωνίες των ματιών, η άκρη της μύτης και οι γωνίες του στόματος.

#### BlazeFace

Ο τρίτος και τελευταίος αλγόριθμος που επιλέχθηκε ονομάζεται BlazeFace, ένας ελαφρύς και αποδοτικός ανιχνευτής προσώπων που αναπτύχθηκε από την Google το έτος 2020 για εφαρμογές που στοχέυουν σε επεξεργασία δεδομένων σε πραγματικό χρόνο. Ανήκει στην κατηγορία των μοντέλων ενός σταδίου και λαμβάνει ως είσοδο εικόνες RGB μεγέθους $128 \times 128 \times 3$ εικονοστοιχείων (pixel). Η έξοδος του μοντέλου για κάθε πρόσωπο που ανιχνεύεται, είναι τέσσερις συντεταγμένες πλαισίου οριοθέτησης, έξι προσεγγιστικές συντεταγμένες σημείων/οροσήμων προσώπου και ένα σκορ εμπιστοσύνης για την κάθε ανίχνευση.

## 4.3 Αξιολόγηση Μεθόδων Ανίχνευσης Προσώπου

Προκειμένου να προσδιοριστεί ποιος από τους αλγορίθμους που περιγράφηκαν προηγουμένως είναι καταλληλότερος για την εργασία αυτή, εκτελέσαμε πειράματα αξιολόγησης με βάση την ακρίβεια των ανιχνεύσεων και την ταχύτητα επεξεργασίας.

### 4.3.1 Σύνολα Δεδομένων

Για την αξιολόγηση των επιλεγμένων αλγορίθμων χρησιμοποιήσαμε δύο σύνολα δεδομένων (datasets) που περιλαμβάνουν εικόνες με ένα η περισσότερα πρόσωπα.

#### AFLW database

Το πρώτο σύνολο δεδομένων που χρησιμοποιήθηκε στην εργασία μας ονομάζεται AFLW [35] το οποίο αποτελείται από 21123 εικόνες και 24384 ανθρώπινα πρόσωπα. Το συγκεκριμένο σύνολο δεδομένων διαθέτει ένα εκτεταμένο σύνολο ετικετών που εκτός απο την θέση των προσώπων (ground-truth labels) περιγράφουν διάφορα χρήσιμα χαρακτηριστικά όπως το φύλο, την απόκρυψη (occlusion), πρόσωπα με η χωρίς γυαλιά και τις γωνίες περιστροφής (roll, pitch και yaw).

#### AFLW subset

Για την διαδικασία αξιολόγησης των ανιχνευτών επιλέξαμε να φιλτράρουμε τις εικόνες της βάσης AFLW και να δημιουργήσουμε ένα υποσύνολο δεδομένων το οποίο ικανοποιεί τις προδιαγραφές της εργασίας μας. Η εκτενής λίστα ετικετών που διαθέτει η βάση AFLW διευκόλυνε την διαδικασία αυτή. Τα βασικά κριτήρια επιλογής για το αν ενα πρόσωπο σε μια εικόνα θα εισαχθεί στο υποσύνολο μας είναι τα αποδεκτά όρια περιστροφής στις τρείς χωρικές διαστάσεις (yaw $\pm 20\circ$, roll$\pm 25\circ$, pitch$\pm 40\circ$). Επιπλέον, κάποιες εικόνες περιέχουν πολλαπλά πρόσωπα με αποτέλεσμα κάποια απο αυτά να πληρούν τις προυποθέσεις της εργασίας αυτης, ενώ κάποια άλλα στην ίδια εικόνα όχι. Σε αυτή την περίπτωση, η εικόνα αρχικά παραμένει στο υποσύνολο δεδομένων μας. Εάν μία από τις επιλεγμένες μεθόδους δεν μπορεί να ανιχνεύσει κάποιο απο τα πρόσωπα στην εικόνα τότε το πρόσωπο αυτό αποκλείεται από το υποσύνολο για να εξασφαλίστεί η δίκαιη αξιολόγηση των μεθόδων.

#### BioID database

Το δεύτερο σύνολο δεδομένων ονομάζεται BioID και αποτελείται από 1521 εικόνες 23 διαφορετικών προσώπων. Οι συνθήκες στις οποίες καταγράφηκαν οι εικόνες αυτές είναι αρκετά παρόμοιες με τις συνθήκες καταγραφής που αναλύθηκαν στη Ενότητα. Επομένως, η δοκιμή ενός ανιχνευτή προσώπων στο BioID μπορεί να παρέχει μια ισχυρή ένδειξη για το πόσο καλά μπορεί να αποδώσει στο δικό μας σύνολο δεδομένων (test-set).

#### Προεπεξεργασία δεδομένων εισόδου

Οι ανιχνευτές HOG και MTCNN μπορούν να επεξεργαστούν οποιαδήποτε εικόνα εισόδου ανεξαρτήτως μεγέθους. Αντίθετως το μοντέλο BlazeFace δέχεται ως είσοδο εικόνες τυποποιη-

μένου μεγέθους $128 \times 128 \times 3$ pixel. Για τον λόγο αυτό, προσαρμόσαμε το μέγεθος των δεδομένων εισόδου σε $128 \times 128$ για την αξιολόγηση και των τριών ανιχνευτών.

### 4.3.2 Επίδοση ανιχνευτών στο υποσύνολο δεδομένων AWFL

Για την αξιολόγηση των ανιχνευτών μας χρησιμοποιήθηκαν οι μετρικές Ακρίβειας (Precision) και Ανάκλησης (Recall). Πριν δώσουμε τον ορισμό αυτών των μετρικών, θα πρέπει αρχικά να οριστούν οι έννοιες των True Positive, False Positive και False Negative.

Ως True Positive θεωρούμε την ανίχνευση ενός υπάρχοντος προσώπου για την οποία το IoU μεταξύ του πλαισίου οριοθέτησης της ανίχνευσης και του πλαισίου οριοθέτησης του ground truth είναι μεγαλύτερο απο 50%. Θεωρούμε οτι έχουμε μια False Positive πρόβλεψη όταν ο ανιχνευτής εσφαλμένα ανίχνευσε μια περιοχή εικόνας ως πρόσωπο ενώ δεν είναι. Ως False Negative ορίζουμε τις περιπτώσεις στις οποίες δεν ανιχνεύτηκε ένα υπάρχον πρόσωπο.

Η Ανάκληση (Recall) περιγράφει πόσα από τα υπαρκτά πρόσωπα εντοπίστηκαν με ακρίβεια, ενώ η Ακρίβεια (Precision) περιγράφει πόσα από τα πρόσωπα που εντοπίστηκαν ήταν πραγματικά πρόσωπα και όχι False Positives. Η ανάκληση και η ακρίβεια υπολογίζονται ως εξής:

$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$TP = true\ positives,\ FN = false\ negatives,\ FP = false\ positives$$

Σύμφωνα με τα παραπάνω η Ακρίβεια ισούται με το λόγο των σωστών προβλέψεων προς τις συνολικές προβλέψεις ενώ η Ανάκληση ισούται με το λόγο των σωστών προβλέψεων προς το συνολικό αριθμό αντικειμένων. Τέλος να επισημάνουμε οτι για την δίκαιη αξιολόγηση των διαφορετικών επιλεγμένων ανιχνευτών χρησιμοποιήσαμε έναν πυρήνα CPU για την εκτέλεση των πειραμάτων.

Τα αποτελέσματα της αξιολόγησης των τριών ανιχνευτών παρουσιάζονται στον Πίνακα 4.1.

| **Algorithm** | **Faces** | **FN** | **FP** | **TP** | **Precision (%)** | **Recall (%)** | **Time / Image (ms)** |
|---|---|---|---|---|---|---|---|
| HOG | 3425 | 687 | 7 | 2738 | 99.74 | 79.94 | **29.23** |
| MTCNN | 3425 | **294** | 37 | **3131** | 98.83 | **91.41** | 347.41 |
| BlazeFace | 3425 | 940 | **1** | 2485 | **99.96** | 72.56 | 32.73 |

**Table 4.1.** *Results on face detection algorithms on subset of AFLW that meets project requirements*

### 4.3.3 Επίδοση ανιχνευτών στο σύνολο δεδομένων BioID

Η βάση δεδομένων BioID, περιέχει 1521 εικόνες και κάθε εικόνα απεικονίζει ένα μόνο πρόσωπο. Το περιβάλλον καταγραφής και τα χαρακτηριστικά των δεδομένων είναι αρκετά παρόμοια με αυτά στο δικό μας Test Set. Η βάση δεδομένων BioID παρέχει ετικέτες για πέντε ορόσημα (landmarks) προσώπου που περιγράφουν την θεση των ματιών, αλλά δεν παρέχονται groundtruth ετικέτες για τα πλάσια οριοθέτησης (bounding boxes) των προσώπων.

Η αξιολόγηση των ανιχνευτών BlazeFace και MTCNN έγινε χρησιμοποιώντας τη μετρική σφάλματος Σχετικής-Απόστασης (relative-distance error) που περιγράφεται στο [36] καθώς και οι δύο αυτοί ανιχνευτές παράγουν ανιχνεύσεις που εκτός απο την θεση (πλαίσια οριοθέτησης) του προσώπου περιγραάφουν και την θέση κάποιον σημείων/οροσήμων (landmarks) του προσώπου, και πιο συγκεκριμένα τη θέση του κέντρου του ματιού. Το σφάλμα απόστασης υπολογίστηκε για το κάθε μάτι ξεχωριστα και η μέγιστη τιμή αποτελεί την βαση σύγκρισης. Αν η απόσταση του πραγματικού (groundtruth) και ανιχνευμένου κεντρικού σημείου του ματιού είναι μικρότερη ή ίση με 0,25 τότε η ανίχνευση κρίνεται ως True Positive. Στο Πίνακα 4.2 παρουσιάζονται τα αποτελέσματα αξιολόγησης.

| Algorithm | Faces | Right average error | Left average error | TP | FN |
|---|---|---|---|---|---|
| BlazeFace | 1521 | 0.069% | 0.073% | 1497 | 24 |
| MTCNN | 1521 | 0.03% | 0.043% | 1514 | 10 |

**Table 4.2.** *Results of BlazeFace on BioId database*

Το μοντέλο BlazeFace κατάφερε να ανιχνεύσει σωστά 1497 από το σύνολο των 1521 προσώπων στη βάση δεδομένων BioID. Ωστόσο, μετά από χειροκίνητη επισκόπηση των παραγόμενων ανιχνεύσεων, διαπιστώθηκε ότι για κάθε εικόνα στο σύνολο δεδομένων υπήρχε ένα πλαίσιο οριοθέτησης και κάθε πλαίσιο είχε ένα πρόσωπο στο εσωτερικό του. Έτσι, το μοντέλο BlazeFace έχει 100% επιτυχία στην ανίχνευση προσώπων στο σύνολο δεδομένων δεδομένων BioID. Παρόμοια διαδικασία ακολουθήθηκε για την αξιολόγηση του ανιχνευτή MTCNN ο οποίος επίσης παρήγαγε 100% σωστές προβλέψεις.

Για την αξιολόγηση του ανιχνευτή HOG ήταν απαραίτητη η δημιουργία ετικετών που περιγράφουν την θέση του προσώπου (πλαίσια οριοθέτησης) εντός κάθε εικόνας, καθώς ο εν λόγω ανιχνευτής δεν είναι σε θέση να ανιχνεύει ορόσημα/σημεία (landmarks) του προσώπου. Για τον σκοπό αυτό χρησιμοποιήθηκαν οι παραγόμενες προβλέψεις του μοντέλου MTCNN ως βάση σύγκρισης (ground truth) με τις προβλέξεις εξόδου του ανιχνευτή HOG. Ομοίως, ο ανιχνευτής HOG κατάφερε να ανιχνεύσει σωστά ολα τα πρόσωπα σε κάθε εικόνα του συνόλου BioID.

Τα αποτελέσματα της παραπάνω αξιολόγησης αποτελούν ισχυρή ένδειξη ότι και οι τρεις ανιχνευτές προσώπου θα αποδώσουν καλά στο test set μας.

### 4.3.4 Συμπεράσματα στην αξιολόγηση ανιχνευτών προσώπου

Απο τα πειράματα αξιολόγησης προκύπτουν διαφορές τόσο στην ταχύτητα όσο και στην ακρίβεια μεταξύ των τριών ανιχνευτών. Το μοντέλο MTCNN αποτελεί αξιόπιστη επιλογή καθώς στα πειράματα αξιολόγησης στο υποσύνολο δεδομένων AFLW εμφάνισε τον μεγαλύτερο αριθμό True Positives (3131), επιτυγχάνοντας Recall 91,41% και Precision 98.83%. Ωστόσο, με βάση τα αποτελέσματα αξιολόγησης (Πινακας 4.1) φαίνεται ξεκάθαρα οτι το μοντέλο MTCNN δεν μπορεί να χρησιμοποιηθεί για την εργασία μας καθώς η ταχύτητα με την οποία επεξεργάζεται τα δεδομένα εισόδου υπολογίζεται περίπου στα 333 ms ανα εικόνα.

Οι μεθόδοι ανίχνευσης προσώπου HOG και BLazeFace αποτελούν ταχύτερες εναλλακτικές, με τον HOG να είναι ο πιο γρήγορος ανιχνυετής. Το μοντέλο BlazeFace είναι εντυπωσιακά γρήγορο παρά το γεγονός ότι είναι ένα βαθύ νευρωνικό δίκτυο, επιτυγχάνοντας ταυτόχρονα την υψηλότερη ακρίβεια (Precision - 99.96%) και το μικρότερο αριθμό False Positives (126). Ωστόσο, ένα μειονέκτημα του μοντέλου BlazeFace είναι οτι δεν λειτουργεί καλά σε εικόνες πολύ υψηλής ανάλυσης καθώς απαιτείται προεπεξεργασία των δεδομένων για να προσαρμοστούν στο κατάλληλο μεγεθος εισόδου ($128 \times 128 \times 3$ pixel). Η διαδικασία αυτή μπορεί να οδηγήσει σε κακή απόδοση, καθώς χάνεται πολύτιμη πληροφορία από την εικόνα, ειδικά όταν τα πρόσωπα εντός μιας εικόνας βρίσκονται σε μεγάλη απόσταση από την κάμερα καταγραφής. Αυτό εξηγεί και το χαμηλό ποσοστό Recall (72.56%) και τον υψηλό αριθμό False Negatives (940) που πέτυχε κατα την αξιολόγηση μας στο υποσύνολο δεδομένων της βάσης AFLW. Ωστόσο, η περαιτέρω αξιολόγηση του μοντέλου BlazeFace στο σύνολο δεδομένων BioID είχε ως αποτέλεσμα 100% σωστές ανιχνεύσεις. Δεδομένου οτι το περιβάλλον καταγραφής και τα χαρακτηριστικά των δεδομένων στο BioID είναι αρκετά παρόμοια με αυτά στο δικό μας test Set θεωρούμε ότι το μοντέλο BlazeFace θα έχει παρόμοια απόδοση/συμπεριφορά και στο δικό μας σύνολο δεδομένων.

Ο ανιχνευτής HOG αποτελεί επίσης μια αξιόπιστη επιλογή για την εργασία μας. Με βάση τα αποτελεσματα αξιόληγης είναι ο πιο γρήγορος ανιχνευτής και επιτυγχάνει Recall 80% στο υποσύνολο AFLW και 100% σωστές ανιχνέυσεις στο σύνολο δεδομένων BioID. Αξίζει να σημειωθεί πως ο ανιχνευτής HOG έχει χρησιμοποιηθεί με επιτυχία για την ανιχνευση προσώπων σε πραγματικό χρόνο [37]. Με βάση τα αποτελέσματα που μας παρέχονται φαίνεται οτι η εκτέλεση του ανιχνευτή HOG πάνω σε μια GPU μπορεί να επιτύχει ταχύτητα επεξεργασίας 0.289 ms ανα εικόνα.

Ωστόσο, ένας πρόσθετος στόχος της εργασίας μας είναι να διερευνηθούν υλοποιήσεις ανιχνευτών από τον τομέα της βαθιάς μάθησης, καθώς τα τσιπ IPU και GPU μπορουν να επιταχύνουν σημαντικά την εκτέλεσή τους.

Για τους προαναφερθέντες λόγους, επιλέξαμε να χρησιμοποιήσουμε το μοντέλο **BlazeFace** για το στάδιο ανίχνευσης προσώπου. Η αναλυτική περιγραφή του ανιχνευτή BlazeFace παρατίθεται στο αγγλικό τμήμα της εργασίας.

## 4.4 Eyelid-closure detection

Μόλις ολοκληρωθεί το στάδιο ανίχνευσης προσώπου και ληφθεί το πλαίσιο οριοθέτησης ξεκινάει η διαδικασία καθορισμού του ποσοστού κλεισίματος των βλεφάρων. Κάθε άνθρωπος διαφέρει ως προς τον τρόπο που ανοιγοκλείνει τα μάτια του. Οι διαφορές που εμφανίζονται συχνά αφορούν την ταχύτητα με την οποία κλείνουν η ανοίγουν τα μάτια και τον βαθμό σύσφιξης (squeezing).

#### Μετρική Eye Aspect Ratio (EAR)

Για να προσδιοριστεί πόσο ανοιχτά η κλειστά είναι τα μάτια σε μια δεδομένη χρονική στιγμή, χρειάζονται έξι σημεία που περιγράφουν την θέση των βλεφάρων στην εικόνα: δύο στις γωνίες, δύο στο άνω βλέφαρο και δύο στο κάτω βλέφαρο όπως φαίνεται στο Σχήμα 4.1. Εάν αυτά τα σημεία εντοπιστούν με ακρίβεια, τότε μπορούν να αξιοποιηθούν για τον υπολογισμό μιας μετρικής που ονομάζεται Eye Aspect Ratio (EAR) [38].

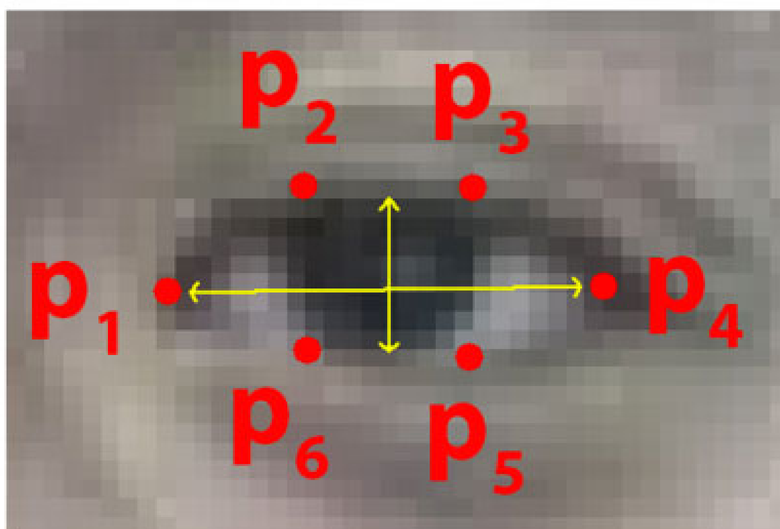



**Figure 4.1.** *The 6 facial landmarks associated with the eye. Image from [38]*

Η τιμή της μετρικής EAR υπολογίζεται απο την εξίσωση 4.1. Ο αριθμητής αυτής της εξίσωσης υπολογίζει την απόσταση μεταξύ των κατακόρυφων σημείων (p2,p3,p5,p6), ενώ ο παρονομαστής υπολογίζει την απόσταση μεταξύ των οριζόντιων ορόσημων σημείων (p1, p4). Οι τιμές που λαμβάνει η εν λόγω μετρική κυμαίνονται κοντά στη τιμή ένα οταν τα μάτια είναι ανοιχτά και πλησιάζει το μηδέν οταν ειναι κλειστά. Αξίζει να αναφερθεί επίσης οτι ο λόγος διαστάσεων (aspect ratio) του ανοιχτού ματιού έχει μικρή διακύμανση μεταξύ διαφορετικών ανθρώπων και δεν επηρεάζεται απο την κλίμακα της εκάστοτε εικόνας και την περιστροφή του προσώπου στο επίπεδο.

$$EAR = \frac{||p2 - p6|| + ||p3 - p5||}{2 * ||p1 - p4||} \tag{4.1}$$

where p1, ... , p6 are the six 2D landmark locations of the eye.

Χρησιμοποιώντας αυτή την απλή εξίσωση, μπορούμε να αποφύγουμε πολύπλοκες τεχνικές επεξεργασίας εικόνας και απλά να βασιστούμε στον λόγο των αποστάσεων των βλεφάρων για να προσδιορίσουμε αν ένα άτομο ανοιγοκλείνει τα μάτια του. Για την ανίχνευση των παραπάνω σημείων που περιγράφουν την θέση των βλεφάρων σε κάθε μάτι θα χρησιμοποιήσουμε έναν ανιχνευτή οροσήμων.

### 4.4.1 Ανίχνευση οροσήμων προσώπου - Landmark detection

Η ανίχνευση ορόσημων προσώπου, γνωστή και ως ανίχνευση χαρακτηριστικών προσώπου, είναι η διαδικασία εντοπισμού συγκεκριμένων σημείων σε ένα πρόσωπο, όπως οι γωνίες του στόματος, η άκρη της μύτης και το κέντρο των ματιών. Οι περισσότεροι αλγόριθμοι ανίχνευσης ορόσημων προσώπου λαμβάνουν ως είσοδο την θέση του προσώπου. Επομένως, η διαδικασία εντοπισμού ορόσημων προσώπου είναι μια διαδικασία δύο βημάτων: εντοπισμός της περιοχής του προσώπου στη εικόνα, δηλαδή οι συντεταγμένες του πλαισίου οριοθέτησης, την οποία λαμβάνει και επεξεργάζεται ο ανιχνευτής οροσήμων. Στο Σχήμα 4.2 φαίνονται δύο παραδείγματα ανίχνευσης ορόσημων.

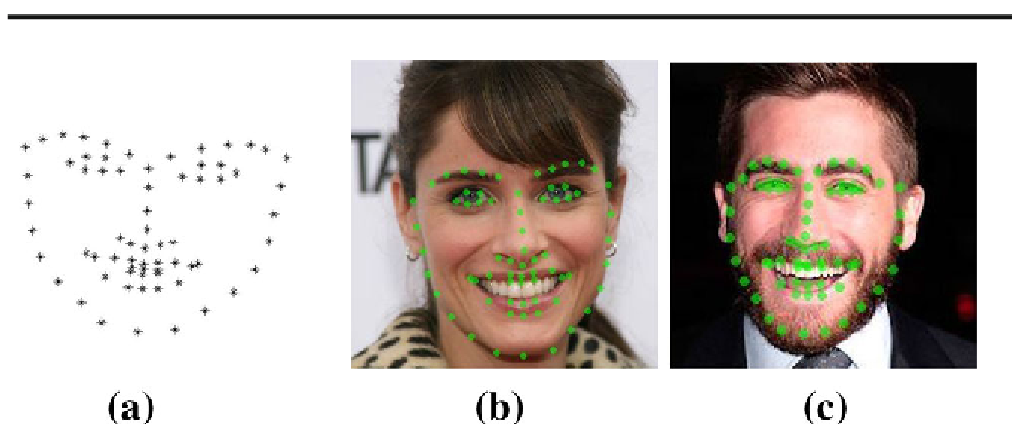



**Figure 4.2.** *Example of 68-landmark detection on two faces. Image from [39].*

### 4.4.2 Επιλογή ανιχνευτή οροσήμων

Η δεύτερη, και ίσως η πιο καθοριστική για το αποτέλεσμα, βασική επιλογή που είχαμε να κάνουμε αφορούσε τον ανιχνευτή οροσήμων.

Ο αλγόριθμος "Ensemble of Regression Trees" (ERT) [40] θεωρείται μία από τις πιο δημοφιλείς μεθόδους για την ανίχνευση ορόσημων. Πρόσφατες εκτεταμένες έρευνες [41],[42] αξιολογούν ότι ο αλγόριθμος ERT είναι ένα αξιόπιστο και γρήγορο μοντέλο ανίχνευσης ορόσημων το οποίο, σύμφωνα με τους συγγραφείς, επιτυγχάνει ταχύτητα επεξεργασίας περίπου 1 ms ανά εικόνα. Η υψηλή ταχύτητά επεξεργασίας που προσφέρει ικανοποιεί το παράθυρο μέγιστου χρόνου επεξεργασίας των 2 ms για την συνολική διαδικασία ανίχνευσης.

Ωστόσο, εκτός από την γρήγορη ταχύτητα επεξεργασίας, είναι εξίσου σημαντικό ο αλγόριθμος που θα επιλεγεί να είναι σε θέση να παρέχει ακριβείς ανιχνεύσεις, ειδικά για τα σημεία που περιγράφουν τη θέση των ματιών. Για το λόγο αυτό, χρησιμοποιήσαμε το σύνολο δεδομένων BioID για να αξιολογήσουμε την ακρίβεια των ανιχνεύσεων που παράγει ο ανιχνευτής ERT. Η αξιολογηση αφορά τη προ-εκπαιδευμένη υλοποίηση που διατίθεται μέσω της βιβλιοθήκης ανοιχτού κώδικα Dlib [33], η οποία έχει ως έξοδο 68 ορόσημα προσώπου (Σχήμα 4.2).

Η μετρική αξιολόγησης που χρησιμοποιήθηκε ονομάζεται inter-ocular distance normalized error [42]. Η βάση δεδομένων BioID παρέχει ετικέτες (labels) που περιγράφουν την groundtruth θέση κάθε ματιού. Για κάθε εικόνα, δίνονται συνολικα 10 ζεύγη συντεταγμένων $(x, y)$, 5 για κάθε μάτι, τα οποία επεξεργαστήκαμε για να υπολογίσουμε το κεντρικό σημείο για κάθε μάτι ξεχωριστά. Αντίστοιχα, απο το σύνολο των 68 ανιχνευμένων σημείων που εξάγει ο ανιχνευτής ERT, χρησιμοποιήσαμε τα έξι σημεία που περιγράφουν την θέση κάθε ματιού για να καθορίσουμε ενα κεντρικό σημείο. Στην συνέχεια, συγκρίναμε το κεντρικό σημείο αυτό με την groundtruth θέση και υπολογίσαμε το σφάλμα για κάθε μάτι ξεχωριστα. Επιπλέον, υπολογίστηκε και ο μέσος χρόνος επεξεργασίας ανα εικόνα. Τα αποτελέσματα παρατίθενται στον Πίνακα 4.3.

| **Landmark Detector** | **Right-Eye error** | **Left-Eye error** | **Time per image** |
|---|---|---|---|
| Dlib's 68-Landmark Detector | 2.61 % | 3.64 % | 1.06 (ms) |

**Table 4.3.** *The evaluation error metric is measured as the euclidean distance of each detected landmark to the true (annotated) landmark, divided by the inter-ocular distance for scale invariance [42].*

Παρατηρούμε οτι ο αλγόριθμος παρουσίασε ικανοποιητικά αποτελέσματα καθώς το σφάλμα δεν ξεπερνά το 4% και ο μέσος χρόνος εκτέλεσης είναι περίπου 1 ms ανα εικόνα. Επιπλέον, αξίζει να αναφερθεί ότι ο αλγόριθμος ERT έχει χρησιμοποιηθεί με επιτυχία για την ανίχνευση της θέσης των ματιών σε εικόνες και τον υπολογισμό της μετρικής Eye Aspect Ratio (EAR) σε πραγματικό χρόνο [37]. Με βάση τα παραπάνω, κρίθηκε οτι ο ανιχνευτής Ensemble of Regression Trees (ERT) [40] είναι κατάλληλος για το στάδιο ανιχνευσης οροσήμων στα πλαίσια της παρούσας εργασίας.

Η αναλυτική περιγραφή του ανιχνευτή ERT παρατίθεται στο αγγλικό τμήμα της εργασίας.

## Κεφάλαιο 5

# Σχεδιασμός και Υλοποίηση

Ο στόχος της εργασίας αυτής είναι η υλοποίηση ενός συστήματος ανίχνευσης βλεφαρίσματος σε πραγματικό χρόνο. Το σύστημα αυτό μπορεί να χωριστεί σε πέντε διακριτά βήματα τα οποία παρουσιάζονται στο Σχήμα 5.1

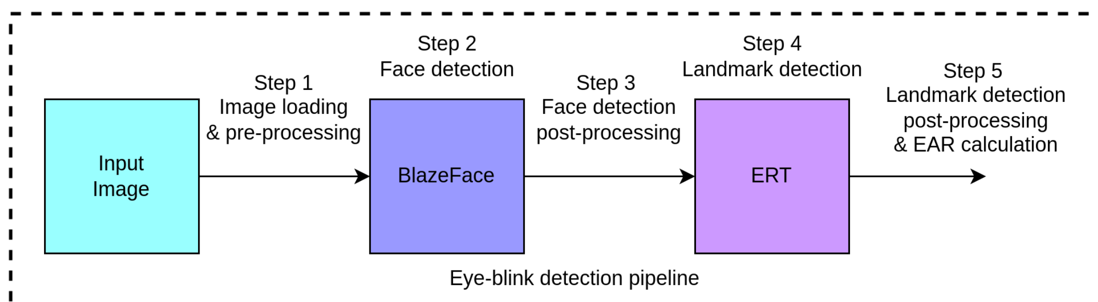



**Figure 5.1.** *Steps of the eyeblink detection pipeline.*

Παρακάτω θα αναλύσουμε συνοπτικά κάθε βήμα ξεχωριστά και θα αναφέρουμε τις τεχνικές βελτιστοποιήσεων που χρησιμοποιήσαμε για καθένα απο αυτά.

## 5.1 Φόρτωση εικόνων και προ-επεξεργασία

Στην τρέχουσα διάταξη, οι εικόνες του συνόλου δεδομένων μας έχουν διάσταση $640 \times 480$ pixels. Όλες οι εικόνες είναι grayscale, πράγμα που σημαίνει ότι ο πίνακας που αναπαριστά την φορτωμένη εικόνα έχει $640 \times 480 \times 1$ εικονοστοιχεία (pixels). Πριν δοθούν σαν είσοδος στο μοντέλο BlazeFace, κανονικοποιούνται ανά batch και η διάστασή τους μετατρέπεται σε $128 \times 128 \times 3$ (RGB εικόνες).

Για την φόρτωση και προ-επεξεργασία των εικόνων χρησιμοποιήθηκε η βιβλιοθήκη Python Imaging Library (PIL). Η PIL είναι μια βιβλιοθήκη ανοικτού κώδικα της γλώσσας προγραμματισμού Python που επιτρέπει την φόρτωση, τον χειρισμό και την αποθήκευση πολλών διαφορετικών μορφών αρχείων εικόνας.

Οι διαδικασίες φόρτωσης και προεπεξεργασίας μιας εικόνας αποτελούν χρονοβόρες εργασίες που μπορεί να διαρκέσουν έως και 2,7 (ms) ανά εικόνα. Για να επιταχύνουμε αυτές τις εργασίες, συνδυάσαμε τη βιβλιοθήκη PIL με τη βιβλιοθήκη Multiprocessing της Python, η οποία επιτρέπει την αξιοποίηση πολλαπλών διεργασιών για παράλληλη εκτέλεση

σε πολυπύρηνους επεξεργαστές.

Για την παράλληλη εκτέλεση εργασιών χρησιμοποιήσαμε την κλάση multiprocessing.Pool η οποία αναπαριστά μια ομάδα διεργασιών. Κάθε διεργασία έχει τον δικό της κατανεμημένο χώρο μνήμης και εκτελείται εντελώς ανεξάρτητα. Γίνεται χρήση της μεθόδου Pool.map(), η οποία δέχεται ως ορίσματα ενα σύνολο δεδομένων και τον αριθμό των διεργασιών καθώς και ένα σύνολο συναρτήσεων που θα πρέπει να εκτελέσειη κάθε διεργασία. Η μέθοδος αυτή χωρίζει το σύνολο δεδομένων σε υποσύνολα, με βάση των αριθμό των διεργασιών που επιλέγεται κάθε φορά, και τροφοδοτεί ένα διαφορετικό υποσύνολο δεδομένων σε κάθε μία απο τις διεργασίες που δημιουργεί. Κάθε γεννημένη διεργασία θα χειριστεί τη φόρτωση και την προεπεξεργασία (αλλαγή διαστάσεων σε 128x128x3 pixels, μετατροπή σε χρωματικό χώρο RGB) του υποσυνόλου δεδομένων που της έχει ανατεθεί και στη συνέχεια θα επιστρέψει τις επεξεργασμένες εικόνες πίσω στην κύρια διεργασία.

## 5.2 Ανίχνευση Προσώπων- BlazeFace

Για το στάδιο της ανίχνευσης προσώπου χρησιμοποιήθηκε μια προ-εκπαιδευμένη υλοποίηση του μοντέλου BlazeFace μέσω της βιβλιοθήκης MediaPipe [43]. Για την κατασκευή και εκτέλεση του μοντέλου, χρησιμοποιήθηκαν οι βιβλιοθήκες βαθίας μάθησης Tensorflow και Keras. Τόσο το Tensorflow, όσο και το Keras αποτελούν βιβλιοθήκες ανοικτού κώδικα και υποστηρίζονται σε πλατφόρμες υλικού IPU και GPU. Πιο συγκεκριμένα για την υλοποίηση σε IPU χρησιμοποιήσαμε την έκδοση 2.4 του IPU Software Development Kit (SDK) που υποστηρίζει την έκδοση TensorFlow 2.4.4. Η ίδια έκδοση του TensorFlow χρησιμοποιήθηκε και για την υλοποίηση σε GPU.

Τα αναλυτικά χαρακτηριστικά των επιταχυντών παρατίθενται στο αγγλικό τμήμα της εργασίας.

### 5.2.1 Inference σε πατφόρμες υλικού MK1 και MK2 IPU

Προκειμένου να επιταχύνουμε αποτελεσματικά το μοντέλο BlazeFace σε ένα τσιπ IPU, έπρεπε να λάβουμε υπόψη μας διάφορους παράγοντες από τους οποίους επηρεάζεται η απόδοση. Τα χαρακτηριστικά των επιταχυντών και το λογισμικό που χρησιμοποιήθηκε για την ανάπτυξη της εφαρμογής παίζουν επίσης βασικό ρόλο στο στάδιο της βελτιστοποίησης. Η μέγιστη δυνατή αξιοποίηση των υπολογιστικών δυνατοτήτων ενός τσιπ IPU, η βελτιστοποίηση της διαχείρισης της μνήμης και η ελαχιστοποίηση της επικοινωνίας μεταξύ κεντρικού υπολογιστή και IPU είναι οι πιο κρίσιμοι παράγοντες που επηρεάζουν την απόδοση του μοντέλου.

Για να εκτελεστεί ένα μοντέλο σε ένα σύστημα IPU πρέπει αρχικά να μετατραπεί ο Python κώδικας σε ένα πρόγραμμα-IPU. Το Tensorflow και το Graphcore χρησιμοποιούν μια διαδικασία πολλών βημάτων για τη δημιουργία ενός προγράμματος-IPU. Η διαδικασία αναλύεται στα ακόλουθα βήματα:

1. **Εξαγωγή ενός υπολογιστικού γράφου (computational graph):** Το Tensorflow εξάγει έναν υπολογιστικό γράφο από τον πηγαίο κώδικα (Python), ο οποίος είναι μια

αναπαράσταση των υπολογισμών που εκτελούνται από το μοντέλο.

2. **Ανάλυση με χρήση XLA:** Το Tensorflow χρησιμοποιεί έναν μεταγλωττιστή ειδικού σκοπού που ονομάζεται XLA (Accelerated Linear Algebra) για να εκτελέσει περάσματα ανάλυσης πάνω στην αναπαράσταση του γράφου υψηλού επιπέδου. Το XLA χρησιμοποιείται για τη βελτιστοποίηση του γράφου σε υπολογισμούς γραμμικής άλγεβρας.

3. **Graphcore XLA backend:** Το Graphcore χρησιμοποιεί ένα προσαρμοσμένο backend XLA για να εκτελεί τους δικούς του μετασχηματισμούς και βελτιστοποιήσεις στον γράφο. Το προσαρμοσμένο backend επιτρέπει στην Graphcore να προσαρμόζει τις βελτιστοποιήσεις στα ειδικά χαρακτηριστικά των τσιπ IPU.

4. **Μετατροπή γράφου σε scheduled-graph:** Στη συνέχεια, ο γράφος αντιστοιχίζεται σε μια αναπαράσταση γράφου χαμηλότερου επιπέδου που ονομάζεται scheduled-graph. Αυτός ο scheduled-graph είναι ένα πλήρως προγραμματισμένο σύνολο λειτουργιών που θα εκτελεστούν σε συγκεκριμένη σειρά.

5. **Μετάφραση σε πρόγραμμα IPU:** O scheduled-graph μεταφράζεται στη συνέχεια σε ένα πρόγραμμα IPU, όπου κάθε κόμβος αντικαθίσταται από την εκτέλεση υπολογιστικών συνόλων (compute sets).

6. **Μοντέλο εκτέλεσης BSP:** Το πρόγραμμα IPU τροποποιείται περαιτέρω σε μορφή που ταιριάζει με το μοντέλο παράλληλης εκτέλεσης BSP (bulk-synchronous parallel) ενός chip IPU, το οποίο αποτελείται από βήματα συγχρονισμού, ανταλλαγής δεδομένων και υπολογισμού.

7. **Μεταγλωττιστής Poplar:** Το νεοδιαμορφωμένο πρόγραμμα χαμηλώνεται και πάλι έτσι ώστε κάθε πλακίδιο (tile) να έχει τη δική του έκδοση του προγράμματος με σαφή βήματα συγχρονισμού και επικοινωνίας. Ο μεταγλωττιστής Poplar (GCD - Graph Compile Domain) χειρίζεται τη τελική έκδοση του προγράμματος σε κάθε tile το οποίο:
   (a) Εκτελεί read/write μόνο σε δεδομένα στη τοπική μνήμη του tile.
   (b) Περιέχει ρητές εντολές συγχρονισμού με άλλα πλακίδια. tiles.
   (c) Περιέχει ρουτίνες επικοινωνίας για την ανταλλαγή δεδομένων με άλλα πλακίδια.

8. **Μεταγλώττιση και εκτέλεση:** Τέλος, το προκύπτον πρόγραμμα που φορτώνεται σε κάθε tile μπορεί να μεταγλωττιστεί με έναν συμβατικό μεταγλωττιστή για να εκτελεστεί.

Παρακάτω φαίνεται η συνολική διαδικασία της μετατροπής (Σχήμα 5.2).

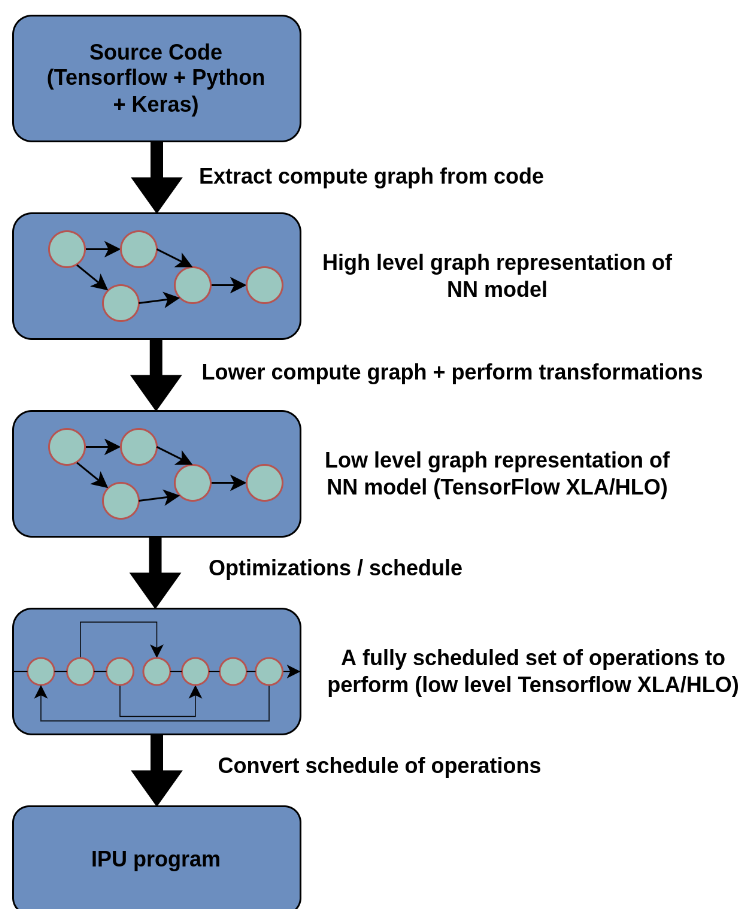



**Figure 5.2.** *A framework lowering to run on an IPU*

Στη συνέχεια θα εξηγηθούν διάφορες τεχνικές βελτιστοποίησης που χρησιμοποιήθηκαν για τη μεγιστιποίηση των επιδόσεων πάνω σε τρείς βασικούς άξονες.

### Maximizing compute capabilities

Για τη βέλτιστη αξιοποίηση της επεξεργαστικής ισχύος, το μέγεθος της παρτίδας (batch size) έπρεπε να επιλέχτεί με τρόπο ώστε το παραγόμενο πρόγραμμα-IPU (υπολογιστικός γράφος) να χωράει στην τοπική μνήμη των tiles και να αξιοποιείται η επεξεργαστική ισχύς του IPU τσιπ. Η απαιτούμενη μνήμη καθορίζεται κατά τη διάρκεια της μεταγλώττισης και αποτελείται από τον κύριο κώδικα (δηλαδή τις πράξεις γραμμικής άλγεβρας), τους τανυστές (Tensors) εισόδου/εξόδου και τον κώδικα ανταλλαγής δεδομένων. Είναι σημαντικό να σημειωθεί εδώ ότι η τιμή του batch-size πρέπει να παραμένει σταθερή καθ'όλη τη διάρκεια εκτέλεσης του μοντέλου μας, διαφορετικά θα προκύψει μια νέα κατασκευή/μεταγλώτιση υπολογιστικού γράφου (re-compilation). Η διαδικασία μετατροπής του πηγαίου κώδικα σε ένα πρόγραμμα-IPU παράγει ένα στατικό υπολογιστικό γράφο και έτσι οι εντολές εκτέλεσης που φορτώνονται στο κάθε tile αλλά και η κατανομή του φόρτου επεξεργασίας εξαρτώνται άμεσα από την τιμή του batch-size.

Η αρχική μας υλοποίηση αφορά τη εκτέλεση του μοντέλου μας σε ένα μεγάλο σύνολο δεδομένων αποθηκευμένο στη κεντρική μνήμη. Για την εκτέλεση και τη μεταγλωττιση του

μοντέλου χρησιμοποιήθηκε η μέθοδος model.predict() της βιβλιοθήκης Keras και το IPUStrategy API. Το IPUStrategy αποτελεί μια υποκλάση του tf.distribute.Strategy API της βιβλιοθήκης Tensorflow για την εκτέλεση εξαγωγής συμπερασμάτων (inference) σε ένα ενιαίο σύστημα με μία ή περισσότερες συνδεδεμένες IPU. Τα αρχικά πειράματα αφορούν την μεταγλώτιση και εκτέλεση του μοντέλου BlazeFace στα MK1 και MK2 IPU τσιπ κάνοντας χρήση διαφόρων μεγεθών batch-size 1,2,4,8,16,32,64,128 σε ένα σύνολο δεδομένων 64000 εικόνων. Τα αποτελέσματα παρουσιάζονται στον Πίνακα 5.1 για το MK1 και στον 5.2 για το MK2.

| Batch_Size | Run_time (s) | Throughput | Time/image (ms) | Compilation_time (s) |
|---|---|---|---|---|
| 1 | 160.483 | 398.796 | 2.508 | 26.8 |
| 2 | 84.241 | 759.725 | 1.316 | 28.3 |
| 4 | 45.19 | 1416.243 | 0.706 | 34.6 |
| 8 | 30.209 | 2118.574 | 0.472 | 38.9 |
| 16 | 21.904 | 2921.841 | 0.342 | 49.1 |
| **32** | **18.696** | **3423.192** | **0.292** | 64.8 |

**Table 5.1.** *Inference BlazeFace on one MK1 IPU. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

| Batch_Size | Run_time (s) | Throughput | Time/image (ms) | Compilation_time (s) |
|---|---|---|---|---|
| 1 | 134.029 | 477.509 | 2.094 | 126.7 |
| 2 | 70.832 | 903.546 | 1.107 | 131.8 |
| 4 | 42.739 | 1497.461 | 0.668 | 139,2 |
| 8 | 25.541 | 2505.775 | 0.399 | 144.7 |
| 16 | 21.291 | 3005.965 | 0.333 | 156,1 |
| 32 | 19.379 | 3302.544 | 0.303 | 236,4 |
| **64** | **14.1394** | **4526.370** | **0.221** | 240,2 |
| 128 | 15.3025 | 4182.311 | 0.2391 | 292.6 |

**Table 5.2.** *Inference BlazeFace on one MK2 IPU. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

Με βάση τα παραπάνω, βλέπουμε οτι το batch size 32 ειναι η μέγιστη τιμή που μπορεί να χρησιμοποιηθεί για την μεταγλλώττιση και εκτέλεση του μοντέλου BlazeFace στο MK1 IPU τσιπ. Η απόδοση του μοντέλου (Throughput) είναι ίση με 3423 εικόνες ανα δευτερόλεπτο και ταχύτητα επεξεργασίας αν εικόνα 0,292 (ms). Η μεγαλύτερη διαθέσιμη μνήμη της δεύτερης γενιάς MK2 τσιπ σε σύγκριση με το MK1 (896 MiB έναντι 304 MiB) μας επέτρεψε να χρησιμοποιήσουμε μεγαλύτερα batch sizes για την μεταγλώττιση και εκτλέση του μοντέλου. Τα καλύτερα αποτελέσματα λήφθηκαν για batch-size 64 επιτυγχάνοντας επεξεργασία 4526 εικόνων ανα δευτερόλεπτο (throughput) και χρόνο επεξεργασίας ανα εικόνα 0,221 (ms). Επομένως, το batch size 32 για το MK1 και 64 για το MK2 είναι τα επιλεγμένα μεγέθη και οι περαιτέρω βελτιστοποιήσεις που θα αναλυθούν στη παρούσα εργασία αφορούν αυτές τις τιμές, εκτός αν αναφέρεται διαφορετικά.

#### Communication Overhead

Ο δεύτερος κρίσιμος παράγοντας που επηρεάζει σημαντικά την απόδοση είναι το κόστος επικοινωνίας (I/O) τόσο για τις μεταφορές δεδομένων μεταξύ του κεντρικού υπολογιστή (CPU) και της συσκευής (IPU), όσο και τις μεταφορες δεδομένων εντός της ίδιας της συσκευής. Οι μεταφορές δεδομένων μεταξύ του κεντρικού υπολογιστή (CPU) και της συσκευής (IPU) πραγματοποιούνται μέσω διεπαφών PCIe, οι οποίες έχουν μέγιστο αμφίδρομο εύρος ζώνης 64 GB/s. Αυτές οι μεταφορές είναι κοστοβόρες και η βελτιστοποίηση τους είναι σημαντική για την απόδοση.

Για την τροφοδοσία των δεδομένων στο μοντέλο μας έπρεπε αρχικά να δημιουργήσουμε ένα αντικέιμενο Dataset με την βοήθεια του tf.Data API. Το αντικείμενο tf.data.Dataset αποτελεί μια ακολουθία στοιχείων, όπου κάθε στοιχείο αναπαρίσταται ως ενας πολυδιάστατος τανυστής (Tensor). Για την μεταφορά των δεδομένων από και προς μια συσκευή IPU κατασκευάζονται δύο ουρές FIFO (FirstInFirstOut queues) που ονομάζονται IPUInfeedQueue και IPUOutfeedQueue. Οι ουρές αυτές δημιουργούνται αυτόματα οταν ένα μοντέλο κατασκευάζεται με τη βιβλιοθήκη Keras και καλείται το IPUStrategy API για την μεταγλλώτιση και εκτέλεση του μοντέλου.

Στην παρούσα εργασία εξερευνήθηκαν συνολικά τέσσερις ρυθμίσεις που προσφέρει η βιβλιοθήκη Poplar της Graphcore που αφορούν την μεταφορά και διαχείρηση δεδομένων.

Η πρώτη ρύθμιση ονομάζεται steps_per_execution και ορίζει τον αριθμό των παρτίδων (batches) που επεξεργάζονται διαδοχικά κατα την εκτέλεση του μοντέλου μας. Η προεπιλεγμένη τιμή είναι 1, πράγμα που σημαίνει ότι κατα την διάρκεια της εκτέλεσης το μοντέλου διαβάζει και επεξεργάζεται μόνο μία παρτίδα δεδομένων τη φορά και στη συνέχεια ζητάει απο τη CPU να τροφοδοτήσει την επόμενη παρτίδα απο τη κεντρική μνήμη. Θέτοντας το steps_per_execution σε υψηλοτερη τιμη σημαίνει πρακτικά οτι αλλάζουμε τον ρυθμό τροφοδοσίας και έτσι το IPU τσιπ λαμβάνει πολλαπλές παρτίδες δεδομένων ασύγχρονα. Αυτό μετριάζει το κόστος επικοινωνίας μεταξύ της CPU και IPU και βελτιώνει την συνολική απόδοση του συστήματος. Είναι σημαντικό να αναφέρουμε στο σημείο αυτό οτι ο συνολικός αριθμός των παρτίδων στο σύνολο δεδομένων επηρεάζει την τιμή του steps_per_execution, καθώς ο πρώτος πρέπει να διαιρείται απο τον δεύτερο. Η εξίσωση 5.1 περιγράφει τη μέγιστη τιμή που μπορεί να τεθεί στη ρύθμιση steps_per_execution:

$$steps_per_execution_{max} = (dataset_length//batch_size) \tag{5.1}$$

Δεδομένου οτι το dataset_length είναι 64000 και τα batch-sizes είναι 32 για το MK1 και 64 για το MK2 IPU τσιπ, προκύπτει οτι η μέγιστη τιμή που μπορεί να λάβει η ρύθμιση steps_per_execution είναι 2000 και 1000 αντίστοιχα. Θέτοντας τη μέγιστη δυνατή τιμή για την εκτέλεση του BlazeFace παρατηρήσαμε αύξηση της απόδοσης κατά **88%** για το MK1 και **55%** για το MK2, όπως φαίνεται στον πίνακα 5.3.

| Device | Batch_Size | Run_time (s) | Throughput | Time/image (ms) |
|---|---|---|---|---|
| MK1 | 32 | 8.746 | 7317.631 | 0.137 |
| MK2 | 64 | 7.99 | 8010.013 | 0.125 |

**Table 5.3.** *Inference results with steps_per_execution = num_of_samples / batch-size. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

Στη συνέχεια εξερευνήσαμε δύο ρυθμίσεις σχετικά με την μεταφορά και διαχείριση δεδομένων εντός του τσιπ IPU. Η πρώτη ρύθμιση μας επιτρέπει να χωρίσουμε τα πλακίδια (tiles) του IPU τσιπ σε δύο ομάδες. Η πρώτη ομάδα ονομάζεται I/O tiles και εκτελεί μόνο λειτουργίες I/O για την ανάκτηση και λήψη δεδομένων, ενώ η δεύτερη ομάδα ονομάζεται Compute tiles και είναι υπεύθυνη μόνο για την εκτέλεση υπολογισμών. Αντίστοιχα, το πρόγραμμα-IPU που φορτώνεται σε κάθε tile χωρίζεται σε δύο υποπρογράμματα που εκτελούνται παράλληλα, ένα για I/O και ένα για υπολογισμούς. Στο Σχήμα 5.3 παρουσιάζεται η ροή εκτέλεσης των δύο υποπρογραμμάτων.

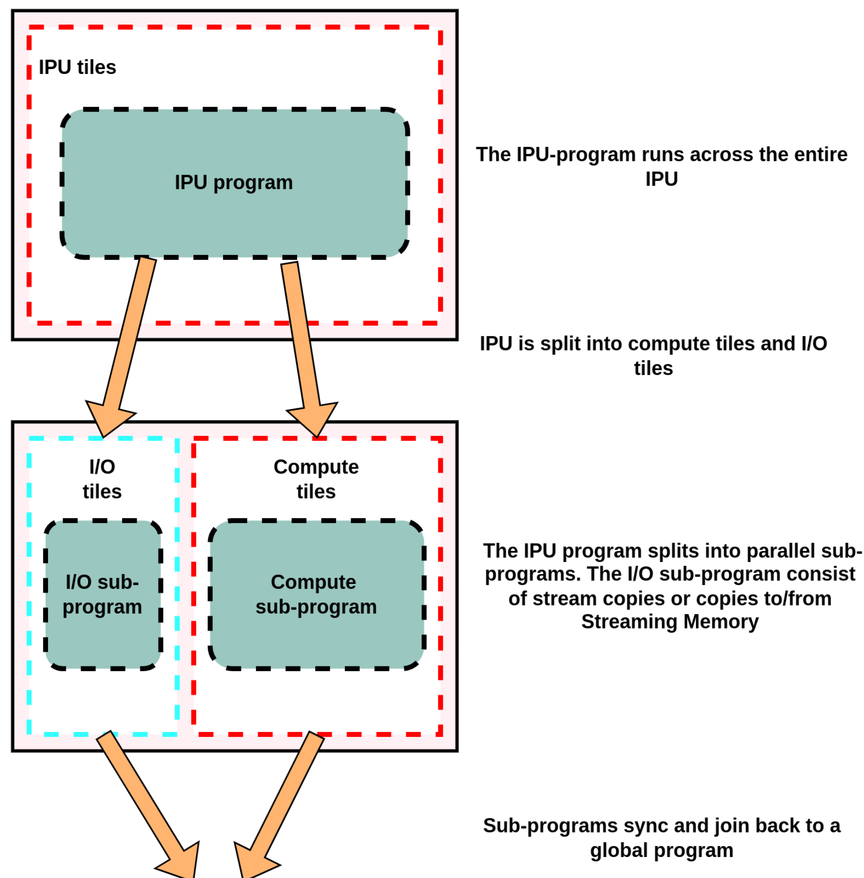



**Figure 5.3.** *Overlapping I/O within IPU*

Η τροφοδοσία ενός μεγάλου όγκου δεδομένων από τη CPU μπορεί να αυξήσει την καθυστερηση (latency) και να επηρεάσει σημαντικά τη συνολική απόδοση του συστήματος. Η χρήση I/O-tiles μπορεί να μειώσει το latency, καθώς οι λειτουργίες μεταφοράς δεδομένων και υπολογισμών εκτελούνται παράλληλα και ανεξάρτητα.

Σύμφωνα με τα έγγραφα οδηγιών (Documentation) της Graphcore [44], η επιλογή του αριθμού των I/O tiles προκύπτει εμπειρικα και μπορεί διαφέρει ανάλογα με τη φύση της εφαρμογης και τα χαρακτηριστικα του εκάστοτε μοντέλου προς εκτέλεση. Επιπλέον, αναφέρεται ρητά οτι ο αριθμός των I/O tiles πρέπει να επιλεχτεί προσεκτικά καθώς μπορεί να επηρεάσει σημαντικά την συνολική απόδοση του συστήματος, δεδομένου ότι τα tiles αυτά δεν μπορούν να συμμετέχουν σε υπολογισμούς.

Η δεύτερη ρύθμιση ονομάζεται Prefetch και επιτρέπει την προ-φόρτωση δεδομένων απο τη κεντρική μνήμη στο IPU τσιπ. Η προ-φόρτωση αυτή οδηγεί στο να υπάρχουν πάντα διαθέσιμα δεδομένα προς επεξεργασία πριν τα χρεαστεί το IPU και επομένως μειώνεται ο χρόνος αναμονής για αυτά. Ο αριθμός των δεδομένων που προ-φορτώνονται ελέγχεται από το όρισμα prefect_depth. Ομοίος με πριν, η τιμή του ορίσματος prefetch_depth προκύπτει εμπειρικά.

Τέλος, χρησιμοποιήσαμε μια τέταρτη ρύθμιση που ονομάζεται asynchronous callback. Η ενεργοποίηση αυτής της ρύθμισης ξυπνάει ένα επιπλέον νήμα (thread) το οποίο είναι υπεύθυνο για την μεταφορά των επεξεργασμένων δεδομένων εξόδου απο το IPU πίσω στη CPU. Η ρύθμιση αυτή είναι χρήσιμη σε περιπτώσεις που χρησιμοποιείται μια μεγάλη τιμή για τη ρύθμιση steps_per_execution καθώς η ασύγχρονη φόρτωση πολλαπλών παρτίδων δεδομένων εισόδου σημαίνει οτι θα παραχθεί αντίστοιχα και ένα μεγάλο σύνολο επεξεργασμένων δεδομένων εξόδου και η μεταφορά αυτών μπορεί να είναι αρκετά κοστοβόρα.

Στο πλαίσιο της εργασίας μας διεξήχθη ενα πείραμα αναζήτησης πλέγματος (grid search) με στόχο την εύρεση των βέλτιστων τιμών για τις ρυθμίσεις I/O tiles και prefetch_depth. Είναι σημαντικό να αναφερθεί ότι για το πείραμα αυτό χρησιμοποιήθηκαν κάποιες σταθερές τιμές για ορισμένες ρυθμίσεις. Πιο συγκεκριμένα, χρησιμοποιήσαμε τη μέγιστη δυνατή τιμή για τη ρύθμιση steps_per_execution, δηλαδή 1000 με batch-size 32 MK1 IPU και 2000 με batch-size 64 για το MK2 IPU. Επίσης ενεργοποιήσαμε και τη ρυθμιση asynchronous callback. Διερευνήσαμε τη χρήση 16, 32, 64 και 128 I/O σε συνδυασμό με την προφόρτωση ενός έως τριών παρτίδων δεδομένων.

Οι βέλτιστες ρυθμίσεις για το MK2 IPU τσιπ ήταν η χρήση 64 I/O tiles και προ-φόρτωση δύο παρτίδων δεδομένων. Η απόδοσή βελτιώθηκε κατά 36,5%, σε σύγκριση με τα αποτελέσματα του πίνακα 5.3, επιτυγχάνοντας Throughput 12614 (εικόνες/δευτερόλεπτο) και ταχύτητα επεξεργασίας 0.079 (ms) ανα εικόνα. Οι ίδιες ρυθμίσεις έδωσαν αντίστοιχα τα καλύτερα αποτελέσματα και για το MK1 IPU τσιπ. Η απόδοση βελτιώθηκε κατά 28,1%, σε σύγκριση με τα αποτελέσματα του Πίνακα 5.3, επιτυγχάνοντας Throughput 9374 (εικόνες/δευτερόλεπτο) και ταχύτητα επεξεργασίας 0.107 (ms) ανα εικόνα. Τα αποτελέσματα παρουσιάζονται στον Πίνακα. 5.4.

| Device | Batch_Size | Run_time (sec) | Throughput | Time/image (ms) |
|---|---|---|---|---|
| MK1 | 32 | 6,827 | 9374.475 | 0.107 |
| MK2 | 64 | 5.074 | 12614.051 | 0.079 |

**Table 5.4.** *Inference results with steps_per_execution = num_of_samples / batch-size, 64 dedicated I/O tiles and Prefetch enabled. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

#### On-Device Inference-Loop Implementation Approach

Η αρχική bulk-inference υλοποίηση είναι χρήσιμη σε περιπτώσεις όπου ένας μεγάλος όγκος δεδομένων βρίσκεται αποθηκευμένος στη κεντρική μνήμη. Ωστόσο, η εργασία αυτή στοχεύει στη δημιουργία μιας on-line εφαρμογής όπου η κάμερα καταγραφής θα τροφοδοτεί απευθείας ένα μικρό αριθμό δεδομένων (π.χ μια παρτίδα εικόνων) στον ανιχνευτή προσώπων και ακολούθως στον ανιχνευτή οροσήμων. Για τον λόγο αυτο, εξερευνήθηκε και αναπτύχθηκε μια δεύτερη υλοποίηση (την οποία στο εξής θα ονομάζουμε on-device inference loop) η οποία διαβάζει και επεξεργάζεται μια μόνο παρτίδα δεδομένων σε κάθε επανάληψη της εκτέλεσης του βρόχου εξαγωγής συμπερασμάτων (inference loop).

Για την προσέγγιση αυτή, χρησιμοποιήθηκε η μέθοδος tf.function(), που προσφέρει η βιβλιοθήκη Tensorflow, η οποία δέχεται μια συνάρτηση κώδικα Python ως είσοδο και κατασκευάζει έναν υπολογιστικό γράφο. Η μέθοδος αυτή συνδυάστηκε με τον μεταγλωττιστή XLA για την μεταγλώττιση ενός βελτιστοποιημένου γράφου και την προσέγγιση on-device inference loop για την αποδοτική εκτέλεση του σε ένα τσιπ IPU. Η προσέγγιση on-device inference loop εκτελεί επανειλημμένα το παραγώμενο μεταγλωττισμένο γράφημα σε μια μικρή παρτίδα δεδομένων εισόδου, παράγει τα δεδομένα εξόδου και, στη συνέχεια, ζητά την επόμενη παρτίδα δεδομένων εισόδου από τη CPU. Η διαδικασία αυτή συνεχίζεται μέχρι να υποβληθούν σε επεξεργασία όλα τα δεδομένα εισόδου.

Για την υλοποίηση αυτή χρησιμοποιήθηκαν batch size 32 και 64 για τους επιταχυντές υλικού MK1 και MK2 αντίστοιχα. Επίσης, η τιμή του steps_per_execution ορίστηκε σε 1 καθώς επεξεργαζόμαστε μία παρτίδα δεδομένων σε κάθε επανάληξη της εκτέλεσης. Τα αποτελέσματα παρουσιάζονται στον πίνακα 5.5.

| Device | Batch_Size | Run_time (sec) | Throughput | Time/image (ms) |
|---|---|---|---|---|
| MK1 | 32 | 7,052 | 9074.475 | 0.108 |
| MK2 | 64 | 5,558 | 11514.051 | 0.08 |

**Table 5.5.** *On-Device Inference loop results for MK1 and MK2 IPUs, Throughput (images/sec), Run_time is the total time of execution for processing 64000 images.*

### 5.2.2 Inference στη Tesla V100 GPU

Για την εκτέλεση inference στη GPU ακολουθήσαμε μια παρόμοια δαδικασία. Χρησιμοποιήθηκε ο μεταγλλωτιστής XLA για την κατασκευή ενός βέλτιστου υπολογιστικού γράφου ο οποίος εκτελείται μέσα σε μια tf.function(). Η υλοποίηση μας ακολουθεί την προσέγγιση

on-devide inference loop όπου το μοντέλο μας λαμβάνει ως είσοδο μια δέσμη δεδομένων σε κάθε επανάληψη της εκτέλεσης.

Η μεγαλύτερη συνολική διαθέσιμη μνήμη στο εσωτερικό της V100 μας επέτρεψε να χρησιμοποιήσουμε μεγαλύτερα μεγέθη παρτίδων κατά τη μεταγλώττιση και εκτέλεση του μοντέλο BlazeFace. Πραγματοποιήθηκαν πειράματα με μεγέθη παρτίδων έως και 512 σε ένα σύνολο δεδομένων 64000 εικόνων. Τα αποτελέσματα παρουσιάζονται στον Πίνακα 5.6.

| Batch_Size | Run_time (s) | Throughput | Time/image (ms) |
|---|---|---|---|
| 32 | 9.81 | 6521.81 | 0.1532 |
| 64 | 5.416 | 11816,838 | 0.0846 |
| 128 | 2.792 | 22968.123 | 0.0436 |
| 256 | 1.704 | 37705.264 | 0.0266 |
| 512 | 0.815 | 79123.324 | 0.0127 |

**Table 5.6.** *Bulk-Inference on Tesla V100 GPU. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

Σημειώνουμε εδώ ότι η GPU είναι ικανή να επεξεργάζεται μεγαλύτερα μεγέθη παρτίδων από τα IPU λόγω του ότι διαθέτει μεγαλύτερη συνολική μνήμη (32 GB έναντι 300 MiB για το MK1 και 900 Mib για το MK2). Η βέλτιστη απόδοση επιτυγχάνεται για το μέγιστο εξεταζόμενο μέγεθος παρτίδας 512.

Στην συνέχεια εκτελέσαμε πειράματα για μικρότερα μεγέθη παρτιδών (1 εώς 16) για να εξετάσουμε τη καθυστέρηση (latency) των αποκρίσεων της GPU. Τα αποτελέσματα παρουσιάζονται στον Πίνακα 5.7.

| | V100 | |
|---|---|---|
| Batch size | Latency | Time/image (ms) |
| 1 | **3.724** | 3.724 |
| 2 | 3.789 | 1,895 |
| 4 | 3.794 | 0,949 |
| 6 | 3.873 | 0,645 |
| 8 | 3.83 | 0,478 |
| 10 | 3.829 | 0.382 |
| 12 | 3.866 | 0,322 |
| 14 | 3.972 | 0,283 |
| 16 | 4.189 | 0,261 |

**Table 5.7.** *Low-latency approach results on V100 Tesla GPU, Latency (ms/batch).*

## 5.3 Ανίχνευση Προσώπων Post-processing

Το τρίτο στάδιο της υλοποίησης αφορά τη μετάφραση των ακατέργαστων δεδομένων εξόδων Tensors του BlazeFace σε πραγματικές συντεταγμένες bounding box που περιέχουν την περιοχή του προσώπου. Οι εντοπισμένες συντεταγμένες αντιστοιχούν στις εικόνες εισόδου 128x128 εικονοστοιχείων και επομένως έπρεπε να τις αντιστοιχίσουμε κατάλληλα στις

διαστάσεις της αρχικής εικόνας (640x480 εικονοστοιχεία). Αυτό το στάδιο εκτελείται απο τη CPU του συστήματος και επεξεργάζεται δεδομένα σε παρτίδες. Ο απαιτούμενος χρόνος εκτέλεσης υπολογίζεται περίπου στα 0,2 ms ανά εικόνα.

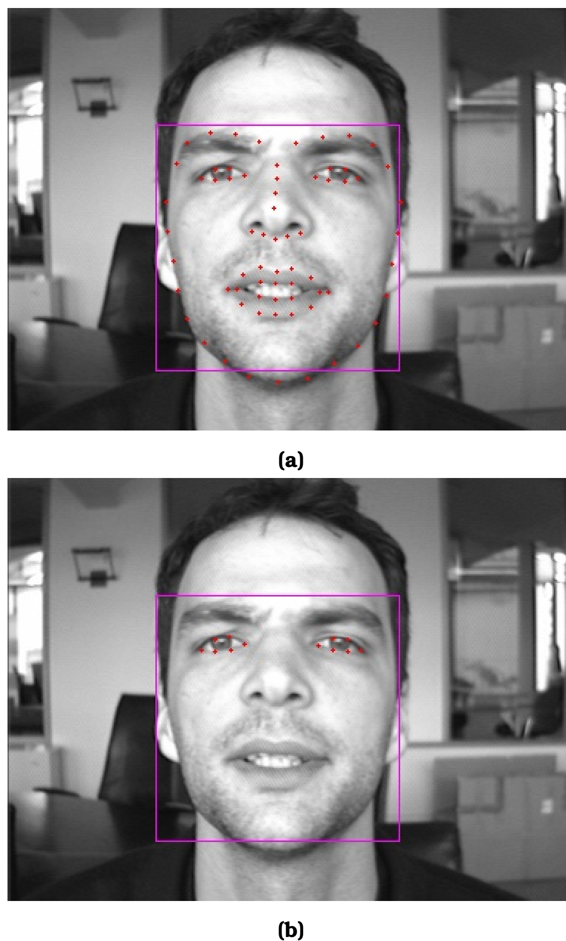
## 5.4 Ανίχνευση Οροσήμων

Το τέταρτο βήμα της υλοποίησης μας αφορά τον εντοπισμό οροσήμων (landmarks) στα πρόσωπα που ανιχνέυτηκαν στο προηγούμενο βήμα. Επομένως, ο ανιχνευτής οροσήμων λάμβάνει ως είδοδο τις συντεταγμένες των πλαισίων οριοθέτησης και εξάγει συνολικά 68 σημεία προσώπου σε κάθε πρόσωπο. Σε αυτό το βήμα, χρησιμοποιήθηκε μια προ-εκπαιδευμένη υλοποίηση του ανιχνευτή ERT που παρέχει η ανοιχτού κώδικα βιβλιοθήκη Dlib.

Ο στόχος της εργασίας είναι ο προσδιορισμός της μετρικής EAR, δηλαδή του πόσο ανοιχτά η κλειστά είναι τα μάτια, σε καθε δεδομένη χρονική στιγμή και επομένως μας ενδιαφέρει κυρίως ο εντοπισμός της θέσης των ματιών. Για τον λόγο αυτό, η υλοποίηση μας χρησιμοποιεί μόνο τις συντεταγμένες που αντιστοιχούν στη θέση των ματιών (6 σημεία σε κάθε μάτι - 12 ζέυγη συντεταγμένων) από το σύνολο των 68 εντοπισμένων σημείων.

Το γεγονός οτι ο προ-εκπαιδευμένος αλγόριθμος είναι υπεύθυνος για την πρόβλεψη 68 σημείων για κάθε εικόνα εισόδου, απο τα οποία χρειαζόμαστε μόνο ένα υποσύνολο, επηρεάζει την συνολική απόδοση του μοντέλου αλλά και το μέγεθος του, καθώς πρέπει να αποθηκεύει περισσότερες ποσοτικοποιημένες πληροφορίες σχετικά με τον τρόπο πρόβλεψης κάθε μιας από αυτές τις θέσεις. Για τους προαναφερθέντες λόγους, χρησιμοποιήθηκε και μια δεύτερη πειραματική υλοποίηση ενός landmark detector (τον οποίο στο εξής θα ονομάζουμε 12-landmark predictor) για τον εντοπισμό των 12 οροσήμων που περιγαφουν την θέση των ματιών. Στο Σχήμα 5.4 φαίνεται η ανίχνευση 68 σημείων που μας δίνει η προ-εκπαιδευμένη υλοποίηση της βιβλιοθήκης Dlib και μια ανίχνευση 12 σημείων απο τον 12-landmark detector.

Η αναλυτική περιγραφή του 12-landmark detector παρατίθενται στο αγγλικό τμήμα της εργασίας.



(a)

(b)

**Figure 5.4.** *a) Depicts a detection of Dlib's pre-trained 68-landmark detector. b) Depicts a detection from our custom 12-landmark detector. The image is taken from the BioID dataset [36]*

### 5.4.1 Αξιολόγηση ανιχνευτών στα συνολα δεδομένων IBUG-300W και BioID datasets

Για την αξιολόγηση των ανιχνευτών οροσήμων χρησιμοποιήσαμε αρχικά το σύνολο δεδομένων IBUG- 300W και τη μετρική Mean Average Error (MAE). Τα αποτελέσματα παρουσιάζονται στον πίνακα 5.8.

| Landmark Detectors | Size | Train-set error (%) | Test-set error (%) | Number of landmarks |
|---|---|---|---|---|
| Dlib 68-Landmark Detector | 99.7 MB | 6.9 % | 6.2 % | 68-points |
| 12-Landmark Detector | 25.6 MB | 2.1 % | 4.2 % | 12-points |

**Table 5.8.** *Mean Average Error (MAE) evaluation*

Για την περαιτέρω αξιολόγηση, ως προς την ακρίβεια και την ταχύτητα, των ανιχνευτών χρησιμοποιήσαμε και πάλι τη βάση δεδομένων BioID. Η μετρική αξιολόγησης που χρησιμοποιήσαμε ονομάζεται inter-ocular distance normalized error [42] και η συνολική διαδικασία που ακολουθήθηκε έχει αναλυθεί στο Κεφάλαιο 4. Τα αποτελέσματα της αξιολόγησης μας παρατίθονται στον Πίνακα 5.9

| Landmark Detector | Average error | Time/image (ms) |
| --- | --- | --- |
| Dlib's 68-Landmark Detector | 3.125 % | 1.06 |
| 12-Landmark Detector | 3.205 % | 1.02 |

**Table 5.9.** *Accuracy evaluation on the BioID database*

### 5.4.2 Παράλληλη εκτέλεση διεργασιών σε πολυπύρηνες CPU

Για την επιτάχυνση της διαδικασίας ανίχνευσης οροσήμων χρησιμοποιήσαμε τη βιβλιοθήκη multiprocessing, της γλώσσας προγραμματισμού Python, για την δημιουργία πολλαπλών διεργασιών και την παράλληλη εκτέλεση αυτών σε πολυπύρηνες CPU. Κάθε διεργασία φορτώνει και εκτελεί ένα αντίγραφο του ανιχνευτή. Ο αριθμός τον εικόνων που επεξεργάζεται καθε διεργασία είναι N/P, όπου N ορίζεται ως ο συνολικός αριθμός των εικόνων εισόδου και P ως ο αριθμός των διεργασιών που δημιουργούμε. Εκτελέσαμε πειράματα με 1, 2, 4, 8, 16, 32 και 64 διεργασίες σε ένα σύνολο δεδομένων 64000 φωτογραφιών. Τα αποτελεσματα παρουσιάζονται στον πίνακα 5.10.

| Processes | Time/image (ms) N=64000 |
| --- | --- |
| 1 | 1.059 |
| 2 | 0.563 |
| 4 | 0.316 |
| 8 | 0.157 |
| 16 | 0.089 |
| 32 | 0.071 |
| 64 | 0.060 |

**Table 5.10.** *Multi-process CPU approach for landmark detection on a sequence of 64000 frames*

## 5.5 Post-processing & Eye-blink Detection

Το τελικό στάδιο της υλοποίησης μας χρησιμοποιεί τα σημεία αναφοράς που εξάγει ο ανιχνυετής οροσήμων για να υπολογίσει τη μετρική Eye Aspect Ratio (EAR). Αυτό το στάδιο συνδυάζεται με το στάδιο ανίχνευσης ορόσημων και εκτελείται παράλληλα στη CPU με χρήση πολλαπλών διεργασιών.

Ένα παράδειγμα των ανιχνεύσεων ορόσημων σε ένα ανοιχτό και κλειστό μάτι, καθώς και η γραφική απεικόνιση των τιμών της μετρικής EAR στον άξονα του χρόνου, παρουσιάζονται στο Σχήμα 5.5.

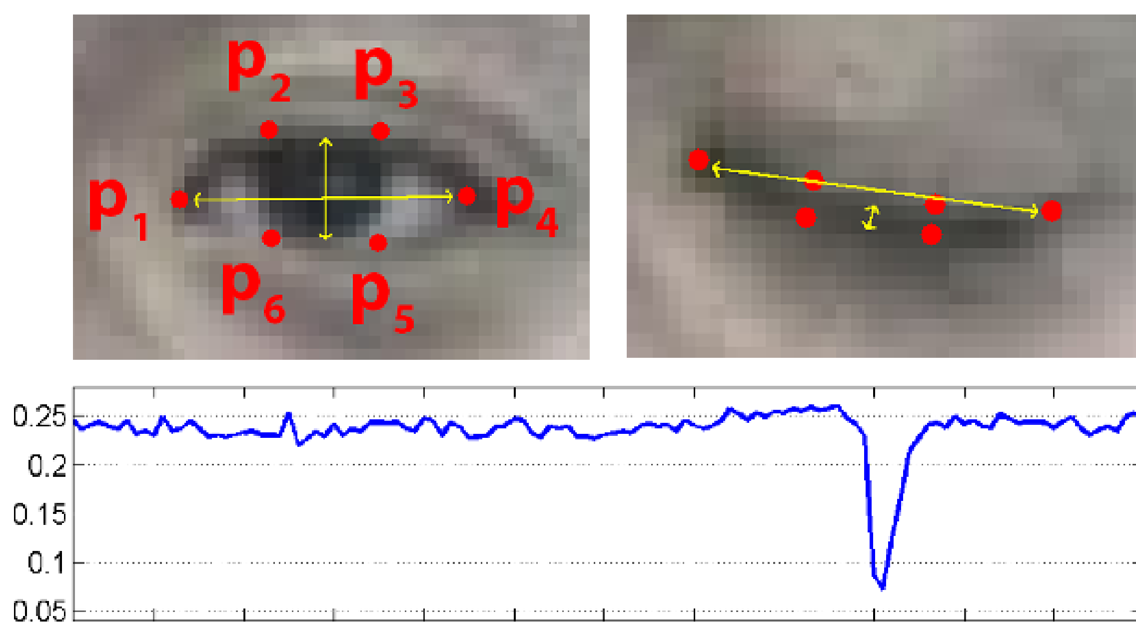



**Figure 5.5.** *The 6 facial landmarks associated with the eye and eyeblink response plot. Image from [38]*

Κεφάλαιο 6

# Αξιολόγηση

Σε αυτό το κεφάλαιο, αξιολογείται η ολοκληρωμένη υλοποίηση για την ανίχνευση του ποσοστού κλεισιμάτος των ματιών σε κάθε καρέ ενός βίντεο. Η πειραματική διάταξη που χρησιμοποιείται για τη παρούσα εργασία αναλύεται στην Ενότητα 6.1. Στην Ενότητα 6.2, παρουσιάζεται η ολοκληρωμένη υλοποίηση καθώς και τα επιταχυνόμενα αποτελέσματα των επιλεγμένων αλγορίθμων για την ανίχνευση προσώπου και οροσήμων. Επιπλέον, παρουσιάζουμε πώς κλιμακώνεται η απόδοση του αλγόριθμου με διαφορετικό υλικό CPU και IPU και επίσης αξιολογούμε την απόδοση κάθε πλατφόρμας υλικού που χρησιμοποιήθηκε όσον αφορά την καθυστέρηση (latency), την απόδοση (throughput) και την ενεργειακή απόδοση. Η Ενότητα 6.3 αναλύει τη δυνατότητα επεκτασιμότητας της εκτέλεσης του βήματος ανίχνευσης προσώπου σε πολλαπλά τσιπ IPU.

## 6.1 Περιβάλλον Διεξαγωγής Πειραμάτων

Οι προδιαγραφές του επιταχυντή GPU Tesla V100 που χρησιμοποιήθηκε αναλύονται στην Ενότητα 5.2 του αγγλικού κειμένου. Η CPU είναι μια AMD EPYC 7551, της οποίας οι προδιαγραφές συνοψίζονται στη δεύτερη στήλη του Πίνακα 6.1.

| **Specification** | **V100 Host** |
|---|---|
| Clock Speed | 2.0 GHz |
| Number of cores | 32 (64 threads) |
| PCIe controller | PCIe 3.0 (128 lines) |
| L1 cache | 32 x 64 KiB |
| L2 cache | 32 x 512 KiB |
| L3 cache | 64 MB |

**Table 6.1.** *Specifications of GPU Host AMD EPYC 7551 CPU*

Για την αξιολόγηση στις πλατφόρμες υλικού IPU, χρησιμοποιήσαμε ένα σύστημα IPU-POD 16 και ένα σύστημα διακομιστή IPU-server. Το πρώτο έχει τέσσερα rack IPU-M2000s με 16 τσιπ MK2-IPU συνολικά που τρέχουν σε κεντρικό διακομιστή, ενώ το δεύτερο είναι ένα σύστημα με 8 κάρτες C2 PCIe το οποίο συνολικά διαθέτει 16 MK1-IPU τσιπ. Τα βασικά χαρακτηριστικά των τσιπ IPU MK1 και MK2 αναλύονται στην Ενότητα 5.2 του αγγλικού κειμένου. Η CPU του συστήματος IPU-server είναι μια Intel Xeon Platinum 8168, ενώ η

CPU του συστήματος IPU-POD 16 είναι μια CPU AMD-EPYC 7742. Οι προδιαγραφές και για τις δύο CPU συνοψίζονται στη δεύτερη και τρίτη στήλη του Πίνακα 6.2 αντίστοιχα.

| Specification | MK1 Host | MK2 Host |
|---|---|---|
| Clock Speed | 2.70 GHz | 2.25 GHz |
| Number of cores | 24 (48 threads) | 64 (128 threads) |
| PCIe controller | PCIe 3.0 (48 lines) | PCIe 4.0 (128 lines) |
| L1 cache | 24 x 32 KB | 64 x 32 KB |
| L2 cache | 24 x 1024 KB | 64 x 512 KB |
| L3 cache | 33 MB | 256 MB |

**Table 6.2.** *Χαρακτηριστικά των MK1 Host Intel Xeon Platinum 8168 CPU και MK2 host, AMD-EPYC 7742 CPU*

## 6.2 Eye-Blink Response Detection

Σε αυτή την ενότητα θα συζητήσουμε τα αποτελέσματα της επιτάχυνσης των επιμέρους βημάτων της υλοποίησης μας (Data Preparation, BlazeFace, Ensemble of Regression Trees (ERT)) καθώς και την ολοκληρωμένη υλοποίηση της εφαρμογής για την ανίχνευση της μετρικής EAR.

### 6.2.1 Φόρτωση εικόνων και προ-επεξεργασία

Τα αρχικά βήματα φόρτωσης και προεπεξεργασίας των εικόνων εισόδου εκτελέστηκαν από πολλαπλές διεργασίες (processes) σε πολυπήρηνες CPU. Ο αριθμός των διεργασιών ορίστηκε να είναι ίσος με τον αριθμό των φυσικών πυρήνων της CPU του εκάστοτε συστήματος (Πίνακας 6.1 και Πίνακας 6.2). Τα αποτελέσματα επιτάχυνσης παρουσιάζονται στον Πίνακα 6.3.

| System | # Processes | time per image (ms) | Speedup |
|---|---:|---:|---:|
| Naive version | 1 | 2.6 | - |
| AMD EPYC 7551 (GPU) | 32 | 0.381 | 6,8 |
| Intel Xeon Platinum 8168 (MK1) | 24 | 0.438 | 5,9 |
| MD-EPYC 7742 (MK2) | 64 | 0.262 | 9,9 |

**Table 6.3.** *Execution time of the combined image loading and pre-processing step for the host CPUs of our systems.*

Από τα παραπάνω αποτελέσματα παρατηρούμε ότι καταφέραμε να μειώσουμε σημαντικά τον απαιτούμενο χρόνο εκτέλεσης των δύο αυτών βημάτων επιτυγχάνοντας ενα speedup μεταξύ 6 και 10 μονάδων.

### 6.2.2 Ανίχνευση Προσώπου σε πλατφόρμες υλικού IPU & GPU

Για την ανίχνευση προσώπου χρησιμοποιήθηκε το μοντέλο Blazeface. Για την ανάπτυξη της αρχιτεκτονικής του νευρωνικού δικτύου χρησιμοποιήθηκαν τα εργαλεία TensorFlow

v2.4.4 και η ενσωματωμένη βιβλιοθήκη Keras. Επίσης, έγινε χρήση του μεταγλωτιστή XLA για την εξαγωγή και εκτέλεση ενός βελτιστοποιημένου υπολογιστικού γράφου στις πλατφόρμες υλικού IPU και GPU. Η αξιολόγηση βασίζεται στα αποτελέσματα των παρακάτω υλοποιήσεων:

- Εκτέλεση της αρχικής υλοποίησης του μοντέλου BlazeFace στην CPU (βλ. Ενότητα 4.5).
- Εκτέλεση του βελτιστοποιημένου υπολογιστικού γράφου με χρήση XLA στους επιταχυντλες MK1 και MK2 IPU (βλ. Ενότητα 5.2).
- Εκτέλεση του βελτιστοποιημένου υπολογιστικού γράφου με χρήση XLA στον επιταχυντή Tesla V100 GPU (βλ. Ενότητα 5.2).

Τα αποτελέσματα παρουσιάζονται στον πίνακα 6.4.

| Implementaion | Time/Image (ms) | Speedup |
|---|---:|---:|
| BlazeFace serial version | 32.727 | - |
| MK1 BlazeFace implementation | 0.107 | 306 |
| MK2 BlazeFace implementation | 0.079 | 414 |
| Tesla V100 BlazeFace implementation | 0.0127 | 2576 |

**Table 6.4.** *Performance comparison of different implementations of the BlazeFace model. Speedup is calculated relative to the slowest implementation.*

Απο τα αποτελέσματα του παραπάνω πίνακα βλέπουμε οτι επιτυγχάνουμε ένα speedup της ταξής του 305× και 414× εκτελώντας το μοντέλο BlazeFace στους επιταχυντές MK1-IPU και MK2-IPU αντίστοιχα. Η αρκετά μεγαλύτερη διαθέσιμη μνήμη της Tesla V100 GPU (32GB) μας επετρεψε να τρέξουμε το μοντέλο με μεγαλύτερα batch-sizes (εως και 512). Αυτό οδήγησε στην επίτευξή ενός κατα πολύ μεγαλύτερου speedup της ταξεως του 2576×, σε σχέση με την αρχική σειριακή εκτέλεση του μοντέλου στη CPU. Η περιορισμένη μνήμη των MK1-IPU (312 Mib) και MK2-IPU (918 Mib) τσιπ μας περιορίζει να χρησιμοποιήσουμε αρκετά μικρότερα μεγέθη παρτίδων εικόνων (32 και 64 αντιστοιχα) καθώς για μεγαλύτερες τιμές ήταν αδύνατο να μεταγλλωτιστεί το μοντέλο μας και να παραχθεί ο απαραίτητος στατικός υπολογιστικός γράφος για εκτέλεση στα IPU τσιπ.

### 6.2.3 Ανίχνευση οροσήμων σε πολυπύρηνες CPU

Για την εκτέλεση του ανιχνευτή Ensemble of Regression Trees (ERT) ακολουθήθηκε μια data-parallel προσέγγιση με τη βοήθεια της βιβλιοθήκης multiprocessing της γλώσσας προγραμματισμού Python κάνοντας χρήση πολλαπλών διεργασιών (processes) σε πολυπύρηνες CPU (βλ. Ενότητα 5.4.2). Στον Πίνακα 6.5 παρουσιάζονται τα αποτελέσματα της αρχικής σειριακής υλοποίησης καθώς και τα αποτελέσματα της επιταχυνόμενης υλοποιήσης με χρήση πολλαπλών διεργασιών.

| System | # Processes | Time/Image (ms) | Speedup |
|---|---|---|---|
| Original serial version | 1 | 1.06 | - |
| AMD EPYC 7551 (GPU) | 32 | 0.071 | 14,9 |
| Intel Xeon Platinum 8168 (MK1) | 24 | 0.078 | 13,6 |
| MD-EPYC 7742 (MK2) | 64 | 0.06 | 17,7 |

**Table 6.5.** *Performance comparison of original sequential landmark-detection algorithm with the multiprocessing-accelerated version. Speedup is calculated relative to the slowest implementation.*

### 6.2.4 Αποτελέσματα ολοκληρωμένης υλοποίησης για Eye-Blink Response Detection

Η συνολική διαδικασία ανίχνευσης μπορεί να χωριστεί σε τρία χρονοβόρα βήματα:

1. **Φόρτωση, Αποκωδικοποίηση και προεπεξεργασία εικόνας**: Οι εικόνες είναι αποθηκευμένες σε μορφή JPEG σε έναν σκληρό δίσκο SSD (Solid State Drive). Κάθε εικόνα ανακτάται, αποκωδικοποιείται και μετατρέπεται σε μορφή πίνακα των 128x128x3 εικονοστοιχείων. Κάνουμε χρήση πολλαπλών διεργασιών για την παράλληλη επεξεργασία κάθε παρτίδας δεδομένων εισόδου. Η ταχύτητα επεξεργασίας υπολογίζεται περίπου ως 0.4 (ms/εικόνα) για τη Intel Xeon Platinum 8168 CPU (MK1 Host), 0.3 (ms/εικόνα) για την CPU AMD-EPYC 7742 CPU (MK2 Host) και 0.4 (ms/εικόνα) για τη AMD EPYC 7551 (GPU Host).

2. **Face Detection σε IPU και GPU**: Η CPU τροφοδοτεί τα προεπεξεργασμένα δεδομένα στον επιταχυντή, ο οποίος επεξεργάζεται παρτίδες εικόνων και επιστρέφει τα εντοπισμένα πλαίσια οριοθέτησης (bounding boxes). Η ταχύτητα επεξεργασίας που επιτυγχάνει το MK2 IPU τσιπ είναι 0.079 (ms/εικόνα) με batch size 64 ενώ το MK1 IPU τσιπ 0.107 (ms/εικόνα) με batch size 32. Τέλος, ο χρόνος εκτέλεσης στην GPU V100 Tesla με batch size 512 είναι 0.013 (ms/εικόνα).

3. **Landmark Detection**: Το βήμα της ανίχνευσης ορόσημων δέχεται τα εντοπισμένα πλαίσια εκτελείται επίσης παράλληλα από πολλαπλές διεργασίες σε πολυπύρηνες CPU και ο χρόνος εκτέλεσης κυμαίνεται μεταξύ 0,06 - 0,08 (ms/εικόνα).

Τα παραπάνω βήματα εκτελούνται διαδοχικά σε ένα σύνολο 64000 εικόνων. Ο αριθμός των διεργασιών είναι ίσος με τον μέγιστο αριθμό των διαθέσιμων πυρήνων κάθε CPU. Τα αποτελέσματα της εκτέλεσης όλων των πιθανών συνδυασμών των μοντέλων μας και των πλατφορμών υλικού παρουσιάζονται στον πίνακα 6.6.

| Hardware | Landmark model | Time per image (ms) |
|---|---|---|
| MK2 IPU | 68-point | 0.712 |
| MK2 IPU | 12-point | 0.694 |
| MK1 IPU | 68-point | 0.761 |
| MK1 IPU | 12-point | 0.731 |
| Tesla V100 | 68-point | 0.642 |
| Tesla V100 | 12-point | 0.603 |

**Table 6.6.** *Αποτελέσματα των συνδυασμένων εφαρμογών ανίχνευσης του βλεφαρίσματος.*

Η επιτευμένη ταχύτητα επεξεργασίας της ολοκληρωμένης υλοποιήσης στις διάφορες πλατφορμες υλικού υπολογίζεται περίπου στα 1441 FPS στην MK2-IPU, 1367 FPS στην MK1-IPU και 1658 FPS στην Tesla V100 GPU. Όλες οι παραπάνω ταχύτητες ικανοποιούν την αρχική απαίτηση των 500 FPS.

### 6.2.5 Επίδοση υλικού για Low-Latency αποκρίσεις

Η ολοκληρωμένη υλοποίηση αξιολογήθηκε περαιτέρω και για τις τρεις πλατφόρμες υλικού εκτελώντας πειράματα με μικρότερα μεγέθη παρτίδων. Οι μετρικές αξιολόγησης είναι η καθυστέρηση, η απόδοση και η ενεργειακή κατανάλωση.

#### Καθυστέρηση (Latency)

Η καθυστέρηση μετριέται σε χιλιοστά του δευτερολέπτου ανά παρτίδα. Αντιπροσωπεύει το συνολικό χρόνο που απαιτείται για τη λήψη των αποτελεσμάτων από ένα batch. Η καθυστέρηση περιλαμβάνει την εξαγωγή συμπερασμάτων και την ανάκτηση της εξόδου από τη συσκευή.

#### Απόδοση (Throughput)

Η Απόδοση (Throughput) μετράται σε εικόνες ανά δευτερόλεπτο και προκύπτει από τη μέτρηση του χρόνου καθυστέρησης και το batch size. Αυτή η μετρική αντιπροσωπεύει το φορτίο που μπορεί να διαχειριστεί το υλικό για μια εφαρμογή που χρησιμοποιεί μοντέλα βαθιάς μάθησης.

Στα Σχήματα 6.1 και 6.2 παρουσιάζονται τα αποτελέσματα σχετικά με την καθυστέρηση (latency) και την απόδοση (throughput) που παρουσίασαν οι επιταχυντές. Επιλέξαμε να χρησιμοποιήσουμε μικρά μεγέθη batch-size (1, 2, 4, 6, 8, 10, 12, 14 και 16) για τα πειράματα μας καθώς συχνά επιλέγονται για την ανάπτυξη real-time εφαρμογών [45]. Τα αποτελέσματα προέκυψαν απο την εκτέλεση 1000 επαναλήψεων ανεξαρτητα απο το μέγεθος παρτίδας.

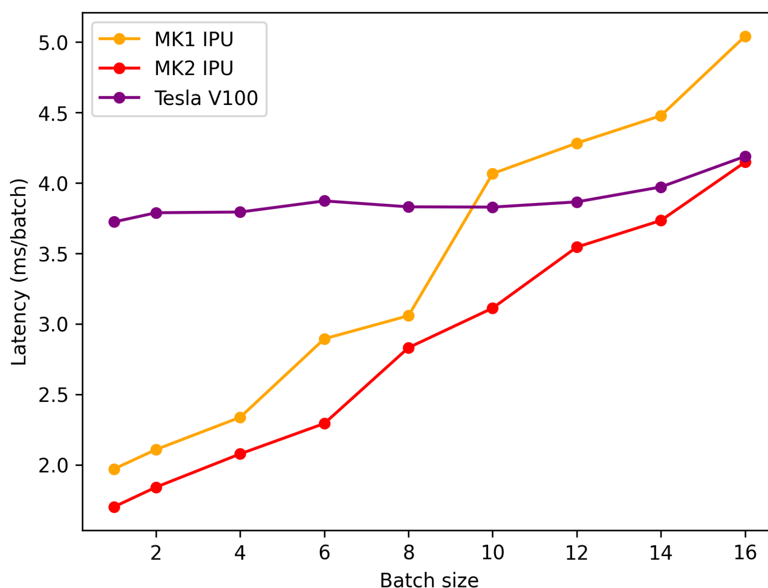



**Figure 6.1.** *Latency results comparison for MK1, MK2 and V100*

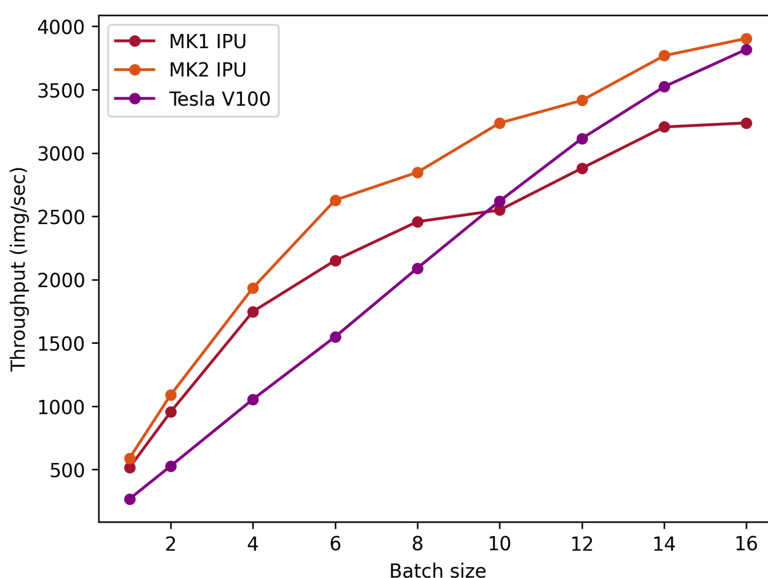



**Figure 6.2.** *Throughput results comparison for MK1, MK2 and V100*

Από τα παραπάνω διαγράμματα παρατηρούμε ότι τόσο τα τσιπ MK1 όσο και τα τσιπ MK2 IPU έχουν καλύτερες επιδόσεις όσον αφορά την καθυστέρηση και την απόδοση για μικρότερα batch sizes (1 έως 8) σε σύγκριση με την GPU Tesla V100, όπου δεν παρατηρείται σημαντική αύξηση της καθυστέρησης όσο αυξάνεται το batch size. Ωστόσο, για μεγαλύτερα μεγέθη παρτίδων (8 εως 16), η GPU V100 κατάφερε να ξεπεράσει το chip MK1 IPU, ενώ το chip MK2 IPU πέτυχε την υψηλότερη απόδοση (throughput) καθώς και τη χαμηλότερη καθυστέρηση.

#### Κατανάλωση Ενέργειας

Η ενεργειακή κατανάλωση είναι ένας σημαντικός παράγοντας για την αξιολόγηση μιας πλατφορμας υλικού. Για την παρακολούθηση της κατανάλωσης ενέργειας μιας μονάδας IPU η Graphcore προσφέρει το εργαλείο gc-monitor. Το αντίστοιχο εργαλείο για την παρακολούθηση μιας GPU ονομάζεται NNVIDIA SMI.

#### Ενεργειακή Απόδοση

Η ενεργειακή απόδοση μετράται σε εικόνες ανά δευτερόλεπτο ανά Watt. Αντιπροσωπεύει την ενεργειακή αποτελεσματικότητα μιας πλατφόρμας υλικού. Η κατανάλωση ενέργειας μετράται σε διαφορετικές χρονικές στιγμές κατά την διάρκεια του inference και υπολογίζεται ο μέσος όρος. Η ενεργειακή προκύπτει από την απόδοση (throughput) κατα την διάρκεια εκτέλεσης διαιρούμενη με τη μέση κατανάλωση.

| Batch size | Energy Efficiency | | |
|---|---|---|---|
| | **MK1** | **MK2** | **V100** |
| 1 | 4,74 | 8,02 | 4,62 |
| 2 | 8,45 | 14,86 | 13,2 |
| 4 | 15.61 | 24,97 | 21,97 |
| 6 | 18,85 | 29,48 | 28,16 |
| 8 | 20,96 | 37,07 | 36,68 |
| 10 | 22,06 | 42,51 | 45,18 |
| 12 | 25,01 | 44,66 | 45,80 |
| 14 | 27,27 | 46,53 | 48,94 |
| 16 | **27,58** | **51,05** | **48,96** |

**Table 6.7.** *Energy efficiency (images/sec/watt) for IPUs and GPU*

Από τον παραπάνω πίνακα παρατηρούμε ότι και οι τρεις πλατφόρμες υλικού γίνονται πιο αποδοτικές όσο αυξάνεται το μέγεθος του batch-size. Βλεπουμε επίσης οτι η ενεργειακή απόδοση της μονάδας MK2 IPU είναι συγκρίσιμη με αυτή της V100 GPU. Απο την άλλη το πρώτης γενιάς MK1 IPU έχει ασισθητά χαμηλότερη ενεργειακη απόδοση.

## 6.3 Hardware Scalability

### 6.3.1 Εκτέλεση του μοντέλου BlazeFace σε πολλαπλές μονάδες IPU

Το μοντέλο ανίχνευσης προσώπου BlazeFace αποτελεί ένα αρκετά γρήγορο και μικρό μοντέλο όσον αφορά το αποτύπωμα της απαιτούμενης μνήμης, με λίγο περισσότερες από εκατό χιλιάδες παραμέτρους. Το μοντέλο BlazeFace μπορεί να μεταγλλωτιστεί και να εκτελεστεί απο ένα IPU τσιπ και επομένως μπορεί να εφαρμοστεί μια data-parallel προσέγγιση για την εκτέλεση του μοντέλου μας σε πολλαπλές IPU. Ο χρήστης πρέπει να ρυθμίσει αρχικά τον παράγοντα αντιγραφής (replication factor) για να ορίσει το πλήθος των διαφορετικών μονάδων IPU που θα χρησιμοποιηθούν. Στην συνεχεια ο πηγαίος κώδικα μεταγλωτίζεται

για να παραχθεί ο υπολογιστικός γράφος (πρόγραμμα-IPU) και φορτώνεται στις πολλαπλές μονάδες IPU.

Για να πραγματοποιηθεί αυτό χρησιμοποιήθηκε μια λειτουργία εκκίνησης Multi-instance/Single host κατά την οποία εκκινούνται πολλαπλά αντίγραφα του κώδικα μας στον ίδιο κεντρικό διακομιστή με την χρήση της mpirun. Κάθε instance λαμβάνει και επεξεργάζεται ενα διαφορετικό υποσυνόλου δεδομένων και διαθέτει εναν ξεχωριστό μεταγλωττιστή (Graph Compile Domain - GCD). Επομένως, κάθε instance/replica του BlazeFace στην εκάστοτε μονάδα IPU λειτουργεί αυτόνομα και τα επεξεργασμένα δεδομένου εξόδου αποστέλλονται πίσω στον κεντρικό υπολογιστή (CPU) ξεχωριστα (Σχήμα 6.3).

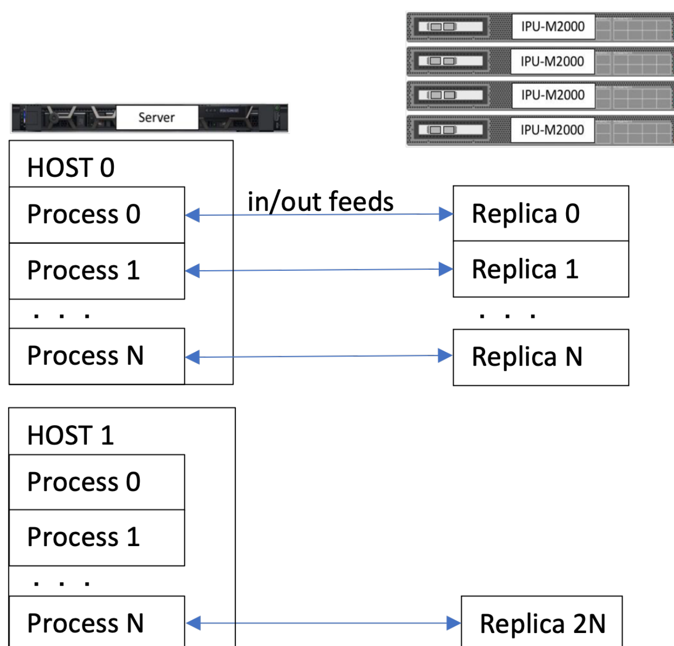
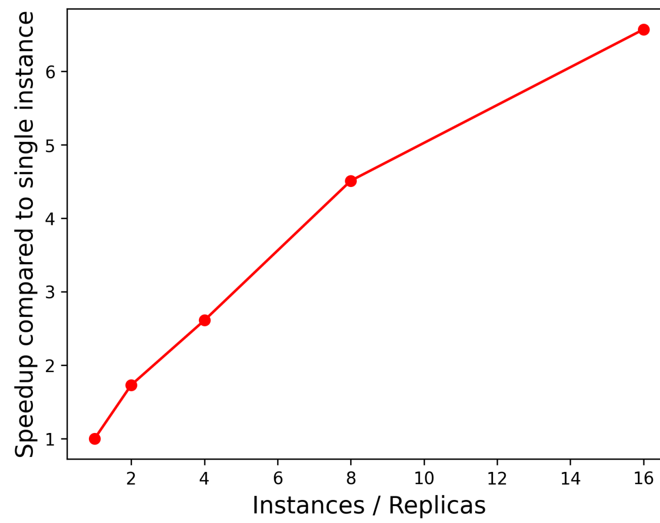
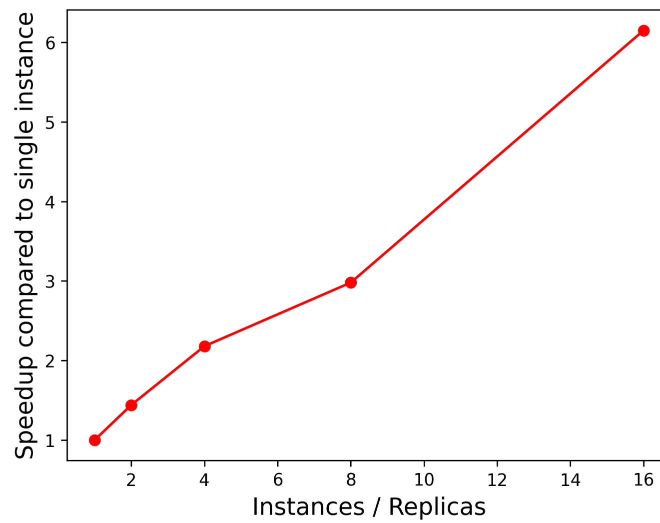



**Figure 6.3.** *Multi-instance replication*

Αξιολογούμε την κλιμακωσιμότητα της απόδοσης του μοντέλου BlazeFace με πολλαπλά instances στο IPU-POD16 cluster με 16 τσιπ IPU MK2 και στο IPU-Server cluster με 16 τσιπ IPU MK1. Το test-set μας περιέχει 64000 εικόνες και κάθε αντίγραφο του μοντέλου μας θα επεξεργαστεί ένα υποσύνολο εικόνων ίσο με 64000 / number_of_instances. Τα αποτελέσματα φαίνονται Στο Σχήμα (6.4 φαίνεται η κλιμάκωση της απόδοσης σε σχέση με τον αριθμό των μονάδων IPU.




**(a)**




**(b)**

**Figure 6.4.** *Multi-instance Speedup for a) MK1 IPU-Server with 16 MK1 IPU chips and b) MK2 IPU-POD16 with 16 MK2 IPU chips*

Με βάση τα παραπάνω, η απόδοση φαίνεται να κλιμακώνεται σχεδόν γραμμικά σε σχέση με τον αριθμό των μονάδων IPU.

### 6.3.2 Ελάχιστοι απαιτούμενοι πόροι υλικού που πληρούν τις προδιαγραφές

Η πλήρης υλοποίηση αποτελείται από πέντε στάδια: φόρτωση εικόνας/προεπεξεργασία (ILP), ανίχνευση προσώπου (FP), ανίχνευση προσώπου-μετεπεξεργασία (FDPP), ανίχνευση ορόσημου (LD) και υπολογισμός EAR (EAR). Το στάδιο της ανίχνευσης προσώπου εκτελείται σε μια πλατφόρμα υλικού IPU ή GPU, ενώ τα υπόλοιπα στην κεντρική CPU. Τα βήματα ILP και LP υπολογίζονται με τη χρήση πολλαπλών διεργασιών. Ως εκ τούτου, ο συνολικός χρόνος ανίχνευσης της απόκρισης του βλεφαρίσματος μπορεί να οριστεί ως εξής:

$$time = ILP + FD + FDPP + LD + EAR \tag{6.1}$$

Μια ταχύτητα ανίχνευσης 500 FPS ισοδυναμεί με μέγιστο χρόνο ανίχνευσης 2 ms. Ως εκ τούτου, πραγματοποιήσαμε διάφορα πειράματα και στις τρεις πλατφόρμες υλικού για να βρούμε το ελάχιστο απαιτούμενο batch-size και τον αριθμό διεργασιών που ικανοποιούν την απαίτηση χρόνου εκτέλεσης < 2 ms. Τα αποτελέσματα παρουσιάζονται στον Πίνακα 6.8.

| Device | Host | Processes | batch_Size | time/image (ms) |
|---|---|---|---|---|
| MK1 | Intel Xeon P. 8168 | 8 | 10 | 1.744 |
| MK2 | AMD EPYC 7742 | 4 | 8 | 1.536 |
| V100 | AMD EPYC 7551 | 6 | 10 | 1.698 |

**Table 6.8.** *Ελάχιστοι απαιτούμενοι πόροι υλικού για εκτέλεση σε χρόνο <2ms)*

Κεφάλαιο 7

# Analyzing Performance of IPU and GPU Platforms for CNN-Based Model Training

Στο Κεφάλαιο 5 διερευνήθηκε η επιτάχυνση του ανιχνευτή προσώπων BlazeFace σε δύο διαφορετικές πλατφόρμες υλικού (IPU, GPU). Για την διαδικασία εξαγωγής συμπερασμάτων (inference) επιλέχθηκε να χρησιμοποιηθεί μια προ-εκπαιδευμένη υλοποίηση καθώς η εκπαίδευση ενός robust και αποτελεσματικού ανιχνευτή είναι μια πολύ χρονοβόρα και απαιτητική [23] [46]. Σε αυτό το κεφάλαιο θα διερευνήσαμε και και θα συγκρίνουμε τις δυνατότητες των διαθέσιμων επιταχυντών υλικού (IPU, GPU) στην διαδικασία εκπαίδευσης ενός μοντέλου CNN βασισμένου σε εικόνες.

Για τον λόγο αυτό κατασκευάσαμε ένα πειραμάτικό end-to-end training pipeline του μοντέλου BlazeFace με βάση την πρωτότυπη εργασα [10] και εκτελέσαμε πειράματα για να συγκρίνουμε τις δυνατότητες των διαθέσιμων επιταχυντών υλικού (IPU, GPU) στην διαδικασία εκπαίδευσης ενός μοντέλου CNN.

## 7.1 Υλοποίηση

### 7.1.1 Δεδομένα

Για την εκπαίδευση του μοντέλου BlazeFace επιλέχθηκαν δύο σύνολα δεδομένων. Αρχικά χρησιμοποιήσαμε το σύνολο δεδομένων FDDB [47] το οποίο αποτελείται από 2845 εικόνες και περιγράφει 5171 προσώπα. Το δεύτερο σύνολο δεδομένων ονομάζεται 300W-LP [48] όπου ο συνολικός αριθμός δειγμάτων ανέρχεται στα 61255.

### 7.1.2 Λογισμικό

Για την κατασκευή της αρχιτεκτονικής του μοντέλου και την εκπαίδευση σε πλατφόρμες υλικού IPU και GPU χρησιμοποιήσαμε τις βιβλιοθήκες Tensorflow και Keras. Πιο συγκεκριμένα για την εκπαίδευση του μοντέλου σε τσιπ IPU χρησιμοποιήσαμε το Poplar SDK v2.4 και η βιβλιοθήκη TensorFlow v2.4.4. Η ίδια έκδοση της βιβλιοθήλης TensorFlow χρησιμοποιήθηκε και για τα πειράματα στην V100 Tesla GPU.

### 7.1.3 Αξιολόγηση αποτελεσμάτων

Η εκπαίδευση του μοντέλου πραγματοποιήθηκε για 150 εποχές στο σύνολο δεδομένων FDDB και για 100 εποχές στο σύνολο δεδομένων 300W-LP. Είναι σημαντικό να αναφερθεί οτι χρησιμοποιήθηκε μια κάρτα C2 PCIe, με δύο συνδεδεμένες MK1 IPU, για την εκπαίδευση του μοντέλου μας καθώς η τιμή του TDP (250 W) συμβαδίζει με αυτήν των επιταχυντών MK2 IPU και Tesla V100 GPU. Τα αποτελέσματα παρουσιάζονται στο Σχήμα 7.1 και στο Σχήμα 7.2 για τα σύνολα δεδομένων FDDB και 300W-LP αντίστοιχα.

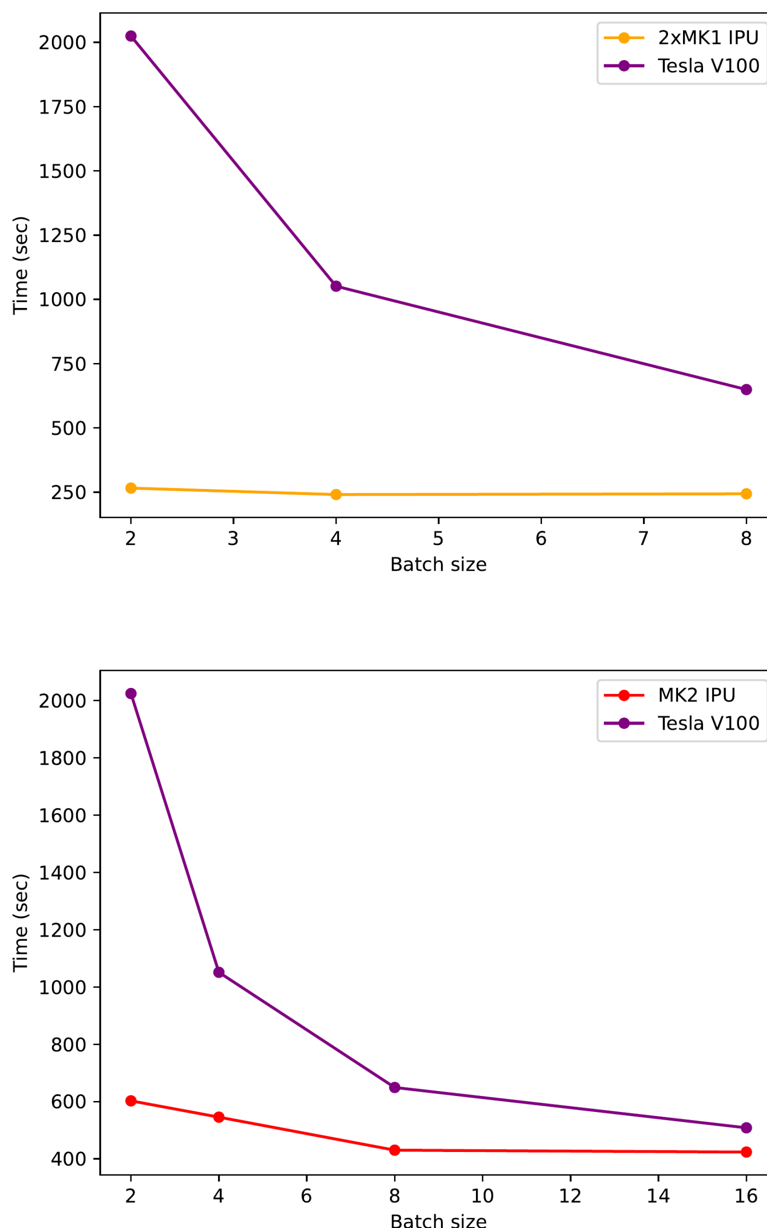



**Figure 7.1.** *Training time of BlazeFace model on the FDDB dataset. (top) The results for various batch sizes on V100 GPU and MK1 IPU chips. (bottom) The results for various batch sizes on V100 GPU and MK2 IPU chips.*

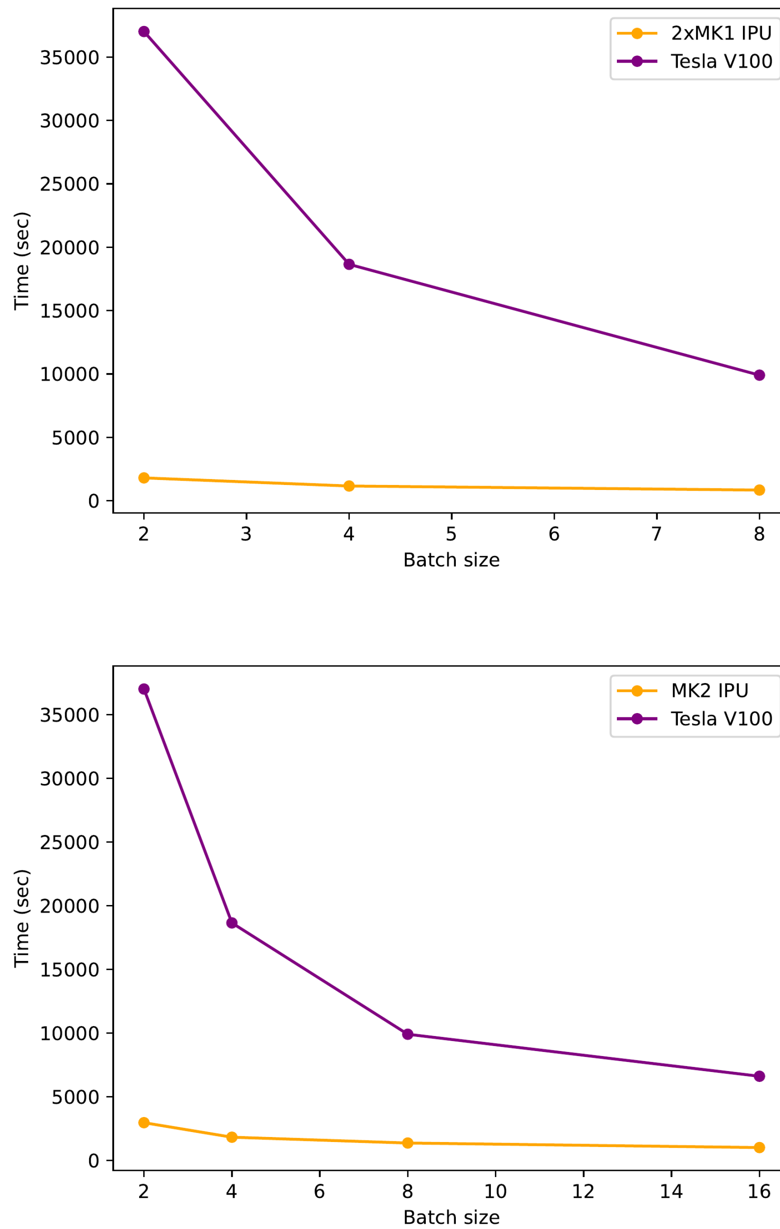



**Figure 7.2.** *Training time of BlazeFace model on the 300W-LP dataset. (top) The results for various batch sizes on V100 GPU and MK1 IPU chips. (bottom) The results for various batch sizes on V100 GPU and MK2 IPU chips.*

Απο τα παραπάνω διαγράμματα παρατηρούμε οτι και τα δύο τσιπ IPU υπερτερούν της V100 Tesla GPU. Πιο συγκεκριμένα, η επίδοση των IPU για μικρά μεγέθη παρτιδών (batch sizes) είναι σαφώς ανώτερη αλλά ακόμα και για μεγαλύτερες τιμές το δίκτυο εκπαιδεύεται σχεδόν στο μισό χρόνο σε σύγκριση με την GPU.

Στα πειράματά μας, πραγματοποιήσαμε επίσης δοκιμές με μεγαλύτερα μεγέθη παρτίδων για την εκπαίδευση του μοντέλου μας στην GPU Tesla V100, καθώς προσφέρει μεγαλύτερο ποσό συνολικής μνήμης σε σύγκριση με τις IPUs. Για την εκπαίδευση στη βάση δεδομένων FDDB, χρησιμοποιήσαμε μεγέθη παρτίδων έως 64 και έως 128 για τη βάση δεδομένων 300W-LP. Μια λεπτομερής παρουσίαση των αποτελεσμάτων της εκπαίδευσης μπορείτε να βρείτε στους πίνακες του παραρτήματος A.1 και A.2. Αυτοί οι πίνακες περιέχουν κάποιες πρόσθετες πληροφορίες σχετικά με τους συνολικούς χρόνους ταχύτητας εκπαίδευσης (χρόνος ανά εποχή, χρόνος ανά βήμα), καθώς και την επιτυγχανόμενη απώλεια (loss) και την απώλεια επικύρωσης (validation loss) για κάθε εκτέλεση.

Η υπερπαράμετρος batch size επηρεάζει σημαντικά την εκπαίδευση ενός μοντέλου και θα πρέπει να εξετάζεται προσεκτικά με βάση τα ειδικά χαρακτηριστικά του εκάστοτε συνόλου δεδομένων, την αρχιτεκτονική του μοντέλου και τους διαθέσιμους υπολογιστικούς πόρους. Η χρήση μεγαλύτερου μεγέθους παρτίδας μπορεί να οδηγήσει σε πιο σταθερά βάρη (weights) και ταχύτερη σύγκλιση (convergence) [49] [50]. Αυτό οφείλεται στο γεγονός ότι ένα μεγαλύτερο μέγεθος παρτίδας επιτρέπει στον αλγόριθμο βελτιστοποίησης να χρησιμοποιεί περισσότερες πληροφορίες από τα δεδομένα εκπαίδευσης για τον υπολογισμό της συνάρτησης απώλειας σε σχέση με τις παραμέτρους του μοντέλου. Με περισσότερες πληροφορίες, τα μεγέθη που υπολογίζονται είναι πιο αντιπροσωπευτικά ως προς την πραγματική υποκείμενη κατανομή των δεδομένων και λιγότερο ευαίσθητα στις τυχαίες διακυμάνσεις, γεγονός που μπορεί να οδηγήσει σε πιο σταθερές ενημερώσεις των βαρών ενός μοντέλου. Ωστόσο, υπάρχουν και ορισμένα μειονεκτήματα όταν χρησιμοποιούνται μεγαλύτερα μεγέθη παρτίδων, όπως η ανάγκη για περισσότερη μνήμη και περισσότερους υπολογιστικούς πόρους.

Από την άλλη πλευρά, τα μικρότερα μεγέθη παρτίδων οδηγούν σε συχνότερες ενημερώσεις βαρών ενσωματώνοντας πιο ποικίλες πληροφορίες στο μοντέλο, αλλά μπορεί να οδηγήσουν σε πιο θορυβώδει βάρη, καθώς κάθε παρτίδα αντιπροσωπεύει μόνο ένα μικρό δείγμα του συνολικού συνόλου δεδομένων και μπορεί να υπόκειται σε μεγαλύτερη μεταβλητότητα δειγματοληψίας. Επιπλέον, ένα μικρότερο μέγεθος παρτίδας μπορεί να οδηγήσει σε πιο δραστικές αλλαγές στα βάρη και μπορεί να απαιτεί πιο συχνές προσαρμογές στον αλγόριθμο βελτιστοποίησης (optimizer). Για παράδειγμα, ορισμένοι αλγόριθμοι βελτιστοποίησης, όπως ο SGD [51], αποδίδουν καλύτερα με μικρά μεγέθη παρτίδων, ενώ άλλοι, όπως ο Adam [52], μπορούν να διαχειριστούν και μεγαλύτερα μεγέθη παρτίδων.

Είναι σημαντικό να σημειωθεί στο σημείο αυτό ότι, εκτός από τη βελτιστοποίηση του συνόλου δεδομένων εισόδου tf.Data.Dataset και τη μεταγλώττιση με XLA, δεν πραγματοποιήσαμε καμία άλλη λεπτομερή ρύθμιση των υπερπαραμέτρων στα πειράματα εκπαίδευσης (π.χ. optimizer, learning_rate κ.λπ.). Ο στόχος των πειραμάτων που πραγματοποιήθηκαν είναι να

συγκρίνουμε την απόδοση της εκπαίδευσης σε κάθε πλατφόρμα όσον αφορά την ταχύτητα εκτέλεσης.

Η χρήση μεγαλύτερων μεγεθών παρτίδων είχε ως αποτέλεσμα πολύ καλύτερες επιδόσεις στη V100 Tesla GPU. Οι GPU είναι βελτιστοποιημένες για παραλληλισμό και υψηλή απόδοση, γεγονός που τις καθιστά κατάλληλες για γρήγορη επεξεργασία μεγάλων παρτίδων δεδομένων. Ωστόσο, καθώς μειώνεται το μέγεθος της παρτίδας, η επιβάρυνση που συνδέεται με τη συνεχή μεταφορά δεδομένων και τη διαχείριση της μνήμης μπορεί να γίνει αρκετά έντονη και να οδηγήσει σε χαμηλότερες επιδόσεις.

Οι IPU, από την άλλη πλευρά, αποδίδουν καλύτερα σε μικρά μεγέθη παρτίδων. Θεωρούμε ότι αυτό οφείλεται στην distributed αρχιτεκτονική τους, η οποία είναι βελτιστοποιημένη για πράξεις πινάκων και high band επικοινωνία υψηλής ταχύτητας μεταξύ των στοιχείων επεξεργασίας (π.χ. I/O tiles, IPU exhange fabric). Αυτό επιτρέπει στις IPUs να χρησιμοποιούν αποτελεσματικά τους πόρους τους και να επεξεργάζονται αποτελεσματικά μικρές παρτίδες δεδομένων, μειώνοντας παράλληλα την επιβάρυνση μνήμης που συνδέεται με μεγαλύτερα μεγέθη παρτίδων.

# Part II

# English Text

Chapter 1

# Introduction

In recent years, the research and development in AI, and more specifically its subset Machine Learning (ML), has gradually increased, spreading in several discipline fields. ML consists of several algorithms and paradigms, in which the most impactful ones are brain-inspired techniques. Among these, one that is based on Artificial Neural Networks (ANNs) has overcome the human accuracy, namely Deep Learning (DL) [5]. The DL techniques have shown many advantages over previous techniques, on the ability to work directly on raw data in large quantities.

These networks are called Deep Neural Networks (DNNs) and have achieved great successes in various research topics such as computer vision, natural language processing, speech recognition. Among those topics, computer vision tasks such as image classification [6], object detection [7] [8] and image segmentation [9] have attracted increasing research interests due to the potential in a wide range of real-world applications like autonomous driving, human-machine interaction and medical image analysis.

However, these applications often require vast amounts of data and compute resources for both training the models and deploy inference tasks. The traditional CPUs, which are considered the backbone of computational power, are often inadequate for handling efficiently the scale and complexity of these tasks. Therefore, there is a significant demand for better hardware accelerators that can handle the computational demands of these applications. In recent years, the GPU and IPU have emerged as popular hardware accelerators for Machine Learning and Deep Learning tasks, due to their ability to parallelize computations and perform matrix multiplications at high speeds. In this context, evaluating the performance of the available hardware accelerators is critical for the development of effective and efficient Machine Learning and Deep Learning applications that can handle the computational demands of the future.

In this thesis, we aim to evaluate the performance of GPU and IPU harware platforms by deploying Machine Learning algorithms and Deep Learning models in both inference and training tasks. To evaluate the inference task, we developed a real-world application for eyeblink-conditioning, which involves detecting the closure of eyelids across time, using solutions from the ML and DL fields. This is a critical task in neuroscientific research, where eyeblink-conditioning experiments are frequently used to study the neural

processes underlying learning and memory. By automating and accelerating the eyeblink response detection process, we aimed to achieve real-time processing speeds and enable neuroscientists to adjust conditioning experiments in real-time. For the training task, we constructed an end-to-end training pipeline for a CNN image-based DL model called BlazeFace, which is a face detection model developed by Google [10].

## 1.1 Motivation

Classical conditioning is a learning process that occurs through associations between an environmental conditioned stimulus (CS) and a naturally occurring unconditioned stimulus (US). One of the most famous examples of classical conditioning is Pavlov's experiment with dogs [53], in which the dog salivates in response to a bell tone. The conditioned stimulus (CS) is a neutral stimulus (e.g., the sound of bell), the unconditioned stimulus (US) is biologically potent (e.g., food) and the unconditioned response (UR) to the US is an unlearned reflex response (e.g., salivation). After pairing is repeated the organism exhibits a conditioned response (CR) to the conditioned stimulus (CS), when it is presented alone.

The Erasmus MC Neuroscience department conducts a classical conditioning experiment which is called Eyeblink conditioning [37]. In this experiment, the CS is a sound that is paired with an airpuff to the subject's eye, the US. Closure of the eyelid is the natural response to this airpuff. After a period of repeated paired presentations, an eyeblink develops which precedes the airpuff. This eyeblink is produced from a learned association between the tone and the upcoming airpuff and is therefore considered a conditioned response (CR).

Eyeblink conditioning (EBC) is a well studied form of classical conditioning used by scientists to extract valuable information about neural structures and mechanisms that underlie learning and memory. Recent studies that make use of this particular conditioning method are [1], where they study what happens in human brains when new motor skills are learned, [2], which studies the impact of several disorders from the autism spectrum on the CRs and [3], where the EBC method is used to investigate cerebellar dysfunction in schizophrenia disorders.

The subject is recorded with a high-speed camera, in order to capture the eyeblink response. When this experiment is performed on humans, a wide area needs to be recorded on video, as the subject moves relatively freely during the test and therefore the position of the head is not fixed. An algorithm combination is then applied to each image in the generated video to measure the closure of the eye.

An early approach for measuring the eyeblink response was to analyse the closing of the eyelids in a video, frame by frame. Scientists had to manually select half of the face in the first frame and then manually select the eye of interest. A template matching

procedure [4] was then used to crop the eye region for each subsequent frame and finally calculate the eyelid closure. The drawbacks of this approach were the need of manual intervention (selecting the face and eye for each new trial) and the necessity of off-line storage, as the actual video must be viewed and processed by a human before the eye-blink graph can be extracted. In addition, the matching-template procedure is sensitive in scale and rotation changes, which leads in dropped frames when the subject moves during the recording of the video. Each blink video has a length of 2 second, shot at 333 frames per second (FPS), and therefore each video contains 666 frames.

Creating an automatic procedure of face and eye detection, at a processing speed that is able to keep up with the video frame rate is an important step towards an online implementation of the eyeblink-response analysis. This would enable researchers to analyze the experiment results in real-time, which in turn would alleviate the need of human intervention and off-line storage. Furthermore, the exploration of using deep learning networks for real-time image processing tasks is of great interest, as most of state-of-the-art DNN approaches for face and landmark detection focus mostly on achieving a better accuracy, whereas the complexity of the model and computational issues rather stay in the background [23] [41].

In particular, Convolutional Neural Networks (CNNs) have been widely utilized and they have shown state-of-the-art performances on computer vision (CV) image-based tasks. With the advances in DNN development, larger models were introduced to tackle harder and more complex tasks, and therefore these CNN based approaches require a large amount of storage, run-time memory, as well as computation power in both training and inference time. In real-world scenarios inference is performed after the neural network has been trained and is used to classify or derive predictions from the given inputs. While in the case of inference, the network only experiences the forward-pass, during the training, it experiences both the forward-pass and the backward-pass. During the latter, the prediction is compared with the label, and the error is used to update the weights through the backpropagation process. As a consequence, training requires a much more extensive computational effort compared to that for inference.

In practice, CNN based models run on machines equipped with suitable hardware accelerators. The panorama of hardware solutions for the development and deployment of DNNs is wide, with the most common being general purpose Central Processing Units (CPU) and Graphics Processing Units (GPU). GPUs have evolved a lot over the years to reach very high performance at the moment and surpass CPUs. However, there is still a growing demand for specialized hardware accelerators with optimized memory hierarchies that can meet the compute and memory requirements of different types of complex DNNs, while maintaining a reduced power and energy envelope [54]. An example of a recent introduced DNN accelerator is Graphcore's Intelligence Processsing Unit (IPU). The IPU's design is based on the Bulk Synchronous Parallel (BSP) [19] model of computation and offers MIMD (Multiple Instruction, Multiple Data) parallelism. The IPU's approach of

accelerating computation is through shared memory, which is distinct from other hardware design approaches. An IPU offers small and distributed memories that are locally coupled to each other. There is no global memory, and cores must share data by passing messages over a high-bandwidth, all-to-all interconnect network. By evaluating the performance of these hardware accelerators in terms of latency, power consumption, and computational efficiency, we aim to provide insights into their strengths and weaknesses for image processing applications and contribute to the ongoing development of effective and efficient hardware and software solutions.

## 1.2 Thesis Scope

This thesis initially focuses on automating and accelerating the process of eyeblink-response detection from video in order to achieve real-time processing speed. Solutions provided from the fields of Machine Learning and Deep Learning are explored and compared to find a suitable algorithm combination which satisfies our project's requirements. Detection consists of two distinct phases: detection of the human face, followed by detection of the eyelid closure.

In the current eyeblink conditioning setup, the camera records the blinks at a framerate of 333 Hz, but it can go up to a maximum of 750 Hz. Neuroscientists have expressed their interest to increase the camera framerate to 500 Hz. This means that the maximum processing time for each frame is 2 ms. It is very unlikely that the selected algorithms will satisfy this requirement out of the box. Therefore, we will need to accelerate the chosen algorithms in order to reach the required processing speed. Machines equipped with multi-core CPUs and device accelerators, GPUs and IPUs, will be used to accelerate our selected solutions. Initially, different inference approaches on the different hardware platforms will be explored and then optimizations will be applied to achieve the maximum processing speed.

In addition to evaluating inference performance, this thesis also aims to test and benchmark the training process of an image-based CNN face detection model, using the BlazeFace model as an example. The training process is complex, requiring efficient utilization of both CPUs and device accelerators, such as GPUs and IPUs. We will explore and apply optimizations to various steps of the training pipeline to achieve optimal performance. By evaluating both inference and training performance, we can compare and contrast the capabilities of different hardware platforms, with a particular focus on GPUs and IPUs.

## 1.3 Contribution

The following contributions were made by the work of this thesis:

- The performance of three different face detection algorithms is evaluated on two

datasets containing a variety of face images. The BlazeFace model was selected as the algorithm that best suits our project requirements because of its combination of speed and accuracy.

- The BlazeFace face-detection model was accelerated on the GPU and IPU hardware platforms. Two distinct inference approaches were explored: initially for bulk computations and then a streamlined version for low-latency responses.
- An Ensemble of Regression Trees (ERT) is selected as the landmark-detection algorithm, which is used to estimate the closure of the eyelid.
- The ERT algorithm was accelerated with the use of Python multiprocessing library on a multi-core CPU system. A speedup of 14,8x and 17,5x was achieved compared to the sequential implementation by using 32 and 64 processes respectively.
- The accelerated face and landmark-detection algorithms were combined and three complete implementations for eyeblink-response detection were deployed on IPU and GPU hardware platforms and achieved a detection speed of 1441 FPS on MK2-IPU, 1367 FPS on MK1-IPU and 1658 FPS on Tesla V100 GPU, which are more than the 500 FPS required for real-time processing.
- An end-to-end training pipeline for the BlazeFace model was constructed to further test and compare the hardware accelerators for training CNN image-based models. The conducted experiments indicated that IPUs outperform GPUs in training speed, especially for small batch sizes where the IPUs were able to run 2-4 times faster.

## 1.4 Thesis organization

The rest of the thesis will be organized as follows: Chapter 2 provides background information on the eyeblink conditioning experiment, object detection in computer vision and convolutional neural networks. In Chapter 3 the different hardware platforms that will be used in this project are analyzed. Chapter 4 covers different solutions for all the tasks (face detection, landmark detection and eyelid-closure detection), which are needed to tackle the eyeblink conditioning experiment. Furthermore, all proposed solutions are compared and based on that a decision on which algorithms shall be used for each case is made. Finally, the details of the selected algorithms are covered as well. Chapter 5 provides a design analysis and details for the acceleration of our selected algorithms. In Chapter 6 the final implementation of the accelerated eyeblink-response detection is described and evaluated on the different hardware accelerators (GPU, IPU). In Chapter 7 the compute capabilities of the different hardware available platforms (IPU, GPU) are compared in the computationally intensive process of training an image-based CNN model. Finally, the thesis is concluded in Chapter 8.

Chapter 2

# Background

In this chapter, we provide background information for a proper understanding of the rest of the thesis. In Section 2.1, we describe the setup of the human eye-blink conditioning system which is currently used by the Erasmus MC Neuroscience department. Section 2.2 will cover the basics of object recognition and detection in computer vision. Finally, in Section 2.3 the underlying knowledge of convolutional neural networks will be discussed.

## 2.1 Eye-blink Conditioning

We have already described the eyeblink conditioning method in Chapter 1. In our case of human eyeblink conditioning, the high-speed camera is positioned approximately one meter away from the facing subject. The camera captures the subject's face as well as some surroundings. The subject is able to move freely a little without going out of the camera's scope. In order to minimize the movement and rotation of the face, the subject's attention is drawn towards the camera (e.g., there is a monitor showing a movie). Too much movement or rotation of the face has a significant impact on the recorded data which can be rendered useless. These properties (position, posture) are of great importance to the selection of face detection algorithm for our project. Using a potentiometer coupled to the eyelid [55] or an electromyography (EMG) on the muscle that closes the eyelid [56], are other methods used to record the eyeblink response. However, computer vision offers a much easier approach of recording the response.

## 2.2 Object Recognition and Detection

Computer vision is defined as the field of study that seeks to develop techniques to help computers see and understand the content of digital images such as photographs and videos. It is a multidisciplinary field that could broadly be called a subfield of artificial intelligence and machine learning, which may involve the use of specialized methods and make use of general learning algorithms. A wide range of application fields, such as medicine (medical image processing/analysis to diagnose a patient), robotics (autonomous vehicles, mobile robots) and computer-human interaction, employ computer vision tasks.

In our case, to record the blink response with the use of computer vision without human interaction, we shall primarily turn to the field of object recognition and detection. Object recognition in computer vision is the ability to determine whether or not an image, or a region of the image, contains a specific object (e.g a vehicle). In a multi-class approach, the goal of object recognition is to tell to which of the X discrete classes (e.g. bike, plain, car, etc.) the object in the picture belongs. Object detection aims to determine the location of some specific objects in digital images or videos. To be more specific, in object detection tasks, the systems are required to not only give the label, but also the location of the target instances. The location information is normally presented by giving the coordinates of the bounding boxes, and the regions which contain the target objects is called foreground, while the regions without targets are named background.

The methods to solve the object detection problem can be divided into hand-engineered feature-based machine learning approaches, or CNN-based deep learning approaches. For the machine learning-based approaches, the idea is to carefully design and extract the features from the images [11][12], and then design the classifiers (e.g. support vector machine) based on them. However, in recent years, deep learning-based approaches [13][14][15] have achieved a significant performance improvement comparing to those machine learning methods, and have been used in a wide range of real-world applications, such as face detection, human pose estimation, etc. Another advantage of deep learning-based methods is that feature extraction can be handled by typical convolution and/or fully connected layers, which makes the whole process end-to-end during both training and inference time.

### 2.2.1 Introductory Object Detection Concepts

#### Bounding Box

A bounding box is an ideal rectangle that serves as a point of reference for object detection and denotes a collision box for that object. Axis-aligned bounding boxes are mainly used in the task of object detection, where the aim is identifying the position and type of multiple objects in the image. Therefore, in addition to the position of the object inside an image it can be used to define additional characteristics of an object, such as class (e.g face, no-face) and confidence (how likely is the object to be at that location).

There are two main conventions followed for representing axis-aligned bounding boxes:

- Specifying the box with respect to the coordinates of its top left ($x_{min}, y_{min}$), and the bottom right point ($x_{max}, y_{max}$).

- Specifying the box with respect to its center ($x_c, y_c$), and its width and height (w, h).

### Non-Maximum Suppresion

Non-Maximum-Suppression [16] is a computer vision method that selects a single entity out of many overlapping entities. The criterion is usually discarding entities that are below a given probability bound. It is a common issue of object detection tasks to have multiple bounding boxes describing the same object. The NMS algorithm will select the boxes with the maximum confidence and suppress all the other predictions overlaping with the selected predictions by more than 50%. The procedure is repeated until one box is finally selected (Fig. 2.1).

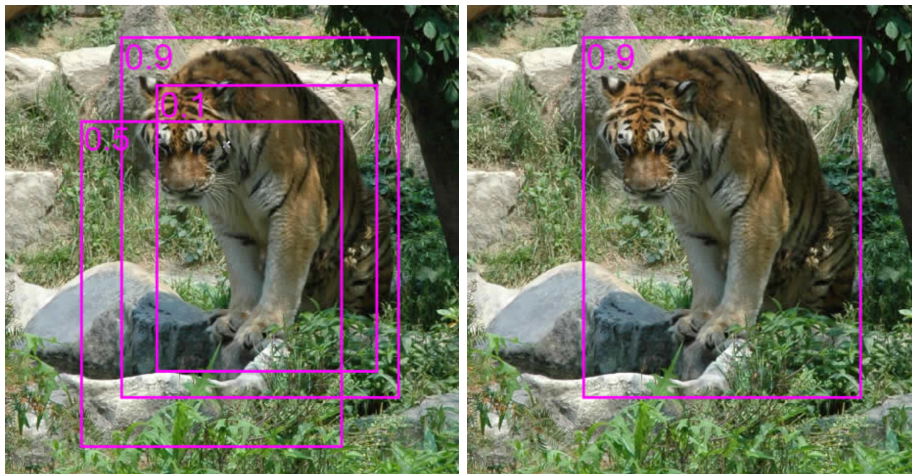



**Figure 2.1.** *Non-Maximum Suppresion example. The NMS algorithm kept only one box with the highest confidence. Image from:* `https://www.interstellarengine.com/ai/Non-maximum-suppression.html`.

### Intersection Over Union

A mechanism to confirm that a detection is correct is necessary in every object detection system. Most systems rely on the bounding box overlap ratio or intersection over union measure, or IoU for short. As defined in [7], "the overlap ratio between a predicted bounding box $B_p$ and ground truth bounding box $B_{gt}$ is given by equation 2.1,

$$IoU = \frac{area(B_p \cup B_{gt})}{area(B_p \cap B_{gt})} \tag{2.1}$$

where $B_p \cap B_{gt}$ denotes the intersection of the predicted and ground truth bounding boxes and $B_p \cup B_{gt}$ their union" (Fig. 2.2), and where the area of a region is measured by the number of pixels it contains. In most detection systems, a detection is considered 'correct' when this value is greater than 50%. This choice was generally made following the PASCAL VOC [7] protocol.

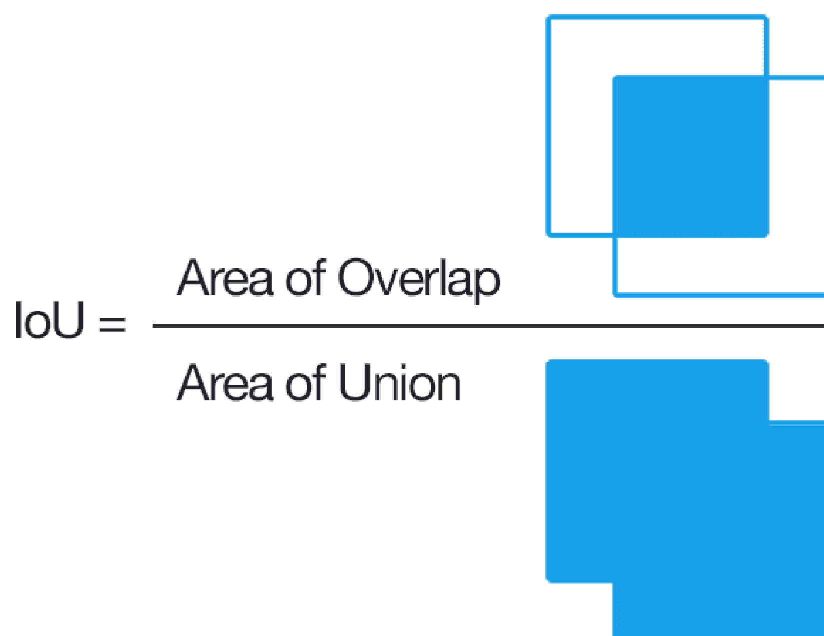



**Figure 2.2.** *IOU definition. Image from:* https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

## 2.3 Convolutional Neural Networks

Convolutional networks, also known as convolutional neural networks, or CNNs, are a specialized kind of neural networks in deep learning, which are widely used in visual recognition problems. Compared to other visual recognition methods, such as SVM classifiers, CNN can handle the input image to the desired result end-to-end without any human-designed features. A typical CNN consists of the combination of the following components:

**Convolutional Layer:** Central to the convolutional neural network is the convolutional layer (Fig. 2.3) that gives the network its name. This layer performs an operation called a convolution. In the context of a convolutional neural network, a convolution is a linear operation that involves the multiplication of the input with a set of weights, also known as kernels or filters, which are determined through network training. These filters have a predefined size, smaller than the input, and during training the filter convolution operation is performed across the entire input in a sliding window fashion. If a filter is designed to detect a specific type of feature in the input, then applying that filter systematically across the entire input image allows the filter an opportunity to discover that feature anywhere in the image.

As the filter is applied systematically across the input array, the result is a two-dimensional array comprised of output values that represent a filtering of the input. This output array is called activation map or feature map, where each position value of the map expresses the probability of the desired feature to be located in this area of the original

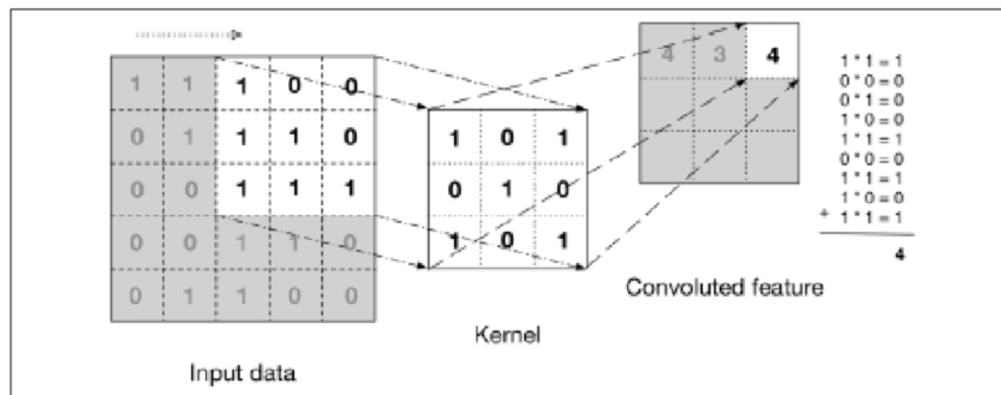



**Figure 2.3.** *Convolution Layer. Image taken from [57]*

image. In addition, multiple filters per convolution layer can be used, outputting multiple activation maps corresponding to different features (one for each filter), and therefore the output of each convolution layer is a three-dimensional "image" of great depth, consisting of different activation maps.

Convolutional layers are not only applied to input data, e.g. raw pixel values, but they can also be applied to the output of other layers. The stacking of convolutional layers allows a hierarchical decomposition of the input. Consider that the filters that operate directly on the pixel values learn to extract low-level features, such as lines. The filters that operate on the output of the first line layers may extract features that are combinations of lower-level features, such as features that comprise multiple lines to express shapes. This process continues until very deep layers are extracting faces, animals, houses, and so on. The order and the way in which the various convolutions (stride, padding) are done are design choices of the CNN designer that we will not deal with in this work.

**Activation Layer:** Most systems that a Convolutional Neural Network is called upon to mimic are natural systems, and for this reason their behavior is not completely linear. In order to introduce the necessary non-linearity into the network, each convolutional level is followed by an activation level which applies to its output a (non-linear) activation function. The most widely used activation function is the Rectified Linear Unit (ReLU).

**Pooling Layer:** We saw that convolutional layers in a CNN systematically apply trained filters to input images in order to create feature maps that summarize the presence of those features in the input. However, a limitation of the feature map outputs is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image. One approach to address this sensitivity is to down sample the feature maps by using a Pooling layer.

A pooling layer is a new layer added after the convolutional layer operating upon each feature map separately to create a new set of the same number of pooled feature maps. Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

This has the effect of making the resulting down-sampled feature maps more robust to changes in the position of the feature in the image, referred to by the technical phrase "local translation invariance".

**Fully connected layer:** Typically, fully connected layers are added at the end of a CNN to connect the hidden layers and the output layer. The parameters in fully connected layers are trained to summarise the features and map them to a vector with each element representing the score of an output class.

In CNNs, the convolutional layers usually takes most of the computation resources, due to a large number of floating-point multiplications between the high-dimension convolutional kernels and the inputs, which are the bottleneck of execution speed for most of the typical CNN models.

#### Training and Inference

Neural Networks learn to achieve the desired results by modifying their internal parameters, i.e., weights and biases. The phase in which the network learns is called training. Once the network has been trained, it can be used to process unknown input during the inference phase when deployed in practise.

One of the most used learning paradigms is supervised learning, thanks to the large amount of (labeled) data that has become available in the so-called big-data era. Supervised learning requires labeled data, i.e., input-output pairs, where the output is the result that the network should obtain from the related input. Supervised learning consists of three steps repeated until convergence:

1. **Forward pass:** the input is fed into the network that produces an output.
2. **Backward pass:** a loss L is computed comparing the produced output and the desired output. The loss L is then used for the backpropagation algorithm [58], which applying the chain rule of calculus computes the gradient $\partial L/\partial w$ for each weight (and bias) of the network.
3. **Parameters update:** each weight and bias is updated by an amount proportional to its gradient. All the gradients can be multiplied by the same factor, defined learning rate, or more complex optimization algorithms can be used, such as Gradient Descent with Momentum [59] or Adam [52].

Other learning paradigms are unsupervised learning and reinforcement learning. Unsupervised learning works with unlabeled data and consists of finding common patterns and structures that data may have in common. Reinforcement learning involves the network (agent) interacting in an environment. An interpreter assesses the correctness of the interactions and returns a reward or punishment to the agent, who aims to maximize the reward.

Chapter 3

# Hardware Platform

## 3.1 General Purpose Processors and GPUs

### 3.1.1 Central Processing Unit (CPU)

As a wide variety of computing problems started to get more complex and computational intensive, the need of faster processors in order to tackle these problems became essential. The focus of early implementations of CPUs, was on making a single processor as fast as possible, resulting on a rapid increase of power consumption. Modern CPUs are built as general-purpose processors, with several features added to be able to support an extensive range of applications. CPUs belong to the category of spatial architectures where the computational structure consists of multiple processing units. These units can have internal control, register files (RF) to store data and be interconnected to exchange data. Vector CPUs have multiple ALUs that can process multiple data in parallel. Most of them adopt the Single-Instruction Multiple-Data (SIMD) execution model, which applies a single instruction to different data simultaneously.

Among the different available technologies, CPU cores are the least used for DNNs inference and training. CPUs have the advantage of being easily programmable to perform any kind of task. Still, their throughput is limited by the small number of cores and, therefore, by the small number of operations executable in parallel. However, the hardware/software stack of CPUs is already well-established and understood and CPUs are also inevitably present in any system. They can provide reasonable speedups on a broad range of applications. In mobile and embedded domains, CPU is still the most widely used computing system due to its high availability, portability and software support. The work of [17],discusses a number of techniques for optimizing DL applications on mobile, server and cluster of CPUs.

### 3.1.2 Graphics Processing Units (GPU)

A CPU can efficiently work together with a Graphics Processing Unit (GPU) to increase the throughput of data and the number of concurrent calculations within an application. While individual CPU cores are faster (as measured by CPU clock speed) and smarter (as measured by available instruction sets) than individual GPU cores, the sheer number of GPU cores and the massive amount of parallelism that they offer makes them very well-suited to accelerate programs with a large amount of data parallelism. Figure 3.1 shows a schematic overview of the differences between a multi-core CPU and GPU.

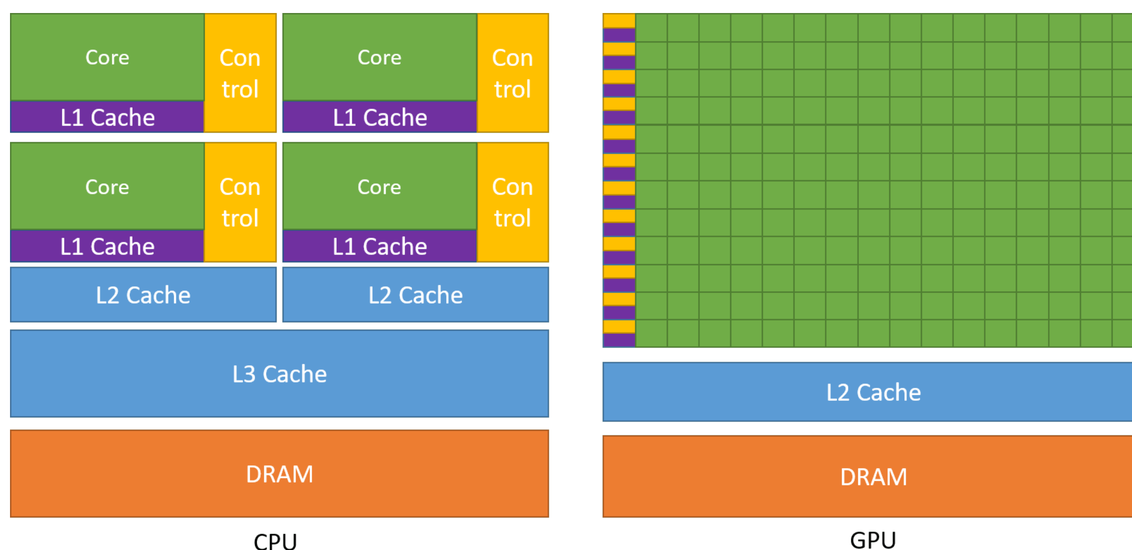



**Figure 3.1.** *Schematic comparison between the chip layout of a multi-core CPU and GPU [18]*

A CPU can never be fully replaced by a GPU, as GPUs complements the CPU architecture by allowing repetitive calculations within an application to be run in parallel while the main program continues to run on the CPU. The CPU can be thought of as the taskmaster of the entire system, coordinating a wide range of general-purpose computing tasks, with the GPU performing a narrower range of more specialized tasks (usually mathematical). Using the power of parallelism, a GPU can complete more work in the same amount of time as compared to a CPU. GPUs were originally designed to create images for computer graphics and video game consoles, but since the early 2010's, GPUs are also utilized to accelerate many applications/problems involving massive amounts of data.

In order to exploit data-parallelism on a GPU the programmer must divide the problem into parallelizable pieces that each get processed by a thread. GPUs are capable to execute a large number of parallel threads. Threads are grouped in blocks and a set of blocks comprises a grid. Blocks are organized as a 3D array of threads and each block has a unique block ID (blockIdx). Threads are executed by kernels, compiled functions that gets executed on GPUs, and each thread is assigned a unique thread ID (threadIdx). The execution flow is shown in Fig. 3.2. The execution starts with a host(CPU) program that launches a kernel on Device (GPU). The GPU scheduler generates a large number of

threads to perform data-parallelism. Before starting the kernel, all the necessary data is transferred from host to device memory. The CPU kick starts the kernel function then execution flow is moved to the device.

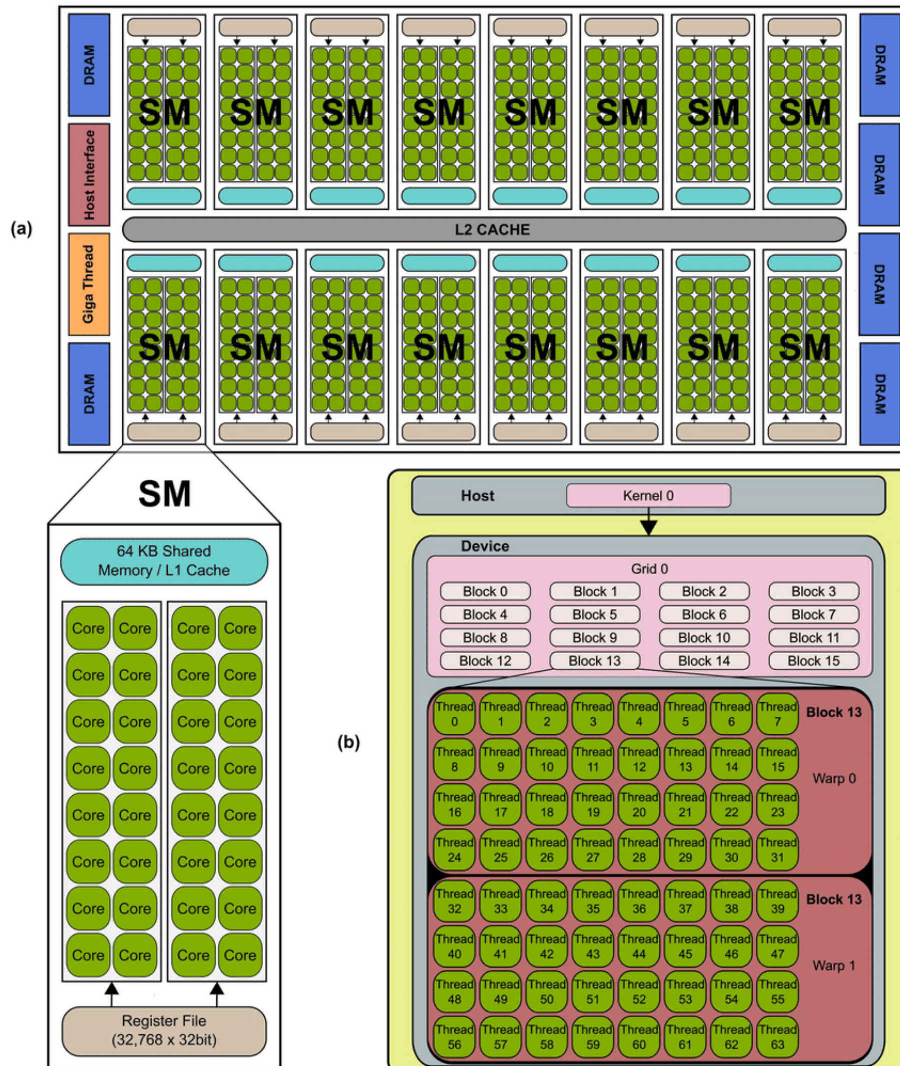



**Figure 3.2.** *The GPU is comprised of a set of Streaming MultiProcessors (SM). Each SM is comprised of several Stream Processor (SP) cores, as shown in (a). The GPU resources are controlled by the programmer through the CUDA programming model, shown in (b). Image from [60].*

During execution, a thread block gets assigned to a Streaming Multiprocessor (SM). The streaming multiprocessors (SMs) are the part of the GPU that actually runs our GPU-kernels. Each SM contains:

- Several caches:
  - Shared memory for fast data interchange between threads
  - Constant cache for fast broadcast of reads from constant memory
  - Texture cache to aggregate bandwidth from texture memory

  - L1 cache to reduce latency to local or global memory

- Execution cores for integer and floating-point operations

- Thousands of registers that can be partitioned among threads of execution.

- Warp schedulers that can quickly switch contexts between threads and issue instructions to warps that are ready to execute

Thread instructions are then executed in groups of 32, called warps. Threads in a warp will perform the same operation on independent data. This execution model is referred to as SIMT (Single Instruction, Multiple Threads).

Even though GPUs have evolved deep memory hierarchies featuring both cache and scratchpad memories [61], their power lies within their fundamental approach to hiding memory latency and the ability to inexpensively switch among threads. In this approach, when a warp of threads is awaiting operands from main memory, the hardware can suspend them and switch to another warp that has received its operands from memory and is ready to continue [62].

### 3.1.3 Programming tools

GPU-based systems are especially well suited for Deep Learning workloads, for both DNNs' inference and especially training. They contain up to thousands of cores to work efficiently on highly-parallel algorithms. Matrix multiplication, the core operation of DNNs, belongs to this class of parallel algorithms. Among the GPU designers, Nvidia can be considered the winner of the ANN/DL applications. In fact, the most popular DL frameworks, such as TensorFlow [63], PyTorch [64] and Caffe [65], support execution on Nvidia GPUs through the Nvidia cuDNN library [66], a GPU-accelerated library of primitives for DNNs with highly-optimized implementations of standard layers. DL frameworks allow to describe very complex neural networks in a few lines of code and run them on GPUs without needing to know GPU programming. cuDNN is part of CUDA-X AI [67], a collection of Nvidia's GPU acceleration libraries that accelerate DL and ML.

## 3.2 Graphcore Intelligence Processing Unit (IPU)

The Intelligence Processing Unit (IPU) is a new kind of massively parallel processor developed by Graphcore specifically for Artificial Intelligence/Machine Learning (AI/ML) workloads. Graphcore provides two generation of IPU chips, the first-generation Colossus MK1 IPU processor - GC2 and the second-generation Colossus MK2 IPU processor - GC2000. In this chapter, we introduce the fundamentals of the IPU's architecture and its programming paradigm.

### 3.2.1 IPU Architecture

The cornerstone of an IPU-based system is the IPU processor which achieves efficient execution of fine-grained operations across a relatively large number of parallel threads. Each IPU processor contains four main components: IPU-tile, IPU-exchange, IPU-link and the PCIe interface.

An IPU processor contains multiple **IPU tiles** where each tile consists of a single multi-threaded processor and its local memory (Figure 3.3). Each tile-core can run six parallel threads in a "round-robin" fashion. There is always one supervisor thread that runs the top level control program of the tile and spawns worker threads to execute tasks. The tile-memories are implemented as SRAMs and therefore offer much higher bandwidth and lower latency than DRAMs. An IPU chip communicates with the host (CPU) to transfer back and forth required data through the PCIe interface. While the first-generation IPU uses only the local memory of its tiles to do so, the second-generation IPU has the option to use external memory (DRAM) for storing model data and parameters. This data can be streamed into the IPU when needed.

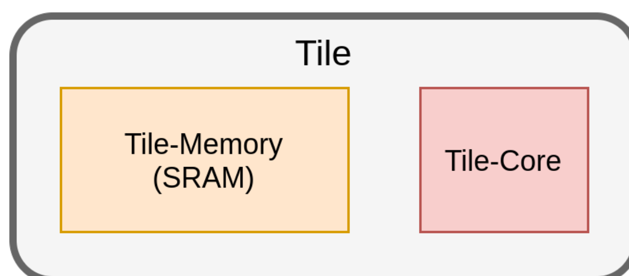



**Figure 3.3.** *IPU-tile*

All tiles within an IPU are connected to an ultra-fast, all-to-all communication fabric called the **IPU-exchange**, an on-chip interconnect for high-bandwidth, low-latency communication among them. In addition, each IPU contains ten **IPU-link** interfaces; the IPU-Link is an interconnect that allows different IPUs to exchange data directly, without going through the host processor or host memory. Besided that, each IPU also contains two PCIe links for communication with CPU-based hosts. We illustrate the IPU architecture with a simplified diagram in Figure 3.4

The IPU-exchange is what allows tiles on an IPU system to work together and exchange data efficiently with each other. A system with multiple IPUs exposes the single IPU devices independently, but it also exposes Multi-IPUs. A Multi-IPU is a virtual IPU device that is comprised of multiple physical IPUs and offers all their memory and compute resources as if they belonged to a single device. The combination of multiple physical IPUs into a virtual single device, allows training and inference of models larger than a single IPU's capacity, while taking advantage of the cumulative compute power. Scaling an ap-

plication to multiple IPUs comes with no additional development effort as the same APIs can target one physical IPU or Multi-IPUs indifferently. Furthermore, in multi-processor systems, the IPU exchange and the links work together to support tile-to-tile communication, regardless of where in the system the two endpoints are located.

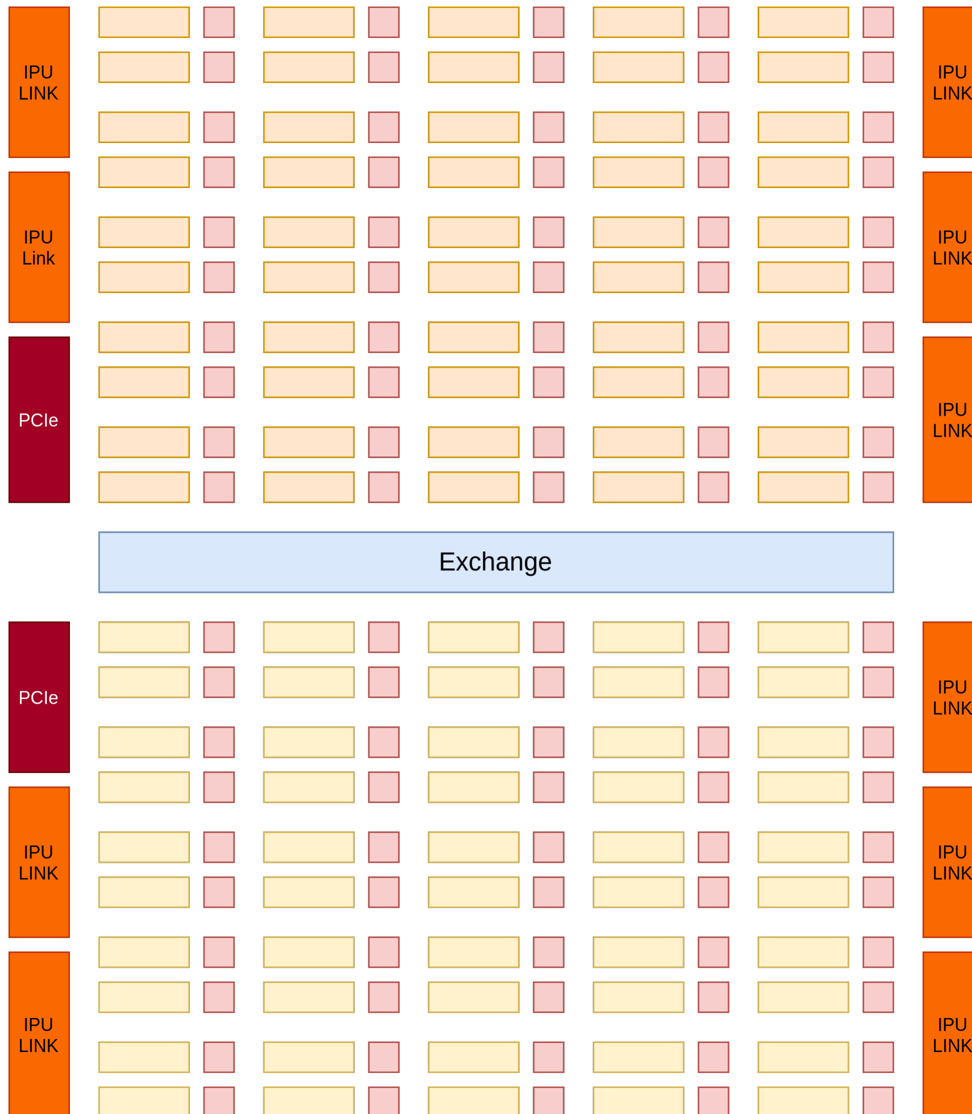



**Figure 3.4.** *Simplified illustration of an IPU processor.*

### 3.2.2 Programming model

IPU-programs are executed on a set of IPUs. This set must be selected by the user before compilation and it cannot change over the course of the program's execution. A program will follow a fixed control flow and operate on large multi-dimensional typed arrays of data (of fixed size) called tensor variables. In the general case, tensors are data structures that are used to describe scalars, vectors and matrices. A single variable may be stored across the memory units of multiple tiles. Each element of the variable is placed on or "mapped" to a specific tile. This is called the tile mapping of the variable.

There are two core operations that a running program uses to manipulate tensor variables; copying data and executing compute sets. Each compute set consists of many vertices that are compute tasks. The vertices determine how a compute set splits its tasks into fine-grained, parallel pieces of computation to use the many tiles and threads on the IPU. Each vertex is tied to a specific tile of the IPU and executes a small piece of code that processes its own input and output. The combination of all the vertices from the multiple compute sets in a program forms the computational graph (Fig. 3.5).

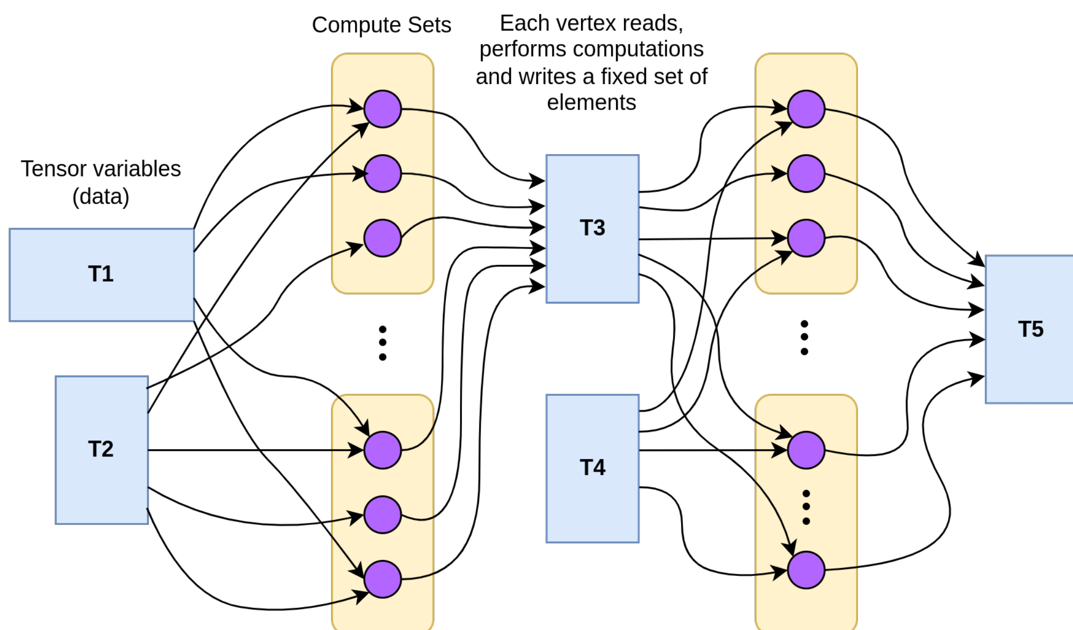



**Figure 3.5.** *Graph representation of variables and processing*

.

In addition to the basic operations which each tile processor has an instruction set specifically designed from scratch for machine learning and artificial intelligence tasks. The instruction set contains:

- Control flow instructions (jumps, conditionals) that can execute arbitrary control flow. The control flow on each processor is independent from those on the other processors.

- Memory access instructions
- Arithmetic instructions for integers and floating-point operations. The floating-point instructions include single-precision (32 bit) and half-precision (16 bit) floating-point operations.
- Instructions to compute common transcendental functions (for example, the exponential function)
- Instructions for random number generation.

### 3.2.3 Parallel Execution

The IPU uses a model of execution called bulk-synchronous parallel (BSP) model [19], where the tiles within the IPU alternate between exchanging data and performing computations on their local data. The execution of a task is organized into steps. Each step is composed of a local computation phase, followed by a global synchronisation phase and finally the data exchange:

- in the **local computation phase**, all tiles execute in parallel, operating solely on their local data;
- as soon as a tile finishes executing, it enters the **synchronisation phase** where it's waiting on other tiles to finish their computation;
- finally when all the tiles have reached the synchronisation state, the **data-exchange phase** begins, where required data is copied between the tiles.

Since data of a tensor variable are stored across the memory units of multiple tiles and each tile can solely operate on its local data, explicit cross-tile data movement instructions are needed during execution of a task. There are data dependencies among the participating tiles, where each tile may depend on the output results of other tiles to continue its computations, and therefore data exchanges between them occur via the IPU's exchange fabric. The global synchronisation phase ensures that all tiles have completed performing their computations (fully or up until a certain point) before sending/receiving data. The waiting period between syncs is not fixed but determined by the time taken for the computation.

The whole process then repeats as all tiles re-enter the compute phase. Each of these steps occurs in parallel across all tiles and the whole IPU can be viewed as executing a sequence of these steps (Fig. 3.6). The order of the steps to be executed is determined by a control program which is loaded on every tile by the host to control the execution of compute and exchange sequences.

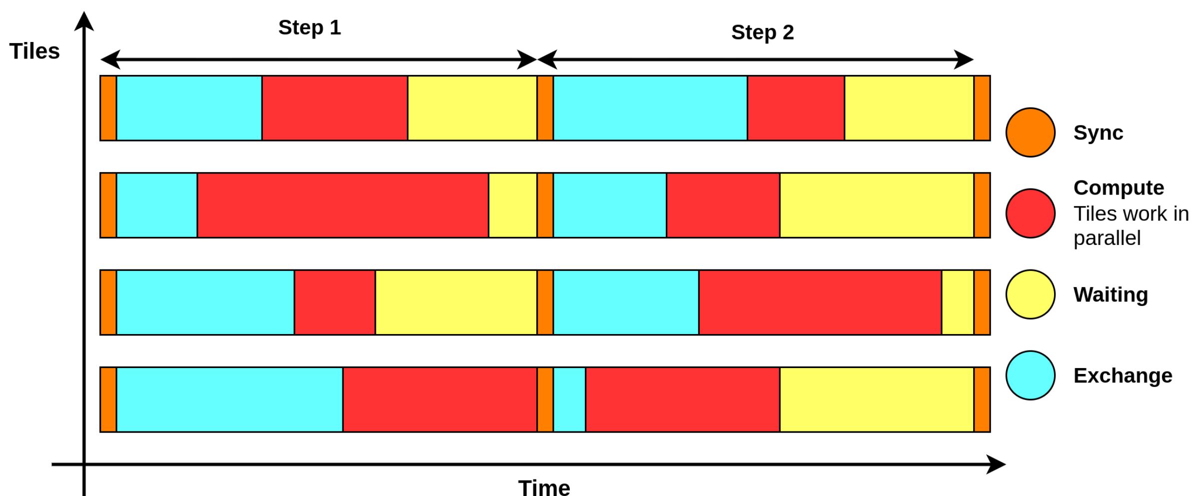



**Figure 3.6.** *Bulk-synchronous parallel execution model across IPU tiles*

.

### 3.2.4 Supported programming tools

Graphcore provides the Poplar Software Development Kit (SDK). This software supports many high-level machine-learning frameworks, C++ and assembly language for programming the IPU. The IPU software supports, among others, TensorFlow [63], Keras [68] and PyTorch [64], all user-friendly frameworks allowing programmers without specific hardware knowledge to access high-performance computing.

Keras is a high-level library, which is used for constructing models using a set of high-level Layer objects, and runs on top of a machine-learning framework such as TensorFlow. TensorFlow is a powerful graph-modelling framework that is widely used for the development, training and deployment of deep-learning models. These are two libraries that provide many useful tools for developing deep-learning programs.

Graphcore provides an implementation of TensorFlow, which also includes Keras support for IPUs, that enables us to train, infer and evaluate models on the IPU hardware. At the time of writing this thesis, version 2.4 of Poplar Software Development Kit (SDK) and version 2.4.4 of TensorFlow were available for developing deep-learning applications.

## 3.3 Discussion

The architecture of IPUs differs significantly from CPUs and GPUs that are commonly used for training machine-learning algorithms. However, even with the vectorized data processing, CPUs are non-competitive with GPUs in aggregate floating-point arithmetic on large and complex workloads. GPUs, on the other hand, have architecturally simpler cores than CPUs but do not offer branch speculation or out of order processing. The typi-

cal arrangment of GPUs is clusters, so all cores in a wrap execute the same instruction at any point in time. Because of this architecture, GPUs excel at regular, dense, numerical, data-flow-dominated workloads and tend to be more energy efficient than CPUs.

The IPU's approach of accelerating computation is through distributed memory, which is distinct from other hardware platforms. An IPU offers small and distributed memories (SRAMs) that are locally coupled to each other through a very fast all-to-all communication fabric. Such a structure allows cores to access data from local memory at a fixed cost that is independent of access patterns, making IPUs more efficient than GPUs when executing workloads with irregular or random-data-access patterns as long as the workloads can fit in IPU memory.

Chapter 4

# Algorithm selection

The algorithm selection is of great importance to this project. We need to make sure that the selected algorithms are not only able to detect the eyeblink response accurately in our recording settings, but also to fit additional requirements in terms of detection speed. In this chapter, we investigate alternative methods for both face detection and eyelid-closure detection.

## 4.1 Requirments for eyeblink-response algorithms

The recording settings of the eyeblink-response videos have previously described in Section 2.1. A set of requirements was drafted regarding the location and posture of the subject, and the recording environment.

Rotation in the three spatial dimensions is defined as yaw, roll and pitch. Since the subject is focused on the screen that is in the same direction as the camera, these are limited to $\pm 20°$, $\pm 25°$and $\pm 40°$, respectively. We observe occlusion of the faces only when the subject is wearing glasses. This requirement is considered a complex one because, while our detector may still be able to detect a face, when it is rotated in certain angles, it is possible that the frame of the glasses partially occlude the eyes; as a result, the blink response can not be recorded. Although the environment in which the videos are recorded must be well-lit, the high-speed camera captures the light flickering, which causes illumination differences among subsequent frames in the video. Furthermore, a subject must sit close enough to the camera for their face to cover at least 20% of the image. Finally, at this moment the camera records at 333 FPS, but the desire has been expressed to increase this to 500 FPS. Therefore, the required frame rate shall be set to 500 FPS which means that the maximum processing time for each frame is 2 ms.

## 4.2 Face-detection

Face detection has been one of the most studied subjects in computer vision for the last 15 years and can therefore be considered as a mature and distinct field from the generic object-detection field. Face detection is an essential early step for various tasks

such as face recognition, facial-attribute classification, face editing, and face tracking, and its performance has a direct impact on the effectiveness of those tasks.

Early face-detection efforts, followed the classical approach. They were mainly based on hand-crafted features which are extracted from an image and then fed into a classifier to detect likely face regions. Two well-known classical face detectors are the Haar-cascade face detector by Viola and Jones [12] and the Histogram of Oriented Gradients (HOG) followed by SVM (classifier) [11]. While these works represent great improvements on the state-of-the art at their time, face-detection accuracy was still limited on challenging images with multiple variation factors like scale, occlusion, pose, expression, illumination and more.

Newer models use Deep Neural Networks (DNNs) or Convolutional Neural Networks (CNNs), which are a type of Neural Networks specialized in image-recognition tasks. Convolutional neural networks have achieved remarkable performance in a variety of computer vision tasks, such as image classification [20] and face recognition [21], [22]. Inspired by the good performance of CNNs in computer vision tasks, many CNN-based face detection methods have been proposed in recent years [23].

Li et al. [69] proposed one of the early deep models for face detection, based on a convolutional neural network cascade. The proposed CNN cascade operates at multiple resolutions, quickly rejects the background regions in the fast low resolution stages, and carefully evaluates a small number of candidates in the last high resolution stage. To improve localization effectiveness, and reduce the number of candidates at later stages, they introduce a CNN-based calibration stage after each of the detection stages in the cascade. The proposed method achieves 14 FPS on a single CPU core for VGA-resolution images and 100 FPS using a GPU. Zhang et al. [34] extended this approach by introducing the Multi-Task Cascaded Convolutional Neural Network or MTCNN which is considered one of the most popular face detection tools. This work leverages a cascaded architecture with three stages of carefully designed deep convolutional networks to detect and locate the face and five facial landmarks in a coarse-to-fine manner.

Two prominent approaches that most recent DNN face-detection models follow are the single-stage approach and two-stage approach:

- **Single-stage models**, such as S3FD[24], RetinaFace[25] and BlazeFace[10], refer broadly to architectures that use a single feed-forward full convolutional neural network [13] to directly predict each proposal's class and corresponding bounding box without requiring a second stage per-proposal classification operation and box refinement. Fig. 4.1(a) exhibits the basic architecture of one-stage detectors.

- **Two-stage face detection models** are mostly deriving from the work of Regional CNN (R-CNN) [26]. Initially they use, as a first step, an algorithm (e.g. Selective Search [27]) or a Region proposal-based CNN model that accepts the image

as input and proposes different possible Regions of Interest (RoI). Then a second model (a CNN feature extractor) computes features from these proposals to infer the bounding box coordinates and class of the object. Examples of such models are CMS-RCNN[28], R-FCN[29] and "Face Detection Using Improved Faster RCNN" [30]. Fig. 4.1(b) shows the basic architecture of two-stage detectors.

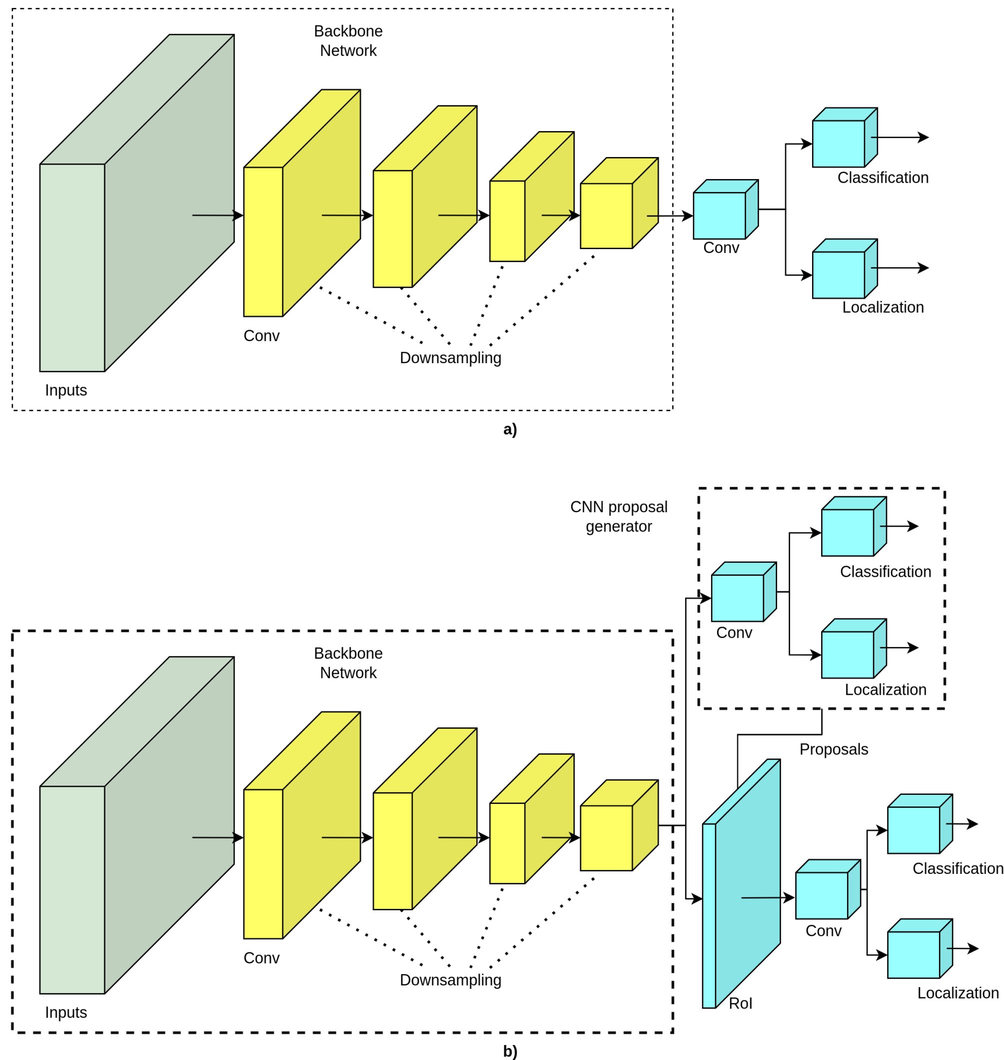



**Figure 4.1.** *(a) shows the basic architecture of one-stage detectors, which predicts bounding boxes from input images directly. (b) exhibits the basic architecture of two-stage detectors, which consists of the region proposal network to feed region proposals into the classifier and regressor. The Backbone network is acting as the basic feature extractor for the face detection task which takes images as input and outputs the feature maps of the corresponding input image.*

Two-stage detectors have high localization and object recognition accuracy, whereas the one-stage detectors achieve high inference speed as they propose predicted boxes from input images directly without the region proposal step. Since our project aims for both accuracy and speed, we decided to not investigate two-stage detectors for this project as they are not designed for real-time applications. In contrast, the single-stage detectors

provide a good tradeoff between the accuracy and efficiency and therefore are more suitable for real-time applications [31] [32].

There are several other DNN architecture approaches that have been proposed and used by the computer vision community to tackle the face detection task which are not covered in this thesis. For a more detailed survey, the reader can refer to the extensive survey of [23] and the many papers that survey points to.

### 4.2.1 Selected Algorithms

It is important to ensure that the selected face-detection algorithms satisfy the requirements described in Section 4.1. Open access and availability are also two important factors that we considered for the algorithm selection. Training an efficient face detection model is a very time-consuming and non-trivial task [23], and because there are pre-trained face detector implementations available online, we shall use one of them and adjust the model to meet the requirements of the project. Therefore, from the face-detection architectures described in Section 4.2, we selected three algorithms which we are going to test regarding accuracy and speed.

#### HOG

The first selected algorithm is the Histogram of Oriented Gradients (HOG) algorithm, which is implemented using the Dlib library [33]. Dlib is a machine learning open-source library which has been developed since 2002, and offers a pre-trained implementation of a HOG face detector. The HOG algorithm implementation in Dlib is work of Dalal et al. [11] combined with downscaling pyramid features from the work of Felzenszwalb et al. [70]. The HOG detector is extracting features from given input images, which are fed into a L-SVM based sliding window classifier, to classify an area of 80x80 pixels as a face based on its trained models. The face detector from Dlib uses five classifiers trained on 3000 images of the Labeled Faces in the Wild (LFW) face dataset [71].

Even though it an early face-detection effort, the HOG algorithm is still widely used (39313 citations) on the field of object and/or face detection. Previous work [37] has successfully used a GPU-accelerated version of the HOG-algorithm-based face detector in combination with a landmark detection model to detect the blinking of an eye in video footage in real time.

#### Multi-Task Cascaded Convolutional Neural Networks

The second algorithm is called Multi-Task Cascaded Convolutional Neural Networks or MTCNN, a CNN based method that was published in 2016 by Zhang et al.[34]. The MTCNN model leverages a cascaded architecture with three stages of CNNs to predict face

and five facial landmarks in a coarse-to-fine manner. More specifically, these three stages are called: P-Net, R-Net and O-Net (Fig. 4.2).

Before the image is passed into the network an image pyramid is created, in order to detect different sized faces within the image. In the P-Net stage, the algorithm produces a candidate window sliding quickly through a small subset of CNNs. The second stage contains the R-Net which refines the windows with the purpose of rejecting the large number of non-face windows present in the reference image with additional CNNs. Finally in the O-Net stage, deeper CNN layers are used to refine the result of the detected facial features in the image and further locate five facial landmark points. Non-Maximum Suppression (NMS) [16] is used to filter out bounding boxes based on their confidence score and the degree of overlap between them.

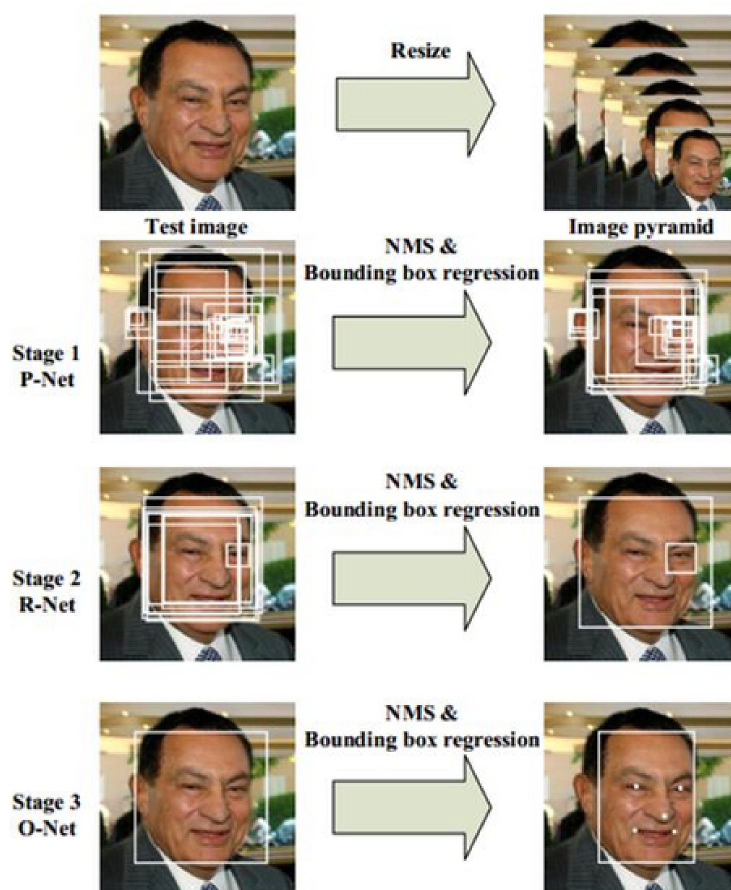



**Figure 4.2.** *MTCNN - stage based iteration approach for face detection. Image from [34].*

The MTCNN face-detector is considered a very popular early CNN-based face detection effort with good accuracy results. In addition, a pre-trained implementation is publicly available through the Python Package Index (PyPI) repository of software for the Python programming language [72].

#### BlazeFace

The third and last selected algorithm is called BlazeFace [10], a lightweight and well-performing face detector developed by Google in the year 2020 for mobile GPU inference and real-world applications. It takes as input RGB images (possibly a video frame) resized to 128x128 pixels, represented as a $128 \times 128 \times 3$ array of single precision float values in the range [-1.0 , 1.0]. The output for each detected face, is four bounding box coordinates, six approximate facial keypoints coordinates (left and right eye-center points, nose tip, mouth, left eye tragion , right eye tragion) and a detection confidence score.

In the recent work of [73], three face-detection models are selected and evaluated in terms of speed for real-time applications . The widely-used HOG face-detection algorithm is compared against two deep learning models, ResNet [74] and BlazeFace [10]. BlazeFace was the fastest despite being a deep convolution model. The HOG algorithm came second and the ResNet model was the slowest (deeper network). The authors conclude that BlazeFace is a very good choice for real-world applications as apart from high inference speed its accuracy results were also found to be quite satisfactory. Finally, it is important to note that Google is providing a pre-trained implementation of BlazeFace through the Mediapipe library [43].

## 4.3 Face-detection algorithm comparison

In order to determine which of the previously described algorithms is best suited for our project, we need to evaluate them in terms of accuracy and speed. For that purpose, we use a large collection of images of faces with annotated locations.

### 4.3.1 Dataset selection

The "Annotated Facial Landmarks in the Wild" (AFLW) dataset [35] is one of the selected image collections for our evaluation phase. The BioID database [36] is another selected database on which we are going to test our algorithms.

#### AFLW database

The AFLW database contains 21123 images with 24384 annotated faces. These images are taken in real-life situations which means that the lighting, position, size and rotation of the faces, background and occlusion are all uncontrolled. The resolution of the images in the database varies. In addition to the location of the faces, annotations for 21 landmarks (when visible), sex, occlusion, glasses, use of color, and three rotation angles (roll, pitch and yaw) are also provided. This extensive annotation makes it easier to test on a subset of images that meets our project's requirements.

#### AFLW subset

From the AFLW database, a subset of faces can be selected that is within our application's expected regions of rotation (yaw $\pm 20°$, roll$\pm 25°$, pitch$\pm 40°$). Faces with and without glasses should both be detected. Lighting conditions are not annotated and can therefore not be filtered. Occluded faces are not filtered out because it is not clear in what way or to what extent they are occluded. Because some images contain multiple faces, it can occur that one or more face(s) in an image meet the requirements, while others don't. In this case, the image is initially left in the subset and we exclude a face from the subset only when it doesn't meet the project requirements.

#### BioID database

The BioID database, consists of 1521 grayscale images ($384 \times 288$ pixel, grayscale) of 23 different persons. The conditions on which the images were recorded strongly resembles our project's conditions and therefore testing a face detector on this database will provide a strong indication of how well will this method perform on our test-set.

#### Pre-processing input data

While pre-processing of the input data is not required for the HOG and MTCNN methods, since they can operate on any input image size, that is not the case for the BlazeFace model which accepts a fixed image size of $128 \times 128 \times 3$ pixels. Therefore, we adjusted the size of the input data to $128 \times 128$ pixels for the evaluation of the three detectors. Resizing the input images affects the resolution. Recent work [75] concerns the effects of image quality in detection tasks. Resizing images for object-detection tasks, in our case human faces, can affect radically the output of the detector in terms of quality. To this end we add an extra requirement, for selecting a subset of AFLW dataset, where images above 900x900 pixels are excluded from the dataset.

### 4.3.2 Performance on constrained AFLW subset

For a fair performance comparison, we run each algorithm on a single CPU core. Accuracy is measured using recall and precision, where recall describes how many of the existing faces were indeed detected, while precision describes how many of the detected faces were indeed real faces and not false positives. Recall and precision are calculated as follows.

$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

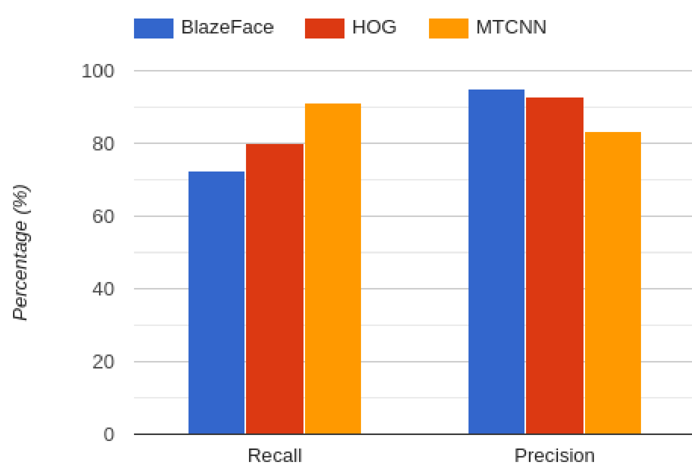
$$TP = true\ positives,\ FN = false\ negatives,\ FP = false\ positives$$

A true positive stands for a correct detection of a face that is annotated in the database, a false positive detection is when the face detector incorrectly detects an image region as

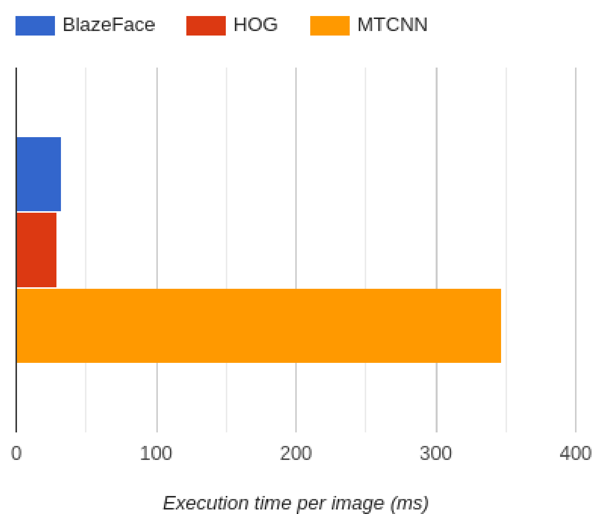
a face and a false negative is a failure to detect an annotated face.The results on AFLW subset of 3425 faces can been seen in Table 4.1 and visualized in Fig. 4.3 a) and Fig. 4.4 b).

| Algorithm | Faces | FN | FP | TP | Precision (%) | Recall (%) | Time / Image (ms) |
|---|---|---|---|---|---|---|---|
| HOG | 3425 | 687 | 211 | 2738 | 92.84 | 79.94 | **29.23** |
| MTCNN | 3425 | **294** | 631 | **3131** | 83.17 | **91.41** | 347.41 |
| BlazeFace | 3425 | 940 | **126** | 2485 | **95.17** | 72.56 | 32.73 |

**Table 4.1.** *Results on face detection algorithms on subset of AFLW that meets project requirements*




**(a)**




**(b)**

**Figure 4.3.** *a) Precision-Recall results of BlazeFace, HOG and MTCNN face detectors on the AFLW database subset. b) Execution time results of BlazeFace, HOG and MTCNN face detectors on the AFLW database subset.*

**False Negatives**

By analyzing the resutls (Table 4.1), we observe that the HOG failed to identify 687 faces (FN). Many of the faces seem to be partially occluded by a variety of objects, such as hair, glasses, hands etc. Furthermore rotated faces, bad lighting and image blurriness are other main causes of false negatives. MTCNN and BlazeFace detectors also suffer from the same problems mentioned above. In addition one of the most common causes of false negatives of the CNN remains a too small face size.

BlazeFace produced the most false negatives, 940 faces in total, which leads to lower recall values. As we described before, BlazeFace has some extra requirements in order to perform well and accurately. The face must occupy at least 20% of the image size, so small faces are always a cause of false negatives. Another requirement is that the face inside the image must be at most 2 meters away from the camera. Even though we have a constrained subset of the AFLW dataset, to meet our project requirements, we don't have any information about the distance of each face from the camera. After manual inspection of the false negatives we confirm that BlazeFace was unable to detect small faces and faces that are far away from the camera. For this project however the faces in the video will fill up at least 20% of the image, which has a resolution of $640 \times 480$, and the subject is under 1 meter away from the camera. So both of these extra requirements are covered.

We strongly believe that even though the recall value of BlazeFace is low, it will perform much better on our test-set. We think it remains a good candicate for our project since it is quite fast (lightweight DNN model architecture) and specifically developed for real-time applications. In order to evaluate this hypothesis we are going to further evaluate BlazeFace on the BioID dataset that strongly resembles our test-set.

**False Positives**

The amount of false positives of each method is an important factor that we have to take under consideration. It is important for our project that the face detector does not generate too many false positives. We define a false positive as a as a face that is detected by the face detector, but is not annotated in the AFLW database.

After manual inspection of the predictions, it turns out that the AFLW database contains annotation mistakes. We found that from the 211 false positives that the HOG face detector produced only 7 were valid (the detector classified a bounding box region as a face while there is not face), and the rest do in fact have a face inside which in not annotated in the AFLW database. This means the HOG face detector actually produces much less false positives than the previous tests suggest. The same applies for MTCNN detector, where out of the 631 false positives only 37 were valid. Finally, BlazeFace detector has only 1 valid false positive and the rest 126 detections do in fact have a face inside. The re-evaluated results can be seen in Table 4.2.

| Algorithm | Faces | $FP_{initial}$ | $FP_{re-evaluated}$ | TP | Precision (%) |
|---|---|---|---|---|---|
| HOG | 3425 | 211 | 7 | 2738 | 99.74 |
| MTCNN | 3425 | 631 | 37 | **3131** | 98.83 |
| BlazeFace | 3425 | 126 | **1** | 2485 | **99.96** |

**Table 4.2.** *Re-evaluated results on face-detection algorithms on subset of AFLW because of annotation errors. $FP_{initial}$ indicates the number of false positives according to AFLW database annotations, while $FP_{re-evaluated}$ indicates the remaining valid false positives after manual inspection*

The reduced number of valid false positives indicates that all three methods have a higher precision (last column in Table 4.2) than suggested in previous tests on the AFLW database (last column in Table 4.1). The BlazeFace model achieved the highest precision of 99.96%.

### 4.3.3 Perfomance on BioID database

The BioID database, contains 1521 images that strongly resembles our test-set and every image depicts a single face. BioID database provides annotations for five facial landmarks but annotations for the groundtruth face boxes is not provided; therefore we evaluated BlazeFace detector by using the relative-distance error metric described in [36]. This distance error is computed as the euclidean distance of the center of each detected eye-landmark that BlazeFace detects with the manually annotated groundtruth eye-center point, which is provided from the dataset, divided by the inter-ocular distance of the eye for scale invariance. We calculate the distance error of both eyes, keep the maximum value and if it is less or equal to 0.25 then we rate the face as a true positive.

| Algorithm | Faces | Right average error | Left average error | TP | FN |
|---|---|---|---|---|---|
| BlazeFace | 1521 | 0.069% | 0.073% | 1497 | 24 |
| MTCNN | 1521 | 0.03% | 0.043% | 1514 | 10 |

**Table 4.3.** *Results of BlazeFace on BioId database*

BlazeFace was able to detect correctly 1497 faces over the total of 1521 faces (Table 4.3) inside the BioID database by using the relative-distance error metric. However, after manual inspection of the detections, we found that for every image in the dataset there was only one detection box and each detection box had a face inside. Thus, the BlazeFace model has 100% success on detecting faces on the BioID database. A similar procedure was followed to evaluate the MTCNN detector, which also produced 100% correct predictions.

For the evaluation of the HOG detector it was necessary to generate labels describing the position of the face within the image (bounding boxes) as this detector is not able to detect facial landmarks. For this purpose the generated MTCNN model predictions were used as a basis for comparison with the output predictions of the HOG detector. Similarly,

the HOG algorithm also managed to correctly detect all faces in the BioID dataset.

The results of the above evaluation are a strong indication that all three face detectors will perform well in our test set.

### 4.3.4 Conclusion on face-detection testing

The above tested detectors have differences in both speed and accuracy. From our experiments we found that MTCNN is the most accurate face detector in the AFLW dataset subset, as it has the highest number of True Positives (3131) and achieves a Recall of 91.41%. It is a reliable option since it produced the lowest number of false negatives (294), and was even able to detect many faces that were not annotated in the AFLW database. However, it is much slower compared to the other two methods, and since speed is an essential factor, the MTCNN model shall not be used as face detector for this project.

Both HOG and BlazeFace are much faster alternatives, with HOG being the fastest method. BlazeFace is quite fast despite the fact that it's a Deep Neural Network, achieving the highest precision and the lowest amount of false positives. On the downside, BlazeFace won't operate well on very high-resolution images, since we have to resize them down to 128 x 128 pixels and lose valuable information from the image which leads to poor performance, for faces that are far away from the camera. This explains the poor recall and the high amount of false negatives on the AFLW database subset. However, after testing BlazeFace on the BioID dataset, where the recording environment and setting are quite similar with our test-set, the model achieved 100% correct detections; a strong indication that BlazeFace will behave similarly on our dataset. Furthermore, an additional goal of our work is to explore detector implementations from the field of deep learning, as the IPU is a hardware chip specialized in machine learning and offers a great acceleration potential by running a deep neural network on top of it though the TensorFlow and Keras API.

For the above-mentioned reasons, the **BlazeFace algorithm** shall be used for face detection in this project.

## 4.4 Eyelid-closure detection

Once the face detection is finished and the location of the face has been obtained, the next step is to detect the eyelid closure. The eye blink is a fast closing and reopening of a human eye. Each individual has a somewhat different pattern of blinks. The pattern differs in the speed of closing and opening, a degree of squeezing the eye and in a blink duration. The eye blink lasts approximately 100-400 ms.

To capture the eye-blinks we shall follow the landmark detection approach, where the

landmarks on the eyelids are used for eyelid-closure detection. On each eye, six landmarks are located: two in the corners, two on the upper eyelid and two on the lower eyelid; see Fig. 4.4. If these are accurate enough, they could be used directly for the calculation of eyelid closure. These landmarks are already successfully used for blink detection by computing a metric called eye aspect ratio (EAR), introduced by Soukupová and Čech [38].

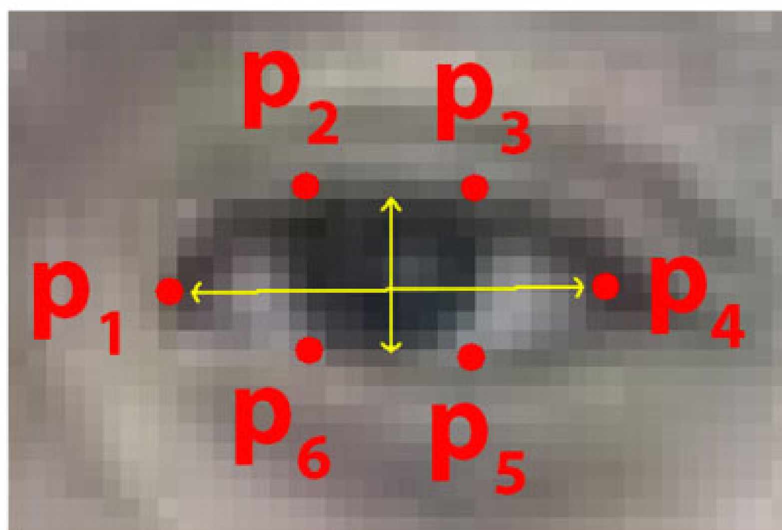



**Figure 4.4.** *The 6 facial landmarks associated with the eye. Image from [38]*

**Eye Aspect Ratio (EAR)**

From the landmarks detected in the image, we derive the eye aspect ratio (EAR) that is used as an estimate of the eye opening state. The numerator of this equation 4.1 computes the distance between the vertical eye landmarks while the denominator computes the distance between horizontal eye landmarks, weighting the denominator appropriately since there is only one set of horizontal points but two sets of vertical points. Using this simple equation, we can avoid complex image processing techniques and simply rely on the ratio of eye landmark distances to determine if a person is blinking.

$$EAR = \frac{||p2 - p6|| + ||p3 - p5||}{2 * ||p1 - p4||} \tag{4.1}$$

where p1, ... , p6 are the six 2D landmark locations of the eye.

The EAR is mostly constant when an eye is open and is getting close to zero while closing an eye. It is partially person- and head-pose insensitive. The aspect ratio of the open eye has a small variance among individuals and it is fully invariant to a uniform scaling of the image and in-plane rotation of the face.

### 4.4.1 Landmark detection

Detecting facial landmarks is a subset of the shape-prediction problem. Given an input image, a shape predictor attempts to localize key points of interest along the shape. Note that most facial landmark detection algorithms require the face to be first detected. Therefore, the facial landmark-detection task is a two-step process: localize the face in the image and detect the key facial structures on the face.

Given the face region, (x,y) coordinates of the face bounding box, we can then apply the detection of key facial structures in the face region. There are many facial regions that a landmark detector can localize such as the mouth, eyebrows, eyes, nose and jaw. Since our project goal is to detect the eyelid closure, we are interested mainly in the landmarks of the eye regions. Examples of landmark detection on two faces are presented in Fig. 4.5.

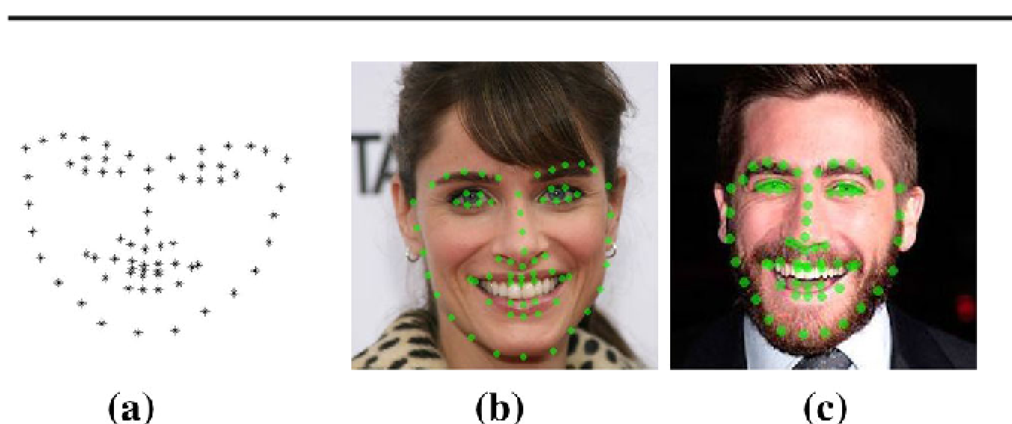



**Figure 4.5.** *Example of 68-landmark detection on two faces. Image from [39].*

In [42], 48 methods of face alignment are compared in terms of speed and accuracy on five databases. Most of the methods discussed in the survey focus on the detection of 68 landmarks while the database on which the most face alignment methods are evaluated, is the HELEN database [76]. Out of the five databases, HELEN contains images with the most similar conditions to the conditions in our project. The evaluation error metric is measured as the Euclidean distance of each detected landmark to the true (annotated) landmark, divided by the inter-ocular distance for scale invariance. According to [42], the ERT algorithm performs second-best in speed and fifth-best in accuracy. The "Ensemble of Regression Trees" (ERT) algorithm [40] is considered one of the most popular and state-of-the-art methods for landmark detection/face alignment. This method belongs to the class of cascaded regressors, which refines its estimates of the landmark locations in a number of consecutive stages. The recent extensive survey of [41] also considers the ERT algorithm as a reliable and fast landmark-detection model (according to the authors, around 1 millisecond per face processing time). An ERT implementation is available through the open-source machine-learning Dlib library [33] which is still actively used in the modern research thanks to the open implementation and speed.

Furthermore, the work of [41] analyzes facial landmark-detection models from the field of deep-learning. The authors discuss the Practical Facial Landmark Detector (PFLD) [77]. PFLD follows a direct approach that enables fast facial landmark detection directly on a mobile device. It is considered the only modern neural-network-based algorithm, whose authors have shown it to work efficiently on a mobile device. In particular, the PFLD 0.25X version uses a tweaked version of MobileNetV2 [78] as feature extractor and achieves an inference speed of 1.2 miliseconds per frame.

### 4.4.2 Selected algorithm

It is important to ensure that the selected landmark detector fits our needs in terms of speed and accuracy, as they have been described in Section 4.1. The PFLD 0.25X was considered for this project but training efficiently a landmark-detection network from scratch is a very time-consuming and non-trivial task. The actively-used ERT algorithm is a suitable choice for this project as its high speed (1 ms per image) satisfies our 2 ms maximum processing-time window for end-to-end eyelid-closure detection. Furthermore, previous work [37] has introduced a multithreaded version of ERT landmark localizer, that is used to calculate eyelid closure based on the Eye Aspect Ratio (EAR) in real-time.

Apart from high speed, the selected algorithm must be able to provide accurate landmark detections, especially for the detected facial landmarks around the eyes. Therefore, we shall use once again the BioID database to further review the accuracy of Dlib's ERT implementation, since it's a very similar dataset to our test-set. The error was calculated in the same way as in the above described survey [42]. From the 68 landmarks that the algorithm predicts, we make use of the six detected landmarks on each eye to calculate the center of each eye. We then compare the predicted centers with the annotated locations. We calculate the error for each eye separately and the average run-time per image. The results are provided in Table 4.4.

| **Landmark Detector** | **Right-Eye error** | **Left-Eye error** | **Time per image** |
|---|---|---|---|
| Dlib's 68-Landmark Detector | 2.61 % | 3.64 % | 1.06 (ms) |

**Table 4.4.** *The evaluation error metric is measured as the euclidean distance of each detected landmark to the true (annotated) landmark, divided by the inter-ocular distance for scale invariance [42].*

The error and the average run-time (1.06 ms, around 1000 FPS) are considered satisfactory. The Ensemble of Regression Trees (ERT) [40] provides a quite fast and accurate way to detect landmarks and a pre-trained implementation is available through the open-source Dlib library. For the above mentioned reasons the **Ensemble of Regression Trees (ERT)** algorithm shall be used for eyelid closure detection in this thesis.

## 4.5 Details of the selected algorithms

In this section the selected algorithms of Section 4.3 and Section 4.4 will be discussed in detail. Subsection 4.5.1 will discuss the BlazeFace face detector, while Subsection 4.5.2 will cover the ERT landmark detector.

### 4.5.1 Face detector

BlazeFace [10] is a lightweight and well-performing CNN face detection model tailored for real-time world applications. The model has been introduced by Google and follows the Single-Shot MultiBox Detector (SSD) approach which is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances (in our case face or no-face) in those boxes, followed by a non-maximum suppression step to produce the final detections.

#### BlazeFace breakdown analysis

The early network is a feature extractor based on MobileNet V1/V2 [79] [78] which we will refer to as the backbone network. This network outputs a rich feature representation of the input image as a collection of stacked feature maps. Then a set of auxiliary convolutional feature layers are added to the network. These layers produce multiple feature maps of different sizes, in successive order, which are stacked after the feature maps coming from the backbone. While a typical SSD model uses feature maps of sizes $1 \times 1$, $2 \times 2$, $4 \times 4$, $8 \times 8$, and $16 \times 16$, BlazeFace uses two feature layers of size $16 \times 16$ and $8 \times 8$ without further downsampling (Fig.4.6 ).

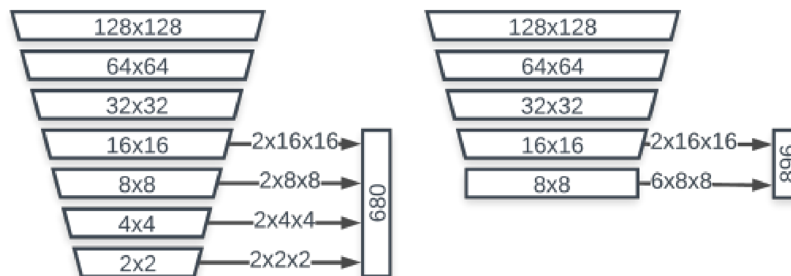



**Figure 4.6.** *Anchor Sceme: SSD (left) vs. BlazeFace (right)*

Each of the convolution feature layers is connected to two heads, one regressor to predict bounding boxes and one sigmoid classifier that produces a score for a category (face or no-face). Fig. 4.7 shows the BlazeFace architecture that consist of backbone feature extractor and auxiliary predictor layers. Each layer predicts some bounding boxes for

the class human face. Eventually predictions from all the layers are accumulated in the post-processing layer which provides the final prediction for the face as the output.

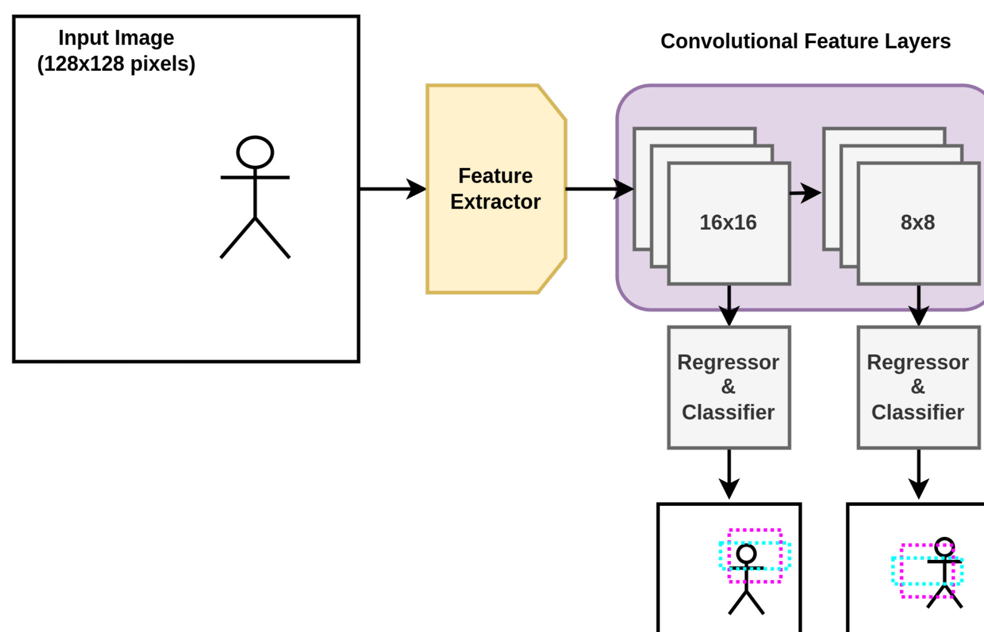



**Figure 4.7.** *BlazeFace architecture that consists of base feature extractor and auxiliary predictor layers.*

Similar to SSD object-detection models, BlazeFace also relies on a set of pre-defined fixed-size bounding boxes to position objects. These boxes are called anchors or priors. Each feature map is divided into number of grids and each grid is associated to a set of priors or default bounding boxes of different dimensions and aspect ratios. In total, there are 896 anchors, and each of the two detection layers is associated with a specific number of scale anchors. The aspect ratio of the anchors is fixed at 1:1 (i.e., square anchor), since the variance in human face aspect ratios is limited. As shown in Fig. 4.8 each grid cell of a $8 \times 8$ feature map have six anchors and each of the anchors predict exactly one bounding box. Therefore, we have $64_{grid-cells} \times 6_{anchors} = 384$ detections. Similarly, every grid cell of a $16 \times 16$ feature map is associated with two anchors, and therefore we have $256_{grid-cells} \times 2_{anchors} = 512$ detections.

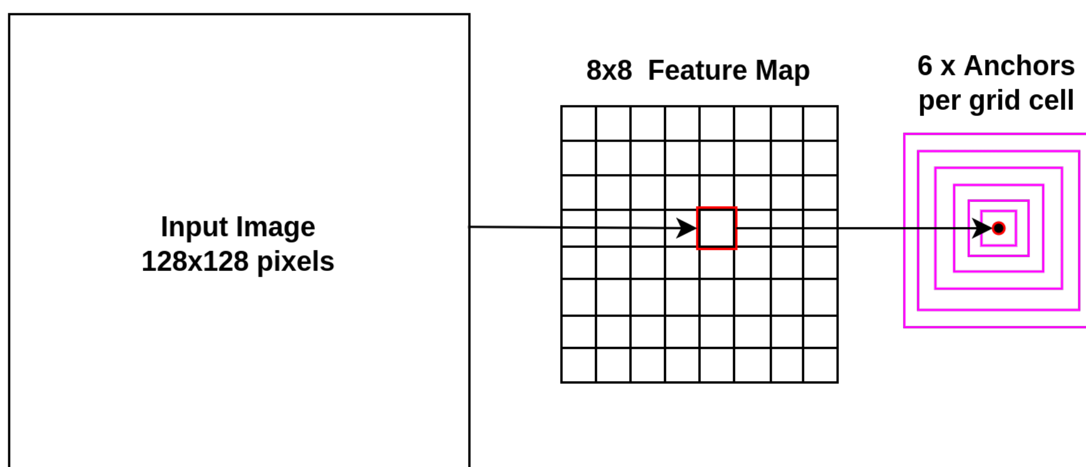



**Figure 4.8.** *Anchor boxes on a $8 \times 8$ feature map*

Feature maps represent the dominant features of the images at different scales, so having anchors on multi-scale feature maps increases the likelihood of human faces of different sizes to be accurately localized and appropriately classified. The $8 \times 8$ feature map anchors provide more accurate predictions for larger face sizes while anchors in the $16 \times 16$ feature map for smaller ones.

Since BlazeFace is not reducing the resolution below $8 \times 8$, the number of overlapping anchors for a detected object is very large. The typical approach, is to make use of Non-Maximum Suppression (NMS) [16], where only one of the several anchors that contain the same face is selected as the final prediction. However, when NMS is used to subsequent video frames a temporal jitter problem occurs because the predictions fluctuate greatly between different anchors. To solve this, BlazeFace is using a blending strategy where instead of selecting only one box, it constructs the final bounding box by calculating a weighted mean between overlapping predictions. While this blending strategy incurs no additional cost in the calculation, the authors of [10] reported an improve of the detection accuracy by 10%.

### 4.5.2 Landmark detector

For landmark detection/localization we used the Ensemble of Regression Trees (ERT), a cascade-regression-shaped predictor implementation found in DLib [33], which is based on the work of Kazemi et al. [40]. The detector is trained on the iBUG 300-W dataset [80],[81].

Landmark localization is the process of localizing the shape of specific landmarks on an object. In the Eye-blink Conditioning (EBC) case study, it is used to estimate the location of 68 (x, y)-coordinates that map to facial structures on the face. The indexes of the 68 coordinates can be visualized on figure 4.9.

These annotations are part of the 68 point iBUG 300-W dataset. It's important to note here that other flavors of facial landmark detectors exist, including the 194 point model that can be trained on the HELEN dataset. Regardless of which dataset is used, the same dlib framework can be leveraged to train a shape predictor on the input training data, this is useful if we would like to train facial landmark detectors or custom shape predictors of our own (we do train our own shape predictor for just detecting eye landmarks).

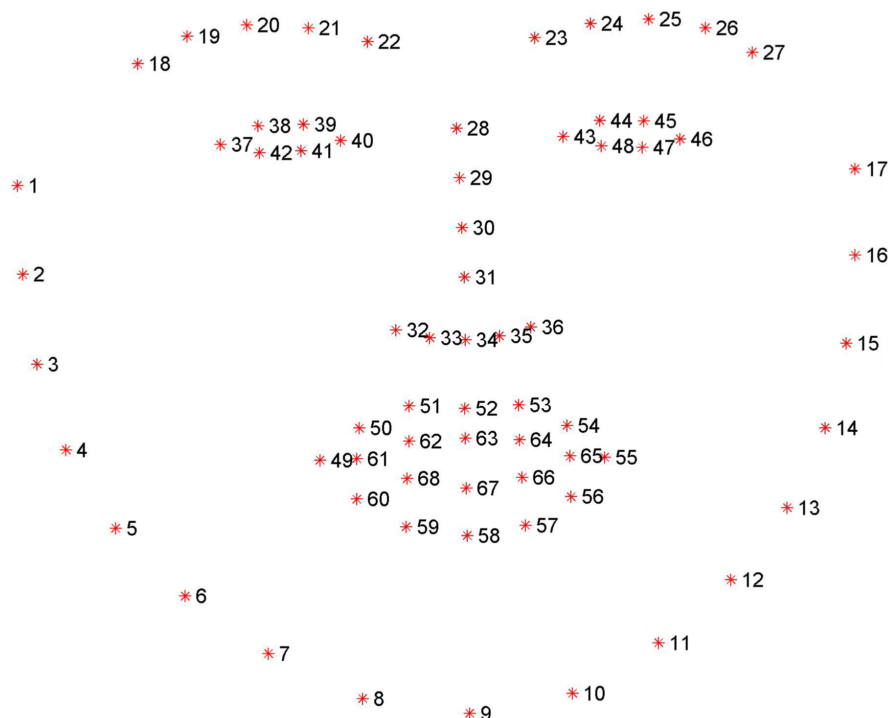



**Figure 4.9.** *Visualizing the 68 facial landmark coordinates from the iBUG 300-W dataset*

#### ERT breakdown analysis

The process of landmark localization starts by placing an initial shape estimate over the center of the faces detected by the face-detector (BlazeFace). This estimation is based on the mean shape of all landmark configurations of images it has been trained with. Then a forest of regression trees gradually calculates a shift of the landmark positions towards the actual facial features based on calculated features and pixel intensity. This is an iterative process, which is also called cascaded regression approach, where feature calculation and landmark shifting using regression trees is happening for each lever of the cascade. The results of all the regression trees in a level are added to the current landmark estimation, which results in the landmark-shape estimate for the next level.

The iterative process can be summarized in the following steps.

- **Initialization:** Initialize the landmark shape estimate. This is the pre-trained mean of all landmark shapes in the training set
- **Feature computation:** Calculate the similarity transform between the estimation of the shape at the current cascade level and the original mean-shape estimation.

The transformation is used to calculate the new location of the feature points for the regression trees

- **Regression-tree estimation:** Traverse each regression tree in the forest based on pixel intensity differences and add their results to the current landmark shape estimate
- **Repeat:** Repeat the feature computation and regression tree estimation step for each level of the cascade

The default ERT model of Dlib consists of 500 regression trees used by each of the 15 levels of the cascade. The depth of each regression tree is 5 layers leading to 16 possible outputs, also known as leaves. In each level of the tree, either the right or the left child is chosen, based on the intensity difference of two pixels. The critical point of this approach is that the location of these pixels is indexed relative to the landmark-estimation shape of the current cascade level. At each level of the cascade, the new locations of the required pixels for the decision-splits of all 500 regression trees are calculated. The feature points are indexed relative to the initial mean shape and undergo the same transformation to calculate their position relative to the current shape estimate.

Chapter 5

# Implementation

In this chapter, we will discuss the implementation details of the eye-blink detection pipeline. This project can be divided in five distinct steps and can be visualized in Fig. 5.1.

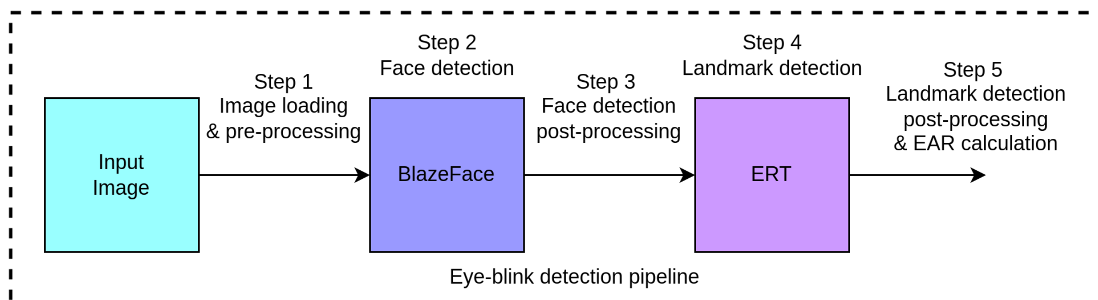



**Figure 5.1.** *Steps of the eyeblink detection pipeline.*

Implementation details and a number of applied optimization techniques for every step of the above pipeline will be analyzed separately. Our goal is to achieve minimum latency (the eyeblink detection pipeline must be able to run end-to-end in under 2ms or 500FPS) and maximum throughput. Section 5.1 will cover a combined multi-core CPU implementation of the image loading and pre-processing steps. In Section 5.2, we will describe two different implementation approaches for the face-detection step. A GPU-based system and an IPU-based system will be used to accelerate the BlazeFace model and achieve high speed inference. In Section 5.3 the face-detection post-processing step is analyzed. Section 5.4 will discuss a multi-core CPU implementation of the original 68-landmark-detection algorithm (ERT). In addition, a 12-landmark-detection (eyes-only) version of the ERT algorithm will be trained from scratch and will be tested for this project. Finally, in Section 5.5 a combined implementation of the landmark-detection post-processing and EAR calculation steps will be discussed.

## 5.1 Image Loading & Pre-processing

In the current eyeblink conditioning set-up, videos are recorded at a resolution of $640 \times 480$ pixels. All images are grayscale which means that every loaded image has a shape of 640x480x1 pixels. BlazeFace expects a square RGB input image of $128 \times 128$ pixels. To keep the original aspect ratio of our images unchanged, we initially padded each image with zeros (distributing the zeros equally on the left and right sides of each image) to make it square and then resized it to the desired size. Furthermore, we converted each single-channel gray image into a three-channel gray image using channel replication to match the desired format.

We make use of Python Imaging Library (PIL) [82] for loading and pre-processing our input images. PIL is a free and open-source library of the Python programming language that adds support for opening, manipulating, and saving many different image file formats. The library supports basic image processing functionality (such as image resizing), including point operations, filtering with a set of built-in convolution kernels, and colour space conversions (i.e. from BGR to RGB).

Loading and pre-processing an image are time-consuming tasks which can take up to 2.7 (ms) per image. To accelerate these tasks, we combine the PIL library with the Python multiprocessing library which allows to leverage multiple CPU processors for parallel execution. The API used is similar to the classic threading module, however the multiprocesing module avoids the limitations of the Global Interpreter Lock (GIL) by using subprocesses instead of threads. A global interpreter lock (GIL) is a mechanism used in Python interpreter to synchronize the execution of threads so that only one native thread can execute at a time, even if run on a multi-core processor. True parallelism in Python is achieved by creating multiple processes, each having a Python interpreter with its own separate GIL.

A convenient approach for parallel-processing tasks is provided by the Pool class which represents a pool of worker processes, where each process has separate memory location (i.e distributed memory) and runs completely independent. More specifically we used pool's map method, a parallel equivalent of the python built-in map method which blocks the main execution until all computations finish. The map method chops the given iterable list of image paths into a number of chunks which submits to the process pool as separate tasks. Therefore, in our case each spawned process will handle the loading and pre-processing of its assigned chunk of images and then return the processed images back to the main process. Execution times for implementations using 1, 2, 4, 8, 16, 32 and 64 processes are shown in Table 5.1.

By running multiple processes over different input data sizes, we observe a relation between the amount of work that each process will handle and the obtained speedup. In cases where the input data size is small (i.e. N=640 images in total), the speedup factor

is increasing to a certain point and then is getting lower as we add more processes to the pool. This is because we are performing a lot of I/O operations where each PIL load image method call results in I/O overhead. Another reason is that multiprocessing is not a 'free' operation and there are overhead function calls, both at the Python level and operating system level, for spawning and destroying the processes. By increasing the amount of data to process (N=6400 and N=64000), we were able to achieve a speedup of 8.8 when making use of 64 processes.

| **Processes** | **Time per image (ms)** | | |
|---|---|---|---|
| | **N=640** | **N=6400** | **N=64000** |
| 1 | 2.679 | 2.688 | 2.64 |
| 2 | 1.375 | 1.384 | 1.397 |
| 4 | 0.728 | 0.698 | 0.703 |
| 8 | **0.589** | 0.433 | 0.418 |
| 16 | 0.676 | 0.347 | 0.334 |
| 32 | 0.765 | 0.309 | 0.303 |
| 64 | 0.845 | **0.307** | **0.262** |

**Table 5.1.** *Execution time of the image loading and pre-processing steps for a varying number of CPU processes. N is the number of input images*

## 5.2 Face Detection - BlazeFace

In this section, we will discuss the acceleration of the face-detection step on various hardware platforms. The TensorFlow and Keras APIs were the main tools we used for the construction and deployment of our pre-trained BlazeFace model, which are both supported on IPU and GPU hardware platforms. More specifically for the IPU implementation we used the 2.4 version of Poplar SDK with its IPU interface to TensorFlow 2.4.4. The same version of TensorFlow was used for the GPU implementation. First we will present the specifications of the used IPU and GPU chips and then we will discuss the design process and optimizations made to efficiently leverage their compute power efficiently.

Google is providing a pre-trained model of BlazeFace through Mediapipe library [43]. The model is stored as a .tflite file from which we were able to extract the weights and store them for future use. Next, we re-constructed the whole BlazeFace architecture from scratch, using TensorFlow and Keras API, in order to load back the extracted weights. This way we have the same pre-trained model ready for deployment (inference only) on IPU and GPU hardware through TensorFlow API and IPU Software Development Kit (SDK).

### 5.2.1 Hardware Specifications

To understand the choices made in the implementation phase we take a closer look at the architecture of the used IPU and GPU hardware chips. Specifications of each chip can be seen in Table 5.2.

| **Specifications** | **Nvidia V100** | **MK1 IPU** | **MK2 IPU** |
|---|---|---|---|
| Technology Node | TSMC 12 nm | TSMC 16 nm | TSMC 7 nm |
| Die Area (mm2) | 815 | 900 | 823 |
| Transistors (Bn) | 21.1 | 23.6 | 59.4 |
| Architecture | SIMD | MIMD | MIMD |
| Cores | 640 (Tensor-cores) | 1216 (IPU-cores) | 1472 (IPU-cores) |
| TeraFLOPS (FP16) | 125 | 125 | 250 |
| DRAM Capacity (GB) | 32 | N/A | 2x64 |
| DRAM BW (GB/sec) | 900 | N/A | 64 |
| SRAM Capacity (MB) | 36 (RF+L1+L2) | 300 | 900 |
| SRAM BW (TB/sec) | 224+14+3 (RF+L1+L2) | 45 | 47.5 |
| Max TDP (Watts) | 300 | 150 | 300 |

**Table 5.2.** *Details of chip-to-chip comparison between recent products from two leading ML hardware competitors: NVIDIA and GraphCore.*

### 5.2.2 Inference on MK1 & MK2 IPU chips

In order to accelerate the BlazeFace detector on an IPU chip efficiently, we had to consider various factors by which our model performance is affected. Features of the IPU hardware and the software used to develop play also a key role in the optimization stage. The Tensorflow ML framework and Keras API were used for the development and deployment of BlazeFace. In order to run the model on an IPU-based system we need to

construct an IPU-program. The implementation flow from a high-level framework, such as TensorFlow, to an IPU program is illustrated in Fig 5.2.

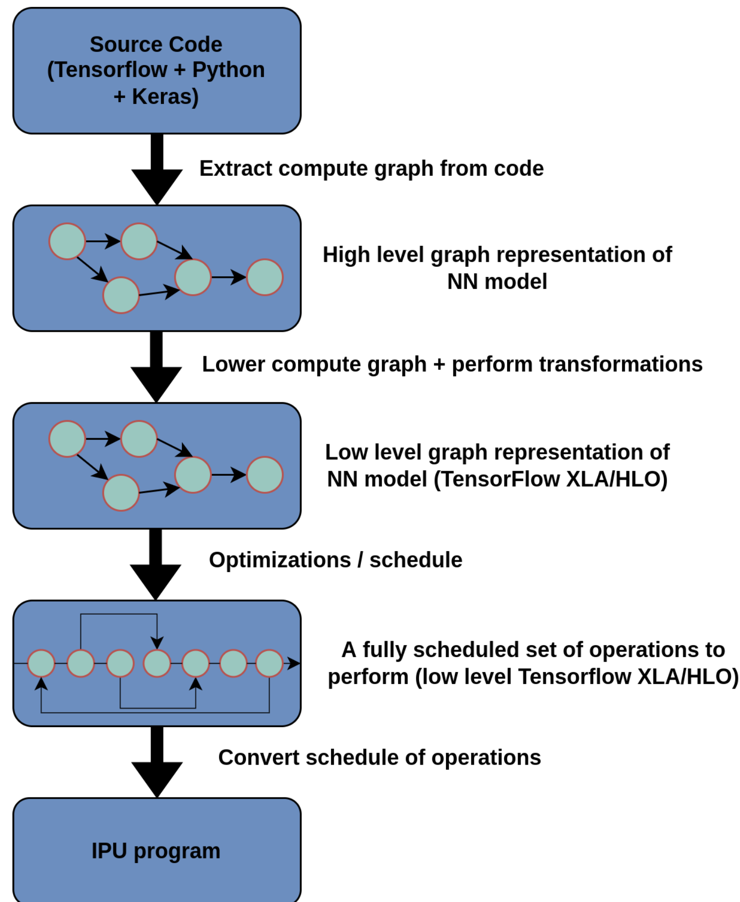



**Figure 5.2.** *A framework lowering to run on an IPU*

Tensorflow and Graphcore use a multi-step process to optimize the performance of machine learning models on IPU chips. The process can be broken down into the following steps:

1. **Extracting a computational graph:** Tensorflow extracts a computational graph from the source code (Python), which is a representation of the computations performed by the model.

2. **Analysis using XLA:** Tensorflow uses a specific-domain compiler called XLA (Accelerated Linear Algebra) to perform analysis passes over the high-level graph representation. XLA is used to optimize the graph for linear algebra computations.

3. **Custom XLA backend:** Graphcore uses a custom XLA backend to perform its own transformations and optimizations on the graph. The custom backend allows Graphcore to tailor the optimizations to the specific characteristics of IPU chips.

4. **Mapping to scheduled-graph:** The graph is then mapped to a low-level graph representation called scheduled-graph. This scheduled-graph is a fully scheduled set of operations that will be computed in a particular order.

5. **Translation to IPU program:** The scheduled-graph is then translated into an IPU program, where each node is replaced by the execution of compute sets.

6. **Lowering to match IPU execution model:** The IPU-program is further lowered into a form that matches the bulk-synchronous parallel execution model of an IPU chip, which consist of sync, exchange and compute steps.

7. **Poplar compiler:** The newly formed program is lowered again so that each tile has its own version of the program with explicit synchronisation and communication steps. This final version of a program in each tile is handled by the Poplar compiler (GCD - Graph Compile Domain) and:

   (a) Only reads and writes data in its own tile memory.

   (b) Contains explicit synchronisation instructions to other tiles.

   (c) Contains data communication routines to exchange with other tiles.

   (d) Just runs that tile's vertices when executing a compute set.

8. **Compiling and execution:** Finally, the resulted per-tile program can be compiled with a conventional compiler to run on the tile. This final version of a program in each tile

Overall, the process used by Tensorflow and Graphcore to optimize the performance of machine learning models on IPU chips includes extracting a computational graph, performing analysis and optimizations using XLA, mapping to a low-level representation, translating to an IPU program, and lowering the program to match the execution model of IPU chips, and finally, compiling the program for execution on the tile.

#### Optimizations techniques

Taking maximum advantage of the compute capabilities of the IPU, optimizing memory management and minimizing communication between host and device are the most crucial factors affecting model performance. Hence, we'll go over the optimization techniques, provided by the Poplar SDK, that we employed to enhance and maximize performance.

**Maximizing compute capabilities**: In the context of ML inference, the concept of batch-size simply refers to the number of combined input samples (e.g., images) which our model will process simultaneously. The purpose of adjusting batch size when testing inference performance is to achieve an optimal balance between latency (speed) and throughput (the total amount of processed samples over time). Because of the lighter load of processing one image at a time, a batch size of 1 often produces the shortest latency times, and can be a good indicator of how a system handles near-real-time inference demands from client devices. Larger batch-sizes (8, 16, 32, 64, or 128) can result in higher

throughput on test hardware, such as IPU or GPU, that is capable of completing more inference work in parallel. However, this increased throughput can come at the expense of latency.

To make optimal use of the processing power, the batch-size has to be selected in a way that all kernel replicas and state variables fits in local memory of the tiles and the processing power of the IPU is taken advantage of. The memory used by the Poplar graph is allocated during compilation and is composed of the main code (i.e. linear algebra operations), the input/output tensors and data exchange code. The BlazeFace model can be considered a "small" model in terms of memory needs due to its lightweight backbone network (inspired by MobileNet V1/V2). A Low memory footprint means more In-Processor-Memory is available and larger batch-sizes can be used to improve performance. Therefore, our starting point was multiple inference runs via larger batch-sizes.

It is important to note here that the value batch-size shall remain constant through the whole execution of our model otherwise a new graph construction (re-compilation) will occur. Because the computational graph is static, the execution instructions loaded on the device depend directly on the batch-size value, as we are going to loop multiple times over data and need to know the number of repetitions in advance. Furthermore, with a different batch-size, a different distribution of the processing onto the tiles will be required, in order to benefit from the synergies of larger batch-sizes.

The tf.distribute.Strategy is a TensorFlow API to distribute training or inference across multiple device units. IPUStrategy is a subclass which targets a single system with one or more IPUs attached and creating our model within the strategy scope was necessary to ensure that it will be compiled and placed on the IPU. Our first implementation was about testing for bulk computations where we load a large number of data from memory. For this approach, the Keras build-in predict() method was initially used for inference on a dataset of 64000 images and the results are presented in Table 5.3 for MK1 and Table 5.4 for MK2.

| **Batch_Size** | **Run_time (s)** | **Throughput** | **Time/image (ms)** | **Compilation_time (s)** |
|---|---|---|---|---|
| 1 | 160.483 | 398.796 | 2.508 | 26.8 |
| 2 | 84.241 | 759.725 | 1.316 | 28.3 |
| 4 | 45.19 | 1416.243 | 0.706 | 34.6 |
| 8 | 30.209 | 2118.574 | 0.472 | 38.9 |
| 16 | 21.904 | 2921.841 | 0.342 | 49.1 |
| **32** | **18.696** | **3423.192** | **0.292** | 64.8 |

**Table 5.3.** *Inference BlazeFace on one MK1 IPU. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

| Batch_Size | Run_time (s) | Throughput | Time/image (ms) | Compilation_time (s) |
|---|---|---|---|---|
| 1 | 134.029 | 477.509 | 2.094 | 126.7 |
| 2 | 70.832 | 903.546 | 1.107 | 131.8 |
| 4 | 42.739 | 1497.461 | 0.668 | 139,2 |
| 8 | 25.541 | 2505.775 | 0.399 | 144.7 |
| 16 | 21.291 | 3005.965 | 0.333 | 156,1 |
| 32 | 19.379 | 3302.544 | 0.303 | 236,4 |
| **64** | **14.1394** | **4526.370** | **0.221** | 240,2 |
| 128 | 15.3025 | 4182.311 | 0.2391 | 292.6 |

**Table 5.4.** *Inference BlazeFace on one MK2 IPU. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

For MK1, batch-size 32 was the highest value we could use to fit our model inside the local memory, obtaining a throughput of 3423 (images/sec) and 0.292 (ms) per image processing time. Since the second-generation MK2 offers more computational power (1472 tiles vs 1216 tiles) and larger on-chip memory compared to MK1 (896 MiB vs 304 MiB), we were able to fit our model with a batch-size up to 128, although best performance of 4526 (images/sec) and 0.221 (ms) per image processing time was obtained with batch size 64. Therefore, 32 for MK1 and 64 for MK2 are the selected batch sizes for our project and further optimizations that we will apply concerns only these values unless stated otherwise.

**Communication Overhead:** Another crucial factor for maximizing IPU performance is I/O optimization. This concerns both the data transfers between the host (CPU) and device (IPU), as well as the memory transactions within the device itself. Data transfers between the host (CPU) and device (IPU) are done via PCIe. These transfers are costly and minimizing them is important for performance.

All the computations of our model are combined into multiple operations in a Poplar graph (see Section 3.2). These operations are called to be executed for each batch in our dataset which adds overhead of passing control to the CPU for each batch. To amortize this overhead, the inference operations are placed into a loop so that they can be executed multiple times on the IPU without returning control to the host. However, the input data needs to take the form of streams of values, and therefore a data pipeline from the dataset into the inference loop on the IPU is needed. The tf.Data API is utilized to construct a Dataset object from data in host memory. The tf.data.Dataset object is an abstraction that represents a sequence of elements, where each element in our case is represented as a multidimensional Tensor. When a Keras model is created inside of an IPUStrategy scope it automatically creates data streams called IPUInfeedQueue and IPUOutfeedQueue for efficiently feeding data to and from the IPU devices.

A data stream is a unidirectional communication from the host to the device, or the opposite, and is defined to transfer a specific number of elements of a given type. This means the buffer storage required by the stream is known. On the IPU side, a stream ob-

ject (FIFO stream) is created and added to the computation graph, while on the host-side the data stream is connected to a buffer allocated in memory. The actual data transaction is done with a Copy program which copies data from the stream to a data-tensor, or from a data-tensor to the stream. There is a maximum buffer size limit of 128 (MiB) per stream copy operation for MK1 and 256 (MiB) for MK2. These I/O streams must synchronise their data transfers, and for that, a callback function is used. The callback function will be called for a device-to-host transfer as soon as the transfer is completed indicating to host that it can read data from the buffer.

To improve I/O communication between host and IPU, we initially made use of steps_-per_execution argument, which sets the number of batches processed sequentially in each execution. The default value is 1, which means that the IPU will only process a single batch and then wait for more data to arrive from the host. By setting steps_per_execution to a higher value we feed our looped inference multiple batches of data asynchronously. This alleviates the extra communication overhead between host and IPU and improved the performance. The total number of batches in the dataset/testset is an important factor that we need to consider when setting the value of steps_per_execution, as the former must be divisible by the latter. Thus, the following empirical formula (Equation 5.1 describes the maximum value of steps_per_execution argument:

$$steps_per_execution_{max} = (dataset_length//batch_size) \tag{5.1}$$

In our case, the dataset_length is 64000 and the batch-size is 32 for MK1 and 64 for MK2 IPU. Therefore, the maximum value of steps_per_execution is 2000 and 1000 respectively. By using the maximum value of steps_per_execution led to a **88%** increase in throughput for MK1 and **55%** increase for MK2, as shown in Table 5.5.

| Device | Batch_Size | Run_time (s) | Throughput | Time/image (ms) |
|---|---|---|---|---|
| MK1 | 32 | 8.746 | 7317.631 | 0.137 |
| MK2 | 64 | 7.99 | 8010.013 | 0.125 |

**Table 5.5.** *Inference results with steps_per_execution = num_of_samples / batch-size. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

After optimizing the host's data loading, we further explored two options provided by IPU TensorFlow and Poplar regarding data movement within the IPU. The first option is to split the IPU into two sets of tile groups, the I/O tile group and the Compute tile group. The former performs only I/O operations to fetch and receive data from outside the chip while the latter can perform general computations. Thus, the IPU-program is also split into two sub-programs to run in parallel, one for I/O and one for computations. The execution flow when using overlapping I/O within IPU is shown in Figure 5.3. Feeding a model enough data to process from the host is a common bottleneck, which affects significantly overall performance. By using a number of dedicated I/O-tiles this bottleneck

can be mitigated, as data transfer and computation overlap over time.

According to [44], the I/O tiles must be able to hold at least a full micro-batch of data and not all the memory on an I/O tile is available for data as a portion of memory is used for code and buffers. Furthermore, is considered more optimal to select a number of I/O tiles that is a power of two. The number of I/O tiles should be carefully tuned, since these tiles cannot participate in computations and using a large number can affect performance. Contrary, using too few I/O tiles can cause the transferred data-tensors to not fit in the available tile memory.

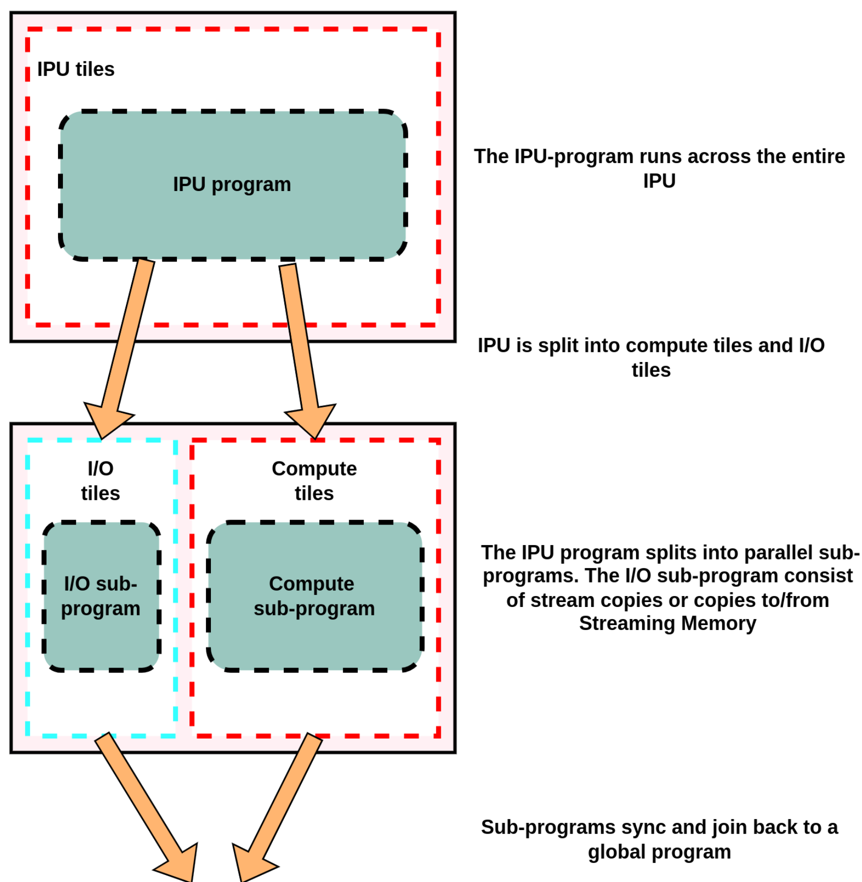



**Figure 5.3.** *Overlapping I/O within IPU*

The second option is called Prefetch and allows TensorFlow and Poplar to move input data logically closer to the IPU before it is needed. By enabling this option the IPU calls the callback function (which is attached to the I/O streams) as soon as possible (i.e. immediately after it releases the stream buffer from a previous transfer) enabling the host to fill the buffer in advance of the transfer. This way, data are available to the IPU before it is needed and therefore less time waiting for them is spent. The number of pre-fetched dataset elements is controlled by the prefect_depth argument.

Furthermore, at the end of the execution on the IPU, a large amount of time is spent dequeuing the resulted detections back to host for further processing (post-processing). Combined with the fact that we have previously set the steps_per_execution to the maximum value, the IPU has to dequeue the whole processed dataset synchronously, which is not efficient. The IPU provides another option called asynchronous callback, where an extra thread is used during execution and is responsible for dequeuing the processed data back to host as soon as they are ready.

A grid search experiment was conducted to explore the optimal values for both prefect depth and I/O tiles when running the BlazeFace model on both MK1 and MK2 IPU-chips with a batch-size of 32 and 64 respectively. We investigated the use of 16, 32, 64 and 128 I/O tiles in combination with prefetching one to three batches of data. We used the maximum value of the steps_per_execution argument (1000 for Mk1 and 2000 for MK2) and the asynchronous callback option was also enabled. Allocating 64 tiles to I/O overlap and prefetching up to two batches of data were the optimal configurations settings for MK2 IPU chip. Our performance improved by 36.5%, compared to the results of Table 5.5, obtaining a throughput of 12614 (images/sec) (Table. 5.6). In the case of MK1 IPU chip, we obtained the best results with the same configuration settings, 64 I/O tiles and a prefetch-depth of two batches. The memory of each tile inside a MK1 IPU system is 256 KiB; almost three times smaller compared to a MK2 tile memory (624 KiB). Therefore, even though a smaller batch-size of 32 is used, the same amount of 64 dedicated I/O tiles reported the best results. The performance improved by 28%, compared to the results of Table 5.5, obtaining a throughput of 9374 images per second (Table. 5.6).

| Device | Batch_Size | Run_time (sec) | Throughput | Time/image (ms) |
|---|---|---|---|---|
| MK1 | 32 | 6,827 | 9374.475 | 0.107 |
| MK2 | 64 | 5.074 | 12614.051 | 0.079 |

**Table 5.6.** *Inference results with steps_per_execution = num_of_samples / batch-size, 64 dedicated I/O tiles and Prefetch enabled. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

**Memory Utilization:** We investigated the memory utilisation of both IPUs with the PopVision Graph Analyzer Tool. As expected, we found that a portion of the memory is consumed by code residing on tiles required to describe the local computation. The

distribution of memory over tiles (Appendix Fig. A.2 & Fig. A.3 ) is even, suggesting efficient memory use in general and effective load balancing among tiles on the IPU. The memory usage spikes on the plot are caused by the 64 dedicated I/O tiles, which are heavily utilized during execution as they constantly fetching data from Host-to-Device.

#### On-Device Inference-Loop Implementation Approach

Deep Neural Networks (DNN) approaches are being increasingly deployed for real-time applications. The real-time nature of these applications creates a requirement for system responsiveness, which imposes strict latency constraints on the inference of the underlying DNN models. The selected BlazeFace model [10] for this project, is a distinct example of a DNN model specifically designed for real-time applications. The bulk-inference approach is useful in cases where a large number of image data reside in memory and we want to process them at once. However, the aim of this project is to build a "streaming" application, where the recording camera directly feeds batches of image data in the subsequent face detection and landmark detection steps. Thus, we experimented with a second implementation approach, called on-device inference loop, in which we read and process a single batch of data on every iteration of the loop.

For this approach, we decided to use a lower-level API, where we have more control over our inference task. Instead of using the build-in Keras predict() method we constructed an on-device inference loop inside of a TensorFlow tf.function(). While the build-in predict() method is useful for bulk-computation tasks, it performs poorly on repetitive calls with a small amount of input data. After investigation, we found that on every predict() call, infeed and outfeed queues are constructed for data manipulation and as soon as execution is finished they get destroyed. This adds constant time overheads on every iteration of our inference task. The creation of these queues on every iteration is the default behavior of the build-in Keras predict() method which we cannot avoid.

The primary advantage of using tf.function() for inference on IPUs is its ability to construct a computation graph from a Python function and optimize it for efficient execution on the IPU hardware.

In contrast, the tf.function() decorator in TensorFlow allows you to create a graph function, which can be optimized for efficient execution on devices like IPUs and GPUs. By converting the computation graph to an optimized form, tf.function() can eliminate some of the overhead associated with dynamic execution in Python, allowing the computation to be executed more efficiently on the IPU. Additionally, tf.function() can be combined with XLA compilation and an on-device inference-loop, which can further accelerate the inference process. When a TensorFlow model is executed on an IPU using tf.function(), the input data is typically first transferred from the host to the IPU, where the model computation is performed. The output data is then transferred back to the host for further processing or storage. However, transferring data between the host and the IPU can be a

bottleneck that limits the overall performance of the model.

The on-device inference-loop approach is designed to reduce the amount of data transfer between the host and the IPU by keeping the input data and model weights on the IPU and only transferring small batches of input data at a time. The inference loop works by repeatedly executing the compiled graph on a small batch of input data, updating the output data, and then fetching the next batch of input data from the host. This process continues until all of the input data has been processed. The on-device inference loop is managed by the Poplar SDK and is automatically used when executing TensorFlow models on IPUs using tf.function(). Apart from loading and executing the model in a tf.function(), we had to manually construct an outfeed FIFO queue, for extracting the output data back to host.

Similarly to the bulk-computation approach we used a batch size of 32 and 64 to compile our model for MK1 and MK2 IPUs respectively. We keep the number of I/O tiles to 64 and same configuration for the outfeed queue but we set the steps_per_execution value to one since we are processing only a single batch of data at a time. The results are presented in Table 5.7.

| Device | Batch_Size | Run_time (sec) | Throughput | Time/image (ms) |
|---|---|---|---|---|
| MK1 | 32 | 7,052 | 9074.475 | 0.108 |
| MK2 | 64 | 5,558 | 11514.051 | 0.08 |

**Table 5.7.** *On-Device Inference loop results for MK1 and MK2 IPUs, Throughput (images/sec), Run_time is the total time of execution for processing 64000 images.*

We observe that the performance of the on-device inference loop approach is quite similar to the bulk-inference approach. Besides achieving satisfactory processing times, the advantage of this approach is that it can accept a single batch of input data directly from the camera and extract the face locations in real-time. This means that the amount of memory needed to store the videos is significantly reduced.

#### Low-latency experiments

As described before, larger batch sizes can improve the performance and scalability of an application (see Tables 5.3 and 5.4). However, we wanted to investigate the trade-off between latency and batch size on IPUs. Therefore, smaller batch sizes of 1, 2, 4, 6, 8, 16 will be investigated with the on-device inference loop approach to test our accelerators for low-latency responses. In this case, we also set the steps_per_execution argument to one and lowered the number of dedicated I/O tiles to 32. The use of a larger number of I/O tiles were also investigated, but performance was decreased. This behavior is expected since our small amount of data on every iteration is not enough to keep all the dedicated tiles busy during execution, and the fact that these tiles cannot participate in computa-

tions led to an underutilization of the IPU chip.

| | MK1 | | MK2 | |
|---|---|---|---|---|
| **Batch size** | **Latency** | **Time/image (ms)** | **Latency** | **Time/image (ms)** |
| 1 | **1.97** | 1.97 | **1.701** | 1.701 |
| 2 | 2.108 | 1,054 | 1.841 | 0,921 |
| 4 | 2.337 | 0,585 | 2.077 | 0,519 |
| 6 | 2.894 | 0,482 | 2.293 | 0,382 |
| 8 | 3.058 | 0,382 | 2.831 | 0,353 |
| 10 | 3.526 | 0,352 | 3.112 | 0.311 |
| 12 | 4.283 | 0,356 | 3.545 | 0,295 |
| 14 | 4.478 | 0,319 | 3.734 | 0,266 |
| 16 | 5.042 | 0,315 | 4.148 | 0,259 |

**Table 5.8.** *Low-latency approach results on MK1 & MK2 IPUs, Latency (ms/batch), steps_-per_execution = 1, 32 dedicated I/O tiles.*

From the results in Table 5.8, we observe that the lowest latencies are obtained for both IPU chips with a batch size of 1. As expected, the latency increases as the batch size gets bigger.

### 5.2.3 Inference on Nvidia Tesla V100

The overall procedure of running a Deep Learning model on a GPU with Tensorflow can be analyzed in the following steps:

1. **Graph Construction**: The first step is the construction of a TensorFlow graph, which defines the computations to be executed. The graph is constructed by adding operations (also known as "Ops") to the graph and connecting the inputs and outputs of the operations. A TensorFlow Op is a node in the computation graph that takes one or more tensors as inputs, performs a computation on these inputs, and returns zero or more tensors as outputs.

2. **Session Creation**: Next, a TensorFlow session is created. A TensorFlow session is an object in TensorFlow that provides an environment for executing operations in a TensorFlow computation graph. It is responsible for setting up the resources necessary to execute the computation graph, such as allocating memory on the GPU or CPU and managing variables.

3. **Operation Initialization**: Before executing the operations, another component of Tensorflow framework, known as TensorFlow executor, is utilized to initialize the operations in the graph. This includes allocating memory for the inputs and outputs of each operation.

4. **Operation Execution**: Once the operations are initialized, the TensorFlow executor traverses the graph and executes each operation individually. During execution, the executor evaluates the inputs to the operations and dispatches the operations

to the appropriate devices (such as CPUs or GPUs). Each TensorFlow Op has a corresponding GPU or CPU implementation, usually written in C++ or CUDA, which is precompiled so that it can be dispatched and executed efficiently. Additionally, the executor manages the transfer of tensors between the GPU and CPU memory to ensure that the computations are executed correctly.

5. **Output Tensors**: Finally, after the operations are executed, the TensorFlow executor returns the results of the computations (output tensors) to the CPU.

The TensorFlow executor and XLA (Accelerated Linear Algebra) compilation can work together to improve the performance of TensorFlow programs. XLA provides an alternative mode of running models: XLA optimization is performed on the TensorFlow graph, before it is executed by the TensorFlow executor. XLA generates GPU kernels for the operations that are specifically optimized for the given model. Because these kernels are unique to the model, they can exploit model-specific information for optimization. The benefit of this over the standard TensorFlow implementation is that XLA can fuse multiple operations (kernel fusion) into a small number of compiled kernels. Fusion can reduce memory requirements and improve performance compared to executing operations individually, as the standard TensorFlow executor does.

As we described in the previous section, the IPU operates by default on an optimized computational graph by using a custom XLA (Accelerated Linear Algebra) compiler. Similarly, XLA compilation was used to construct an optimized graph of BlazeFace face-detector for efficient computations on the GPU-based system. We followed the same on-device inference loop approach ("streaming" application) for executing on the GPU. The higher amount of total memory inside the V100 enabled us to compile and run our model with bigger batch sizes compared to the IPUs. We conducted several experiments with batch sizes up to 512 on a test set of 64000 images. From the results in Table 5.9 we see that the optimal performance is obtained for the maximum investigated batch-size of 512.

| **Batch_Size** | **Run_time (s)** | **Throughput** | **Time/image (ms)** |
|---|---|---|---|
| 32 | 9.81 | 6521.81 | 0.1532 |
| 64 | 5.416 | 11816,838 | 0.0846 |
| 128 | 2.792 | 22968.123 | 0.0436 |
| 256 | 1.704 | 37705.264 | 0.0266 |
| 512 | 0.815 | 79123.324 | 0.0127 |

**Table 5.9.** *Bulk-Inference on Tesla V100 GPU. Run_time is the total time of execution for processing 64000 images, Throughput (images/sec).*

#### Low-Latency Implementation on Tesla V100

The next step was to investigate low-latency responses on the V100 Tesla GPU, similar to what we did with the IPU implementation. We conducted experiments for batch sizes

up to 16 and and results are presented in Table 5.10.

| Batch size | V100 Latency | Time/image (ms) |
|---|---|---|
| 1 | **3.724** | 3.724 |
| 2 | 3.789 | 1,895 |
| 4 | 3.794 | 0,949 |
| 6 | 3.873 | 0,645 |
| 8 | 3.83 | 0,478 |
| 10 | 3.829 | 0.382 |
| 12 | 3.866 | 0,322 |
| 14 | 3.972 | 0,283 |
| 16 | 4.189 | 0,261 |

**Table 5.10.** *Low-latency approach results on V100 Tesla GPU, Latency (ms/batch).*

### 5.2.4 Summary of optimizations for accelerating BlazeFace implementation

A multitude of different optimizations for running the face detection on IPU and GPU have been investigated in this work, the final optimizations are summarized:

#### IPU optimizations

- We made use of the batching method in order to utilize the compute capabilities of the IPUs efficiently while making sure that our model can be compiled and fit inside the local memory. Maximum performance was obtained with a batch size of 32 for MK1 and 64 for MK2.

- We combined the use of 64 dedicated I/O tiles, where data streaming and computations are overlapping over time within the device, with the Prefetch option of the Infeed and Outfeed streams. By enabling the Prefetch option, the CPU can populate the Infeed stream with data before it is needed.

- Because the maximum value of steps-per-execution is used, to minimize the communication overhead as much as possible, the IPU has to dequeue a big amount of data back to host synchronously. By using the asynchronous callback option we enable the device to dequeue data as soon as they are ready by running an extra thread during execution.

#### GPU optimizations

- With a batch size of 512, maximum performance was obtained for the Tesla V100 GPU.

- XLA-compilation generates an optimized model-specific graph by mainly fusing small operation kernels. This leads to less memory requirements and improves execution performance.

## 5.3 Face detection Post-processing

The face detection Blazeface model outputs 896 detections for every image, where each detection contains four bounding box coordinates (x_center, y_center, width, height), twelve landmark coordinates (x1,y1, ..., x12,y2) and a confidence score. The predicted coordinates correspond to 128x128 pixel input image size and therefore we need to map them on the original image dimensions (640x480 pixels).

The third stage is about translating the raw tensor outputs of BlazeFace into actual bounding box coordinates which contain the face region. This stage is running on the host CPU and it's quite fast (0.2 ms per image) since we are able to post-process the raw tensors in batches and therefore no further optimizations are needed.

## 5.4 Landmark Detection

Once the face has been detected, we continue with the landmark detection step. We feed the bounding box coordinates, that we found in previous step, into the landmark detector and a total of 68 facial points is detected on each face. For this step, we initially used the default pre-trained 68-landmark detector of Dlib, which detects 68 facial points on a given input image. The predictor is capable of detecting these points in real-time processing speed with an average of 1.06 (ms) per image processing time.

Our project goal is to capture the eyeblink response (i.e. the ratio of eyelid closure in time), and therefore we are mainly interested in localizing the location of the eyes. Of the total of 68 localized points, we only make use of 12-eye_points and discard the rest.

Furthermore, by using the 68-point detectors the overall **model speed** and **model size** is affected:

**Model speed**: Even though we're only interested in a subset of the landmark predictions, our model is still responsible for predicting the entire set of landmarks.

**Model size**: Since the model needs to know how to predict all landmark locations it was trained on, it therefore needs to store quantified information on how to predict each of these locations. The more information it needs to store, the larger our model size is. For the above mentioned reasons, we decided to train an experimental custom landmark detector to localize just the location of the eyes.

### 5.4.1 Training an experimental custom Dlib landmark detector

Tools for training a custom landmark detector are publicly available through the open-source Dlib library [33]. A landmark detector can be generated from an annotated dataset and by setting certain training options.

#### Annotated dataset

To train our custom Dlib 12-Landmark detector, we'll be utilizing the iBUG 300-W dataset (but with a twist). To create the iBUG-300W dataset, researchers manually annotated and labeled each of the 68 coordinates on a total of 7,764 images. The goal of iBUG-300W is to train a landmark detector capable of localizing each individual facial structure, including the eyes, eyebrows, nose, mouth, and jawline. The dataset itself consists of 68 pairs of integer values — these values are the (x, y)-coordinates of the facial structures depicted in Figure 4.9. Based on the visualization, we can derive which coordinates map to which facial structure. From the total of 68 annotated landmarks we are only interested to keep the ones related to the eyes (37 to 46). Therefore, we will discard all the other annotations and use an adapted version of iBUG 300-W dataset for training a custom 12-Landmark detector.

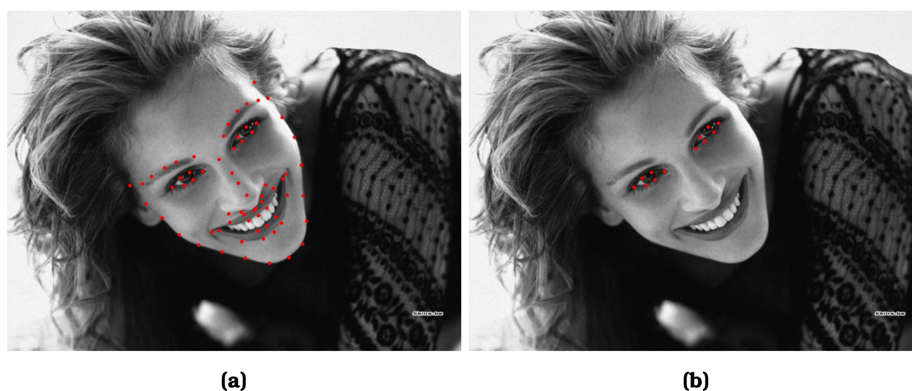



**Figure 5.4.** *a) Depicts the original 68 facial landmarks of an image from the iBUG 300-W dataset [81]. b) Depicts the adapted version with 12 eyes-only landmarks*

#### Training hyperparametes

A set of hyperparameters must be carefully selected for the training process of the landmark-detector. In machine learning, a hyperparameter is a parameter whose value is used to control the learning process. These parameters affect the size, accuracy and speed of the generated model. We will discuss briefly the seven most important hyperparameters that we can set and tune:

- **tree depth**: Specifies the depth of the trees used in each cascade. This parameter represent the "capacity" of the model. Smaller values of tree_depth will lead to more shallow trees that are faster, but potentially less accurate. Larger values of tree_depth will create deeper trees that are slower, but potentially more accurate.

- **nu**: The nu option is a floating-point value (in the range [0, 1]) used as a regularization parameter to help our model generalize. Value close to 1 will emphasize the learning of fixed-data instead of patterns, thus raising the chances for over-fitting to occur. While values closer to 0 will help our model generalize, a considerably larger collection of training samples will be needed in order to perform well.
- **cascade_depth**: Is the number of cascades used to train the model. This parameter affect either the size and accuracy of a model. Choosing a high number of cascades results to a larger and potentially more accurate model. Using fewer cascades results to a smaller model, but it could be less accurate.
- **feature_pool_size**: Controls the number of pixels used to generate features for the random trees in each cascade. Larger amount of pixels will lead the algorithm to be more robust and accurate but to execute slower.
- **num_test_splits**: Is the number of split features sampled at each node. This parameter is responsible for selecting the best features at each cascade during the training process. The parameter affects the training speed and the model accuracy.
- **oversampling_amount**: applies some translation deformation to the given bounding boxes in order to make the model more robust against eventually misplaced face regions.
- **oversampling_translation_jitter**: Controls the amount of translation applied to the dataset.

We refer the reader to the original work of Kazemi et al [40] for a more detailed analysis of all the above parameters.

#### Hyperparameter tuning

A hyperparameter tuning algorithm called *find_min_global* [83] is publicly available via the open-source Dlib library [33] which is based on the work of Cédric Malherbe and Nicolas Vayatis [84]. The above algorithm was used to find the optimal hyperparameter values that will then be used to train our custom landmark detector. A range of values for the different hyperparameters and a maximum number of trials had to be selected.

Execution speed is an important factor on the basis of which the exploration space of the different hyperparameters was chosen. Lower values were chosen for the hyperparameter *tree_depth* which greatly affects the speed of the trained model. However, beyond a fast model we also want it to be able to produce accurate detections. For this reason we chose to explore a larger range of values for the *cascade_depth* and *feature_pool_size* hyperparameters. In addition, the *num_test_splits* hyperparameter does not affect the size and speed of our model but can improve the accuracy, so we choose to explore a larger range of values for this hyperparameter as well. Finally, values closer to zero were explored for the *nu* hyperparameter as it will help our model to generalize. The selected

value ranges for the different hyperparameters are summarized in the Table. 5.11.

| Hyperparameters | Lower bound | Upper bound |
|---|---|---|
| tree depth | 2 | 6 |
| nu | 0.001 | 0.2 |
| cascade_depth | 4 | 25 |
| feature_pool_size | 100 | 1000 |
| num_test_splits | 20 | 300 |
| oversampling_amount | 1 | 40 |
| oversampling_translation_jitter | 0.0 | 0.3 |

**Table 5.11.** *Value range of the hyperparameters to be explored with the find-min-global algorithm.*

The optimal hyperparameter values obtained from running 1000 trials are presented in the second column of Table.5.12. For completeness, in the first column of the same table we present the hyperparameter values used in the pre-trained 68-landmark detection implementation provided by the Dlib library.

| Hyperparameters | Dlib 68-Landmark setting | Custom detector settings |
|---|---|---|
| tree depth | 10 | 4 |
| nu | 0.1 | 0.1033 |
| cascade_depth | 10 | 20 |
| feature_pool_size | 400 | 677 |
| num_test_splits | 20 | 295 |
| oversampling_amount | 20 | 29 |
| oversampling_translation_jitter | 0 | 0 |

**Table 5.12.** *Selected Hyperparameter values*

The above hyperparameter settings were used to train our custom 12-Landmark detector (eyes-only).

### 5.4.2 Evaluating on IBUG-300W and BioID datasets

We initially evaluated the 68-Landmark detector and our custom 12-Landmark detector by using the Mean Average Error (MAE) metric on the IBUG-300W dataset. The results are presented in Table 5.13.

| Landmark Detectors | Size | Train-set error (%) | Test-set error (%) | Number of landmarks |
|---|---|---|---|---|
| Dlib 68-Landmark Detector | 99.7 MB | 6.9 % | 6.2 % | 68-points |
| 12-Landmark Detector | 25.6 MB | 2.1 % | 4.2 % | 12-points |

**Table 5.13.** *Mean Average Error (MAE) evaluation*

To further review the accuracy and speed, we evaluate the models on the BioID database that strongly resembles our testset. We used, once again, the evaluation distance error metric, which has already been described in Section 4.4. The results are

shown in Table. 5.14.

| Landmark Detector | Average error | Time/image (ms) |
|---|---|---|
| Dlib's 68-Landmark Detector | 3.125 % | 1.06 |
| 12-Landmark Detector | 3.205 % | 1.02 |

**Table 5.14.** *Accuracy evaluation on the BioID database*

We observe that both models perform similarly in terms of accuracy and speed. Fig. 5.5.a) depicts a 68-landmark detection obtained from the pre-trained model of Dlib, while 5.5.b) depicts a 12-landmark detection obtained from our custom trained detector on an image of the BioID dataset.

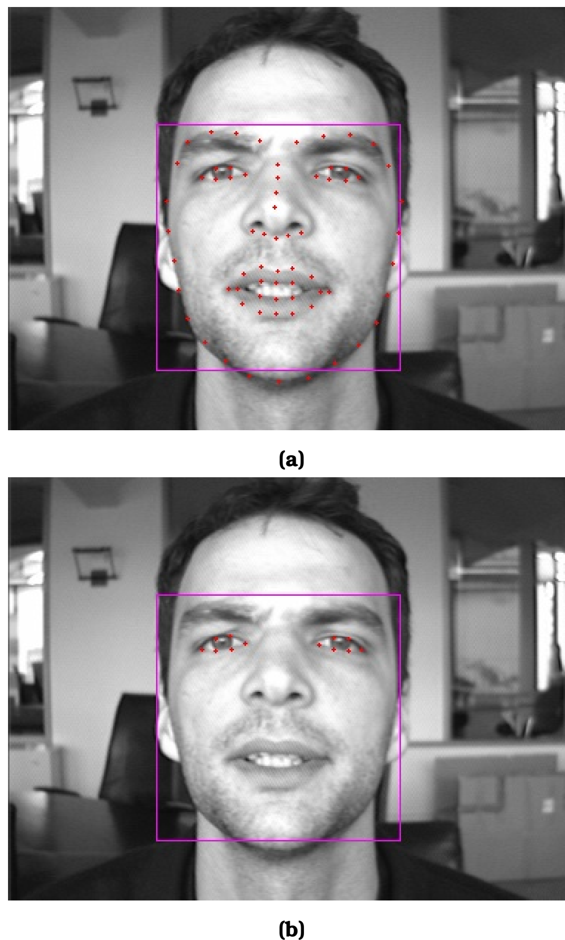



**Figure 5.5.** *a) Depicts a detection of Dlib's pre-trained landmark detector iBUG 300-W dataset. b) Depicts a detection from our custom 12-landmark detector. The image is taken from the BioID dataset [36]*

The implementations above are fast, but we can further accelerate the landmark-detection stage with the use of multiple CPUs.

### 5.4.3 Work-sharing with multiple processes

The landmark-detection stage is running on the host CPU and the multiprocessing library shall be used for multi-core CPU acceleration. Each spawned process will handle the landmark detection of N / P images, where N is defined as number_of_images divided by the number of processes P. Execution times for implementations using 1, 2, 4, 8, 16, 32 and 64 processes are shown in Table 5.15.

| Processes | Time/image (ms) N=64000 |
|---|---|
| 1 | 1.059 |
| 2 | 0.563 |
| 4 | 0.316 |
| 8 | 0.157 |
| 16 | 0.089 |
| 32 | 0.071 |
| 64 | 0.060 |

**Table 5.15.** *Multi-process CPU approach for landmark detection on a sequence of 64000 frames*

## 5.5 Landmark detection Post-processing & Eye-blink detection

The final stage uses the extracted landmarks of the eyes to calculate if a blink has taken place by using the Eye Aspect Ratio (EAR) metric. This stage is combined with the landmark detection step and is executed in parallel by multiple processes.

#### Calculating EAR & Eye-blinks

By calculating the EAR we can determine if a blink is taking place. It generally does not hold that a low value of the EAR means that a person is blinking. A low value of the EAR may occur when a subject closes his/her eyes intentionally for a longer time or performs afacial expression, yawning, etc., or the EAR captures a short random fluctuation of the landmarks. Investigating a larger temporal window ($\pm 3$ images or video frames) will have a big impact on a blink detection for a frame where an eye is the most closed when blinking. An example of the landmark detections on an open and closed eye, as well as the plotted eyeblink response of a processed video, can be seen in Fig. 5.6.

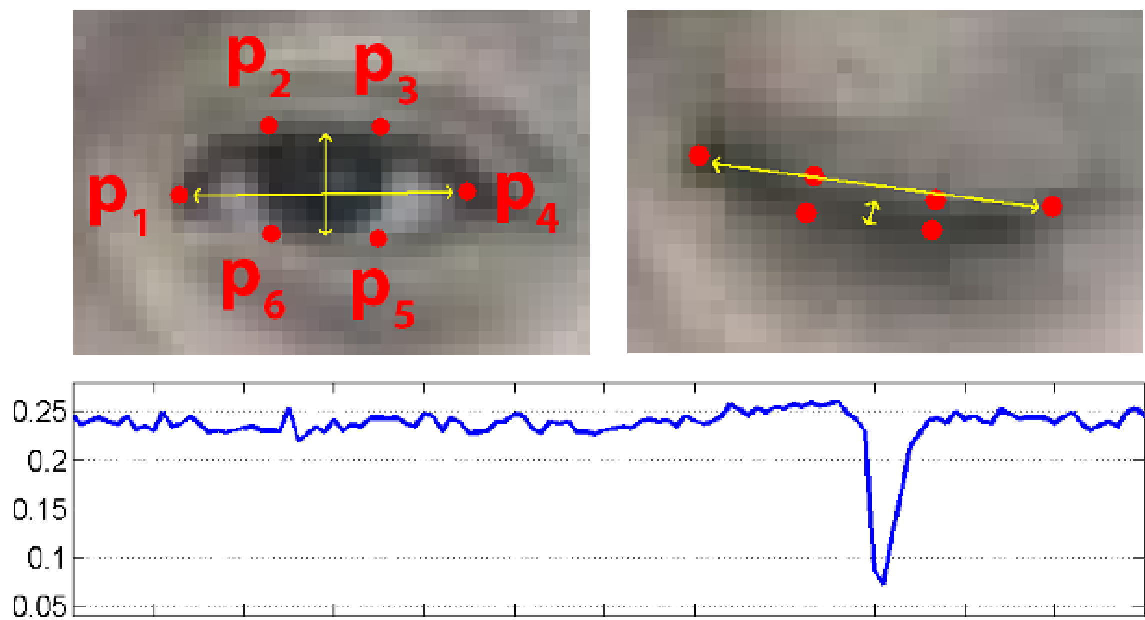



**Figure 5.6.** *The 6 facial landmarks associated with the eye and eyeblink response plot. Image from [38]*

Chapter 6

# Evaluation

In this chapter, the final combined implementation for blink response detection is evaluated. The experimental setups that are used are discussed in Section 6.1. In Section 6.2, the accelerated results of the selected algorithms for the eyeblink-response detection are discussed and compared to the original to review the achieved speedup. In addition, we present how the algorithm performance scales with different CPU and IPU hardware and also we evaluate hardware performance in terms of latency, throughput, and energy efficiency. Section 6.3 discusses the scalability potential of running the face detection step in multiple IPU chips.

## 6.1 Experimental set-up

The specifications of the Tesla V100 GPU that is used for evaluation has been previously described in Section 5.2. The CPU is an AMD EPYC 7551, of which the specifications are summarized in the host column of Table 6.1, along with those of the host memory.

| **Specification** | **V100 Host** |
|---|---|
| Clock Speed | 2.0 GHz |
| Number of cores | 32 (64 threads) |
| PCIe controller | PCIe 3.0 (128 lines) |
| L1 cache | 32 x 64 KiB |
| L2 cache | 32 x 512 KiB |
| L3 cache | 64 MB |

**Table 6.1.** *Specifications of GPU Host AMD EPYC 7551 CPU*

For the evaluation on the IPU hardware platfroms, we used an IPU-POD 16 system and an IPU-server system. The former has four IPU-M2000s racks with 16 MK2 IPU chips in total running on a host server, while the latter is a MK1 PCIe card-based system with 8 C2 PCIe cards and 16 MK1 IPUs on a host server. Each C2 card is a PCIe accelerator card with two MK1 (first generation) IPU processors in it. Key features of MK1 and MK2 IPU chips can be found in Section 5.2. The host CPU of the MK1 IPU-server is an Intel Xeon Platinum 8168, while the host CPU of the IPU-POD 16 system is an AMD-EPYC 7742 CPU. The specifications for both CPUs are summarized in the second and third column

of Table 6.2 respectively.

| Specification | MK1 Host | MK2 Host |
|---|---|---|
| Clock Speed | 2.70 GHz | 2.25 GHz |
| Number of cores | 24 (48 threads) | 64 (128 threads) |
| PCIe controller | PCIe 3.0 (48 lines) | PCIe 4.0 (128 lines) |
| L1 cache | 24 x 32 KB | 64 x 32 KB |
| L2 cache | 24 x 1024 KB | 64 x 512 KB |
| L3 cache | 33 MB | 256 MB |

**Table 6.2.** *Specifications of MK1 Host Intel Xeon Platinum 8168 CPU and MK2 host AMD-EPYC 7742 CPU*

## 6.2 Eye-Blink Response Detection Acceleration results

In this section we shall discuss the acceleration results of the face detection, landmark detection and combined implementation for eyeblink-response detection.

### 6.2.1 Image Loading and pre-processing

The image-loading and pre-processing combined step is performed by multiple processes on the host-side. The number of processes was set to be equal with the number of physical cores of the host-CPU of each system (Table. 6.1 and Table. 6.2). Thus, 32 processes were used for MK1 Host and V100 Host while 64 for MK2 Host. The results are presented in Table 6.3.

| System | # Processes | Time/Image (ms) | Speedup |
|---|---|---|---|
| Naive version | 1 | 2.6 | - |
| AMD EPYC 7551 (GPU) | 32 | 0.381 | 6,8 |
| Intel Xeon Platinum 8168 (MK1) | 24 | 0.438 | 5,9 |
| MD-EPYC 7742 (MK2) | 64 | 0.262 | 9,9 |

**Table 6.3.** *Execution time of the combined image loading and pre-processing step for the host CPUs of our systems.*

From the above results we observe that we can achieve a speedup between 6 to 10 ×.

### 6.2.2 Face Detection on IPU & GPU

Face detection is performed by using the BlazeFace CNN model. The original implementation is constructed through TensorFlow 2.4.4 which provides an intergrated version of Keras library. We maximize the data-level parallelism potential by employing the XLA-compiled model graph on IPU and GPU hardware platforms with batched data. Three different BlazeFace implementations are evaluated:

- The original BlazeFace CPU implementation (see Section 4.5);
- The optimized BlazeFace on-device inference-loop implementation on MK1 and MK2 IPU chips (see Section 5.2);
- The optimized BlazeFace on-device inference-loop implementation on the Tesla V100 GPU (see Section 5.2).

The results are shown in Table 6.4.

| Implementaion | Time/Image (ms) | Speedup |
|---|---|---|
| BlazeFace serial version | 32.727 | - |
| MK1 BlazeFace implementation | 0.107 | 306 |
| MK2 BlazeFace implementation | 0.079 | 414 |
| Tesla V100 BlazeFace implementation | 0.0127 | 2576 |

**Table 6.4.** *Performance comparison of different implementations of the BlazeFace model. Speedup is calculated relative to the slowest implementation.*

Compared to the original sequential version, we achieved a speedup of 306× and 414× with the optimized MK1 and MK2 accelerated versions respectively. The larger amount of memory on the V100 GPU (32GB) enabled us to exploit larger batch-sizes for our inference task. By using a batch-size of 512 we managed to achieve a speedup of 2576×.

### 6.2.3 Landmark Detection on multi-core CPU

The landmark-detection algorithm uses an ensemble of regression trees to refine the estimation of the landmark locations in an iterative process. A data-parallel approach is implemented with multiprocessing library, making use of multiple processes (see Section 5.4.2). In Table 6.5 the accelerated implementations are compared to the sequential implementation.

| System | # Processes | Time/Image (ms) | Speedup |
|---|---|---|---|
| Original serial version | 1 | 1.06 | - |
| AMD EPYC 7551 (GPU) | 32 | 0.071 | 14,9 |
| Intel Xeon Platinum 8168 (MK1) | 24 | 0.078 | 13,6 |
| MD-EPYC 7742 (MK2) | 64 | 0.06 | 17,7 |

**Table 6.5.** *Performance comparison of original sequential landmark-detection algorithm with the multiprocessing-accelerated version. Speedup is calculated relative to the slowest implementation.*

By executing the detector on the Intel Xeon Platinum 8168 CPU (MK1 system) with 32-processes we achieve an approximate speedup of 14× compared to the sequential implementation. Furthermore, a speedup of 15× is achieved by running the detector

on the AMD EPYC 7551 CPU (GPU system) with 32 processes. Finally, the use of 64-processes were investigated for the host CPU of MK2 and a speedup of 17.7× is obtained.

### 6.2.4 Combined implementation for eyeblink-response detection

Of the total process of the eye-blink response detection, we can distinguish three major time consuming steps:

1. **Image loading, decoding and pre-processing**: The images are stored in the JPEG format on a Solid State Drive (SSD). Every image is fetched, decoded and pre-processed into a tensor array of 128x128x3 pixels. This is done by multiple processes concurrently and takes approximately 0.4 (ms) per image for MK1 and V100 hosts and 0.3 ms for MK2 host.

2. **Face Detection on IPU or GPU**: One CPU thread sends the image data to the device machine, which performs face detection on batches of images and returns an array with detection boxes. This step takes approximately 0.079 (ms/image) on a MK2 IPU chip with a batch_size of 64 and 0.107 (ms/image) on a MK1 IPU chip with a batch size of 32. The same step is taking 0.013 (ms/image) on a V100 Tesla GPU with a batch_size of 512.

3. **Landmark Detection**: The landmark detection step is also performed by multiple CPU processes (32 for MK1 and GPU systems, 64 for MK2), running in parallel, which takes approximately 0.07(ms) per image to complete.

We run the above steps sequentially on a test-set of 64000 images. We utilized the on-device inference loop implementation, which means that the accelerated face detection model processes one batch of data in each iteration. Furthermore, the number of processes corresponds to the maximum number of cores available on each CPU for accelerating the landmark detector. The results of running all possible combinations of the selected algorithms on the available hardware platforms are provided in table 6.6.

| Hardware | Landmark model | Time per image (ms) |
|---|---|---|
| MK2 IPU | 68-point | 0.712 |
| MK2 IPU | 12-point | 0.694 |
| MK1 IPU | 68-point | 0.761 |
| MK1 IPU | 12-point | 0.731 |
| Tesla V100 | 68-point | 0.642 |
| Tesla V100 | 12-point | 0.603 |

**Table 6.6.** *Results of the combined eyeblink-detection implementations*

Some stages of our implementation, such as face-detection post-processing and landmark-detection post-processing, have slightly different runtimes, as each platform uses a CPU with different characteristics. However, the deviation observed during the execution of these steps does not exceed 2%, so we consider that such a small deviation does not affect

the overall performance of our implementation.

The achieved processing speed of the complete implementation on the different hardware platforms is estimated to be approximately 1441 FPS on the MK2-IPU, 1367 FPS on the MK1-IPU and 1658 FPS on the Tesla V100 GPU.All the above satisfies the initial requirement of 500 FPS.

### 6.2.5 Low-Latency Hardware Performance

The complete implementation was further evaluated for all three hardware platforms by running experiments with smaller batch sizes. The evaluation metrics are latency, throughput and energy efficiency.

**Latency** is measured in milliseconds per batch. It represents the overall time to get the output results from an input batch. The latency contains the inference and retrieval of the output from the device.

**Throughput** is measured in images per second and is obtained from the latency time measurement and the batch size. This measure represents the load that the hardware can handle for an image-based deep learning application.

The latency and throughput results are visualized in Fig. 6.1 and Fig. 6.2. We experimented with various batch sizes of 1, 2, 4, 6, 8, 10, 12, 14 and 16 due to latency constraint, where small batch sizes are mostly used, especially for real-time inference applications [45]. The results were obtained by running 1000 iterations regardless of the batch size.

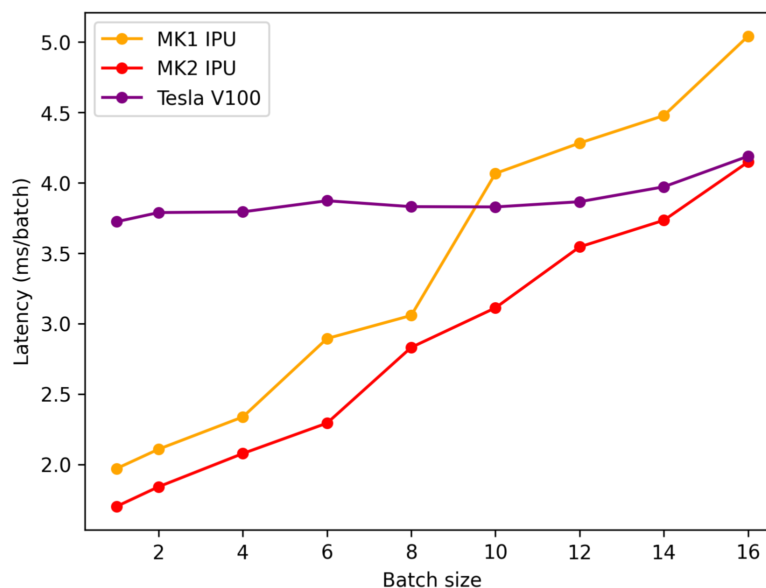



**Figure 6.1.** *Latency results comparison for MK1, MK2 and V100*

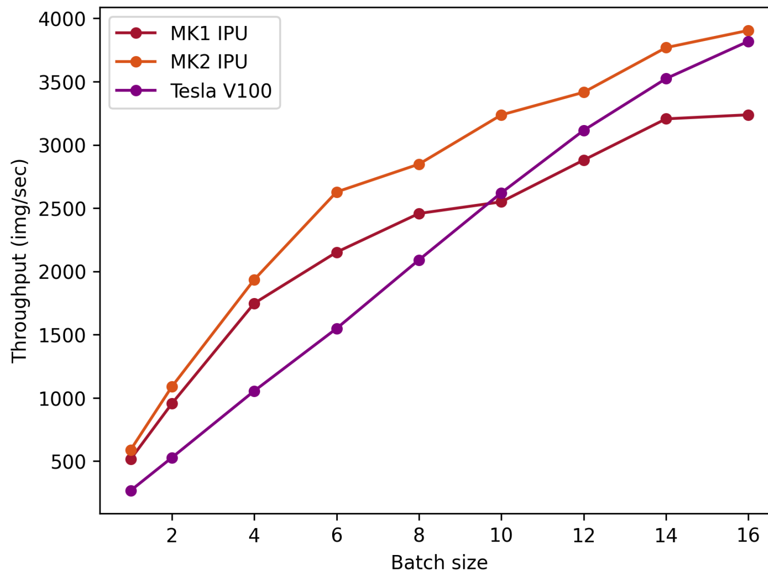



**Figure 6.2.** *Throughput results comparison for MK1, MK2 and V100*

From the above plot diagrams we observe that both MK1 and MK2 IPU chips perform better in terms of latency and throughput for smaller batch sizes (1 to 8) compared to the Tesla V100 GPU where there is no significant increase in latency as the batch size increases. However, for larger batch sizes, the V100 GPU managed to outperform the MK1 IPU-chip, while the MK2 IPU chip achieved the highest throughput and lowest latency.

#### Power Consumption

Power consumption metric is an important factor for hardware evaluation. We made use of gc-monitor from Graphcore driver utilities to measure the power consumption of the IPUs and nvidia-smi interface for the Tesla V100. The measurements are used to compute the energy efficiency of each chip.

#### Energy efficiency

The energy efficiency is measured in images per second per Watt. It represents the energy effectiveness of the hardware on an image-based deep learning application. The power is measured multiple times over a few minutes of inference and averaged. The energy efficiency is the throughput divided by the average power.

From the above table we observe that all three hardware platforms become more energy efficient as the batch-size increases. We also see that the energy efficiency of the MK2 IPU is comparable to that of the V100 GPU. On the other hand, the first generation MK1 IPU has a disproportionately lower energy efficiency. The MK2 has the highest performance of 51.05 images per second per watt.

| Batch size | Energy Efficiency MK1 | MK2 | V100 |
|---|---|---|---|
| 1 | 4,74 | 8,02 | 4,62 |
| 2 | 8,45 | 14,86 | 13,2 |
| 4 | 15.61 | 24,97 | 21,97 |
| 6 | 18,85 | 29,48 | 28,16 |
| 8 | 20,96 | 37,07 | 36,68 |
| 10 | 22,06 | 42,51 | 45,18 |
| 12 | 25,01 | 44,66 | 45,80 |
| 14 | 27,27 | 46,53 | 48,94 |
| 16 | **27,58** | **51,05** | **48,96** |

**Table 6.7.** *Energy efficiency (images/sec/watt) for IPUs and GPU*

## 6.3 Hardware Scalability

### 6.3.1 Face detection on multiple IPUs

As stated before BlazeFace face detection model is considered a quite fast (suitable for real-time applications) and small model in terms of memory footprint, with little more than 100K parameters. In this case, our model can fit inside a single IPU and therefore the concept of data parallelism is well suited and shall be applied to deploy our model on multiples IPUs. IPU-TensorFlow supports automatic data parallelism when multiple IPU devices are configured with the system. Data parallelism is achieved by replication of the Poplar graph, where the number of times the model is replicated is called the replication factor, and higher replication factors allow higher data throughput.

For inference, the way of doing this is to use a Multi-instance/Single host launch mode where multiple instances of our script are launched on the same host server with mpirun. Each instance is connected to a single instance of the replicated model operating independently on a different fraction of our dataset with a separate Graph Compile Domain (GCD). Therefore, each independent replica of BlazeFace operates on a micro batch and the results from each replica are enqueued in the outfeed queue and sent back to the host (Fig. 6.3).

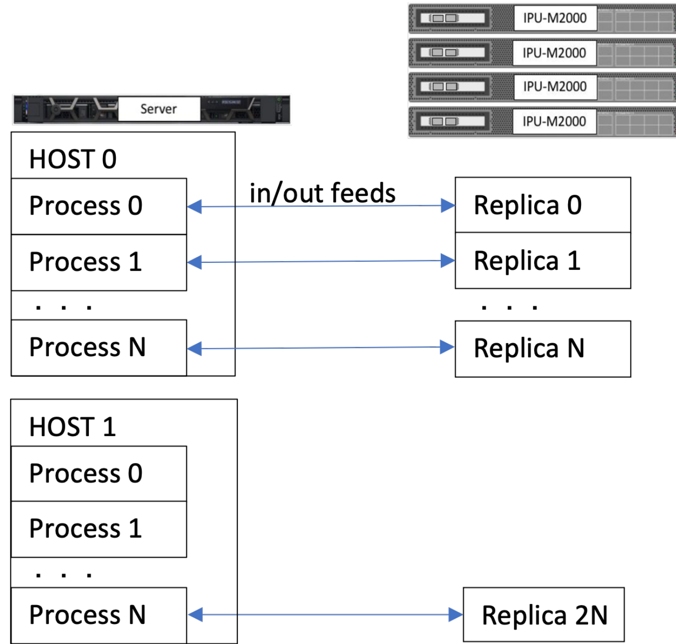



**Figure 6.3.** *Multi-instance replication*

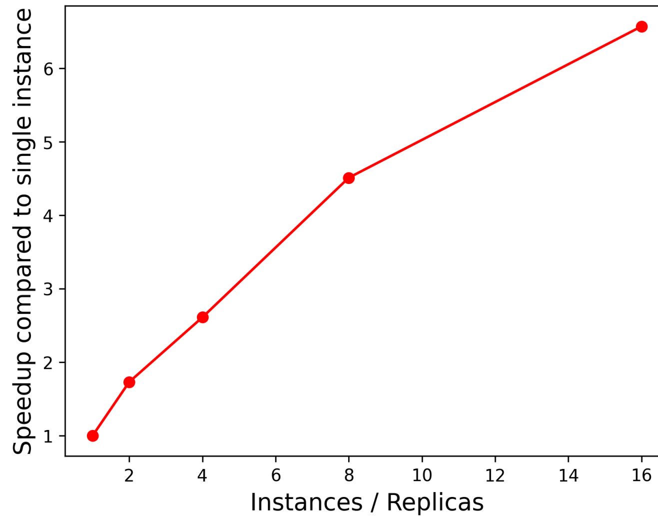
We evaluate the inference scalability of the BlazeFace model throughput with multiple instances on the given IPU-POD16 cluster with 16 MK2 IPUs and the IPU-Server with 16 MK1 IPUs. Our testing dataset contains 64000 images and each instance/replica of our model will operate on a chunk of images equal to 64000 / number_of_instances. The results can been seen in Table. 6.8 and Table. 6.9. From the speedup plots (Fig. 6.4.a and Fig. 6.4.b), we observe that performance scales almost linearly with the number of IPUs.

| Configuration | Throughput (img/s) | Speedup | Scaling efficiency % |
|---|---|---|---|
| 1 IPU | 12454.997 | 1 | 100 |
| 2 IPU | 17939,733 | 1.44 | 72.01 |
| 4 IPU | 27187,765 | 2,18 | 54.56 |
| 8 IPU | 37230,948 | 2,98 | 37.36 |
| 16 IPU | 76555,858 | 6.15 | 38.45 |

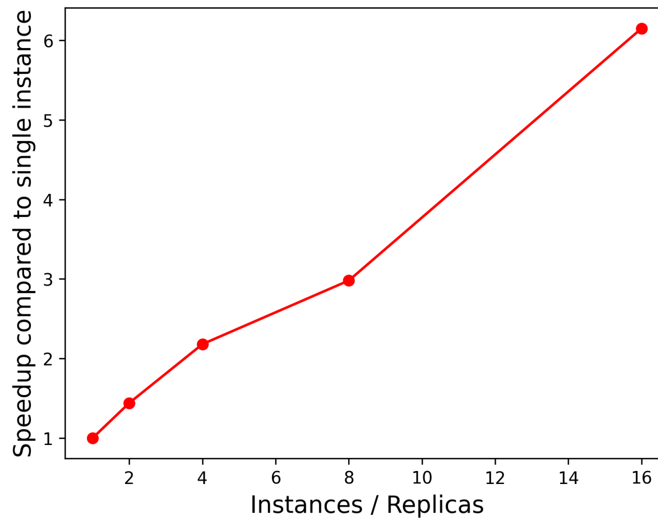
**Table 6.8.** *Inference scalability for BlazeFace model on an IPU-POD16 server, per replica batch_size = 64*

| Configuration | Throughput (img/s) | Speedup | Scaling efficiency % |
|---|---|---|---|
| 1 IPU | 9374.475 | 1 | 100 |
| 2 IPU | 16260,162 | 1,73 | 86.72 |
| 4 IPU | 24464,831 | 2.61 | 65.24 |
| 8 IPU | 42272,126 | 4,51 | 56.36 |
| 16 IPU | 61657,032 | 6,57 | 41.11 |

**Table 6.9.** *Inference scalability for BlazeFace model on an MK1 IPU-server, per replica batch_size = 32*




(a)




(b)

**Figure 6.4.** *Multi-instance Speedup for a) MK1 IPU-Server with 16 MK1 IPU chips and b) MK2 IPU-POD16 with 16 MK2 IPU chips*

### 6.3.2 Minimum hardware to meet the requirements

The required detection speed of the eyeblink response project is 500 frames per second. Because possible future work for this project could be to investigate the possibilities for a more mobile solution, an estimation is made of the minimum required hardware to achieve a detection speed of 500 FPS.

The complete eyeblink-detection implementation consists of five stages: image loading/preprocessing (ILP), face detection (FP), face-detection-post-processing (FDPP), landmark detection (LD) and EAR computation (EAR). The face detection step is performed on an IPU or GPU hardware platform while the rest on the host CPU. The ILP and LP steps are com-

puted by using multiple processes. Therefore, we can define the total eyeblink response detection time as:

$$time = ILP + FD + FDPP + LD + EAR \tag{6.1}$$

A detection speed of 500 FPS equals a maximum detection time of 2 ms. Therefore, we conducted several experiments with the low-latency approach on all three hardware platforms to find the minimum required batch size and number of processes which satisfy the execution time requirement of <2 ms. This can be seen in Table 6.9.

| Device | Host | Processes | batch_Size | time/image (ms) |
|---|---|---|---|---|
| MK1 | Intel Xeon P. 8168 | 8 | 10 | 1.744 |
| MK2 | AMD EPYC 7742 | 4 | 8 | 1.536 |
| V100 | AMD EPYC 7551 | 6 | 10 | 1.698 |

**Table 6.10.** *Minimum required hardware for execution time <2 ms)*

From this table we can deduce that a batch size of 10 and 8 CPU processes are the maximum hardware that needs to be utilized. Note that this estimation requires a CPU with comparable performance per thread as the ones used in our experimental set-ups.

Chapter 7

# Analyzing Performance of IPU and GPU Platforms for CNN-Based Model Training

In Chapter 5 we explored the potential of two IPU-based hardware systems (MK1 and MK2) and one GPU-based hardware system (Tesla V100) to accelerate a face detection inference task with the goal of achieving real-time processing speed. A pre-trained CNN-based model was chosen for the face detection step, as training a robust and accurate Deep Neural Network model is a very time-consuming and non-trivial task [23] [46]. In the inference, the network only experiences the forward-pass, during the training, it experiences both the forward-pass and the backward-pass. As a consequence, the training requires a much more extensive computational effort compared to that for the inference. It involves the holistic use of all the resources in a server from storage and CPU for fetching and pre-processing the dataset to the specialized hardware (GPU, IPU) that perform computation on the transformed data.

In this chapter we will explore and compare the capabilities of the available hardware accelerators (IPU, GPU) in the computationally intensive process of training an image-based CNN model.

## 7.1 Implementation

An open end-to-end training implementation of BlazeFace model will be constructed from scratch.

### 7.1.1 BlazeFace architecture

Our starting point was the original work of [10] which provides all the information needed to construct the model architecture. The overall network structure of BlazeFace is provided in Fig. 7.1. It takes a $128 \times 128$ RGB image as input and extracts features through 5 Single Blazeblocks and 6 Double Blazeblocks (Fig. 7.2).

The reader can also refer to the detailed analysis of BlazeFace in Section 4.5

| Layer/block | Input size | Conv. kernel sizes |
|---|---|---|
| Convolution | 128×128×3 | 128×128×3×24 |
| Single BlazeBlock | 64×64×24 | 5×5×24×1<br>1×1×24×24 |
| Single BlazeBlock | 64×64×24 | 5×5×24×1<br>1×1×24×24 |
| Single BlazeBlock | 64×64×24 | 5×5×24×1 (stride 2)<br>1×1×24×48 |
| Single BlazeBlock | 32×32×48 | 5×5×48×1<br>1×1×48×48 |
| Single BlazeBlock | 32×32×48 | 5×5×48×1<br>1×1×48×48 |
| Double BlazeBlock | 32×32×48 | 5×5×48×1 (stride 2)<br>1×1×48×24<br>5×5×24×1<br>1×1×24×96 |
| Double BlazeBlock | 16×16×96 | 5×5×96×1<br>1×1×96×24<br>5×5×24×1<br>1×1×24×96 |
| Double BlazeBlock | 16×16×96 | 5×5×96×1<br>1×1×96×24<br>5×5×24×1<br>1×1×24×96 |
| Double BlazeBlock | 16×16×96 | 5×5×96×1 (stride 2)<br>1×1×96×24<br>5×5×24×1<br>1×1×24×96 |
| Double BlazeBlock | 8×8×96 | 5×5×96×1<br>1×1×96×24<br>5×5×24×1<br>1×1×24×96 |
| Double BlazeBlock | 8×8×96 | 5×5×96×1<br>1×1×96×24<br>5×5×24×1<br>1×1×24×96 |

**Figure 7.1.** *BlazeFace feature extraction network architecture*

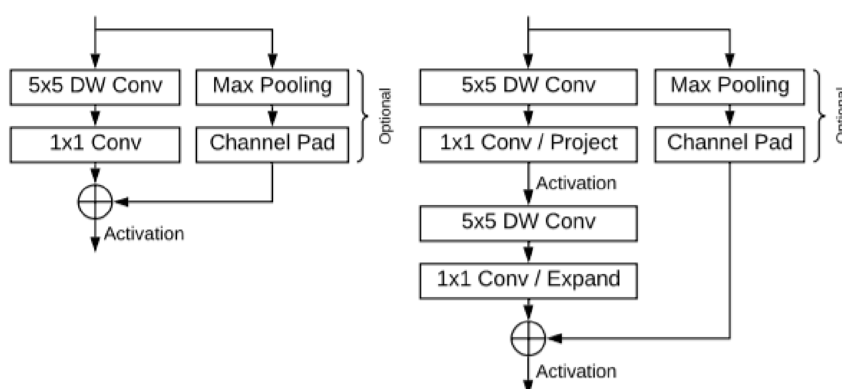



**Figure 7.2.** *BlazeBlock (left) & double BlazeBlock (right)*

### 7.1.2 Loss Function and Optimizer

The training process of the BlazeFace model is closed source and thus important information about which cost function and which optimization algorithm should be used to train the model from scratch is not available.

BlazeFace follows the SSD approach which is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances (in our case face or no-face) in those boxes. Therefore, the face detection task can be described as a combination of two different subtasks:

- Bounding Box detection, which is the localization of the face region in the image.
- Classification, which classifies a bounding box to the right class (in our case as face or no-face) based on a confidence score.

Therefore, the loss function that will be used for training our model must combine the cost functions of these two different subtasks. Recent implementations of single-stage face detectors, such as S3FD [24] (2017) and EXTD(2019) [85], make use of multitask loss function, which is originally introduced in [15]. This multitask loss function is a weighted sum of the localization loss (loc) and the confidence loss (conf):

$$L = \frac{1}{N_{cls}} L_{cls}(p, y) + \frac{1}{N_{box}} L_{box}(t_i, t_i^*) \tag{7.1}$$

The localization loss $L_{box}(t_i, t_i^*)$ is a Smooth L1 loss [86], where $t_i = (t_x, t_y, t_w, t_h)_i$ and $t_i^* = (t_x^*, t_y^*, t_w^*, t_h^*)_i$ represent the coordinates of the predicted box and ground-truth box associated with the positive anchor.

$$L_{box}(t_i, t_i^*) = \sum_{i \in (x,y,w,h)} smooth_{L1}(t_i - t_i^*) \tag{7.2}$$

where

$$smoooth_{L1}(x) = \begin{cases} 0.5x^2, & \text{if} |x| < 1 \\ |x| - 0.5, & \text{otherwise} \end{cases} \tag{7.3}$$

The network prediction for the class of an area of interest is given as a distinct probability distribution across two classes (face vs. no-face).

$$p = (p_0, p_1) \tag{7.4}$$

The classification loss $L_{cls}(p, y)$ is binary cross-entropy [15], also called log loss, over two classes (face vs. no-face/background). With a ground truth label $y \in 0, 1$ and a probability estimate $p = Pr(y = 1)$ , the log loss per sample is the negative log-likelihood of the classifier given the true label:

$$L_{cls}(p, y) = -\log(p_y) \tag{7.5}$$

In our implementation, the two losses are normalized by $N_{cls}$ and $N_{box}$ where the cls term is normalized by the number of positive and negative anchors, and the box term is normalized by the number of positive anchors.

As Optimizer we chose Adam [52], where he is one of the most widely used optimizers in the field of image recognition and the learning rate was set to the default value (0.001).

### 7.1.3 Training Datasets

Two datasets were selected for training the BlazeFace model. The first selected dataset is called FDDB [47] and is one of the most commonly used benchmarks for face detection and consists of 2845 images with 5171 face annotations collected from journalistic articles. It is a really challenging dataset mainly due to the fact that it is rich in occluded and out-of-focus cases.

The second dataset is called 300W-LP [48] and it's available through TensorFlow Datasets (TFDS) library. TFDS exposes public research datasets as tf.data.Datasets or/and as NumPy arrays. It does all the hard work of fetching the source data and preparing it into a common format on disk, and it uses the tf.data API to build high-performance input pipelines, which are TensorFlow 2.x-compatible and can be used with tf.keras models.

The 300W-LP Dataset standardises multiple alignment databases with 68 landmarks, including AFW, LFPW, HELEN, IBUG. From these, we kept only the face bounding boxes for our training. The dataset contains a total of 61,225 samples which we split into 80% train samples and 20% validation samples.

#### Data augmentation

Data Augmentation based on image manipulations offers a set of techniques to improve the size and quality of a training dataset. It is usually based on the generation of additional images or annotations based on transformations performed on the original dataset. It is also one of most basic and widely used techniques for dealing the overfitting problem and improving accuracy when training a Deep Neural Network model [87][88][89]. The technique of data augmentation works in the following way: in each epoch, when images are fed into the network to be trained, some changes are made to the images so that they are not exactly the same with those seen by the model in previous epochs. The modifications made to the images contain some randomness, i.e. many are applied with a certain probability or their degree of influence is determined by some random variable, so that the modified images are not repeated. Several methods for this exist:

**Geometric transformations**, also known as spatial transformations, are transformations of the image coordinate system. It refers to operations that transform an image using variations of the shape. Some of the most frequent geometric transformations are flipping, rotation, cropping, translation, or scaling.

**Color transformation**, also known as photometric transformations, are transformations over the pixels values in the matrices which compose an image, rather than the pixel positions. Some of the most frequent color transformations are changes in brightness, contrast, colorspaces and normalizations.

To make our model more robust to various input object sizes and shapes, each training image is randomly sampled by the following options:

- Photo-metric distortions (Random brightness, contrast, hue)
- Horizontal Flip
- Cropping
- Padding

**Feeding data with tf.data.Dataset API**

In the context of ML training, especially with Deep Neural Networks (DNN), the input data pipeline accounts for significant resource usage and can greatly impact end-to-end performance. A recent study [90] of ML model training with public datasets found that pre-processing data accounts for up to 65% of epoch time, where an epoch is termed as a complete pass over the training dataset. This shows that input data pipelines consume a significant fraction of ML job resources and are important to optimize.

Hardware accelerators used for ML training further increase the need for efficient input pipelines. The input data pipeline of DNN training can be characterized as a three-stage extract, transform, load (ETL) process. The first stage reads input data from a storage system. The second stage transforms the data, commonly on the CPU, to a format amenable to ML training computation. Finally, the third stage loads the data onto the accelerator device that executes the training computation. Today's accelerators, such as GPUs and IPUs, are tailored towards executing the linear algebra operations that are common in ML computations and the input data pipeline operates in parallel with these computations. Ideally, the data pipeline should steadily feed pre-processed data items to the accelerator devices to keep them continuously busy processing data. However, DNN training is often I/O-bound, bottlenecked by fetching the data from storage, or CPU-bound, bottlenecked by applying data pre-processsing in memory.

Raw input data, such as images, undergo both offline and online pre-processing before being ingested for model training. While some data transformations, such as normalization, are applied during offline pre-processing, ML training also requires applying transformations online as examples are fed to the model. For instance, image models commonly rely on data augmentation, e.g. randomly distorting images, to improve accuracy [91]. As described in previous sections, various pre-processing and data augmentation methods (i.e. resizing, normalization, conversion to RGB color-space) are part of our overall training pipeline and therefore optimizing the pipeline is of key importance for the performance.

In order to optimize our input data pipeline we will make use of tf.data API which enables us to efficiently utilize available host resources. The tf.data provides transformations that enable software pipelining, and parallel execution of computation and I/O. Key

transformations that we used in order to build our input data pipeline are:

The **prefetch** transformation decouples the producer and consumer of data by using an internal buffer, making it possible to overlap their computation. Input pipelines can use this transformation to overlap host computation, host-to-device transfer, and device computation. In particular, the transformation uses a background thread and an internal buffer to prefetch elements from the input dataset ahead of the time they are requested.

The **map** transformation applies a user-defined function to each element of the input dataset. When preparing data, input elements may need to be pre-processed. Because input elements are independent of one another, the pre-processing can be parallelized across multiple CPU cores. To make this possible, the map transformation provides the num_parallel_calls argument to control the level of parallelism.

The **cache** transformation can cache a dataset, either in memory or on local storage. This saves some operations (like file opening and data reading) from being executed during each epoch

The **shuffle** transformation maintains a fixed-size buffer which is populated with random data entries. Shuffling data serves the purpose of reducing variance and making sure that our model remains general and overfits less. Since we perform computations on batches of data , the batch gradient descent is activated. The idea behind batch gradient descent is that by calculating the gradient on a single batch, we can get a fairly good estimate of the "true" gradient. That way, we save computation time by not having to calculate the "true" gradient over the entire dataset every time. Therefore, it is important to create batches that are representative of the overall dataset, otherwise our estimation of the gradient could be off.

In addition, the tf.data runtime contains an auto-tuning mechanism that allocates CPU and RAM resources across various parts of the input pipeline in a way that minimizes the (expected) latency of the input pipeline producing an element. This is called tf.data.AUTOTUNE

Therefore, we made use of the tf.data.Dataset API to prepare our dataset, pre-process our input images and apply data augmentation with multiple processes. The tf.data.AUTOTUNE option was utilized to auto-tune the number of used processes. In addition, we enabled data shuffling and configured the buffer size to be equal to the number of training samples. Finally, the prefetch and cache transformations were also applied with auto-tuning enabled. By combining all these transformations we obtained a speedup of roughly 2x in our total training time for all hardware platforms.

### 7.1.4 Training BlazeFace on IPU and GPU hardware platforms

We utilized the Tensorflow and Keras APIs for constructing the model architecture and performed training on IPU and GPU hardware platforms. More specifically for the IPU implementation we used the Poplar SDK v2.4 with its IPU interface to TensorFlow v2.4.4 which integrates the Keras library. The same version of TensorFlow was used for the GPU implementation for fair comparison of the final results.

#### Training BlazeFace on MK1 and MK2 IPUs

In order to be able to train our model on an IPU-based system, our source code first had to be compiled and translated into an IPU-program. This process has been analyzed in more detail in Section 5.2.2. Porting our Tensorflow/Keras model into IPU was relatively easy. Graphcore implementation of Keras provides the IPUStrategy subclass of tf.distribute.Strategy API which targets a system with one or more IPUs attached. Creating variables and Keras model within the scope of the IPUStrategy object is all we had to do to ensure that they are placed on the IPU. Then, the Keras build-in fit() method was used for the training process.

It is important to note that we had to decide all hyperparameter values in advance, since the model compilation is static and the execution instructions loaded on the IPU device depend directly on them. We mainly experimented with different values of the batch-sizes, since our experiments mainly focus on the total runtime that each accelerator needed to complete the training of our model for a certain number of epochs. For MK1, a batch-size of 8 was the highest value we could use to fit our model within local memory, while MK2's larger available on-chip memory allowed us to use a batch-size equal to 16.

In addition, we used the maximum possible value of the steps_per_execution hyperparameter, for all the experiments we conducted, to reduce communication overheads and maximize the performance of our model (see Section 5.2.2 for a more detailed analysis). This value is set based on the selected batch-size value as the number of batches in our datasets had to be divisible by the steps_per_execution argument. Furthermore, some extra care had to be taken when we prepared the dataset for training our Keras model on the IPU. The Poplar software stack does not support using tensors with shapes which are not known when the model is compiled, so we made sure the sizes of our datasets were also divisible by the selected batch-size of each experiment.

Keras models created inside of an IPUStrategy scope automatically create IPUInfeedQueue and IPUOutfeedQueue data streams for efficiently feeding data to and from the IPU devices when using the build-in fit() method. The IPUInfeedQueue accepts batches of input images directly from our tf.data.Datasets. Furthermore, the IPU is using a callback function to indicate that the host can read data from the buffer (for IPU to host transfer) or that the host can populate the stream (for host to IPU transfer). Similarly to the inference task we enable the Prefetch option (see Section 5.2).

We followed Graphcore's official documentation, which suggests to use a prefetch depth of 3 when training a model.

**Training BlazeFace on Tesla V100 GPU**

The tf.distribute.Strategy API was also used to train the Keras model on the GPU-based system. For a fair comparison, we enabled XLA compilation when training our model on the GPU, as IPU-based systems use it by default to perform transformations/optimizations on the model graph.

The fact that the V100 tesla GPU offers a much larger memory size than IPUs allowed us to investigate larger batch sizes for training our model. We conducted experiments with batch-size values up to 64 and 128 for FDDB and 300W-LP dataset respectively. Again, an optimized tf.data.Dataset was constructed for feeding batches of data to the GPU.

## 7.2 Training Evaluation of BlazeFace

In this section we evaluate the training results of BlazeFace model on IPU and GPU hardware platforms. The chosen evaluation metric is the total training time for a specific amount of epochs.

### 7.2.1 Training Resutls

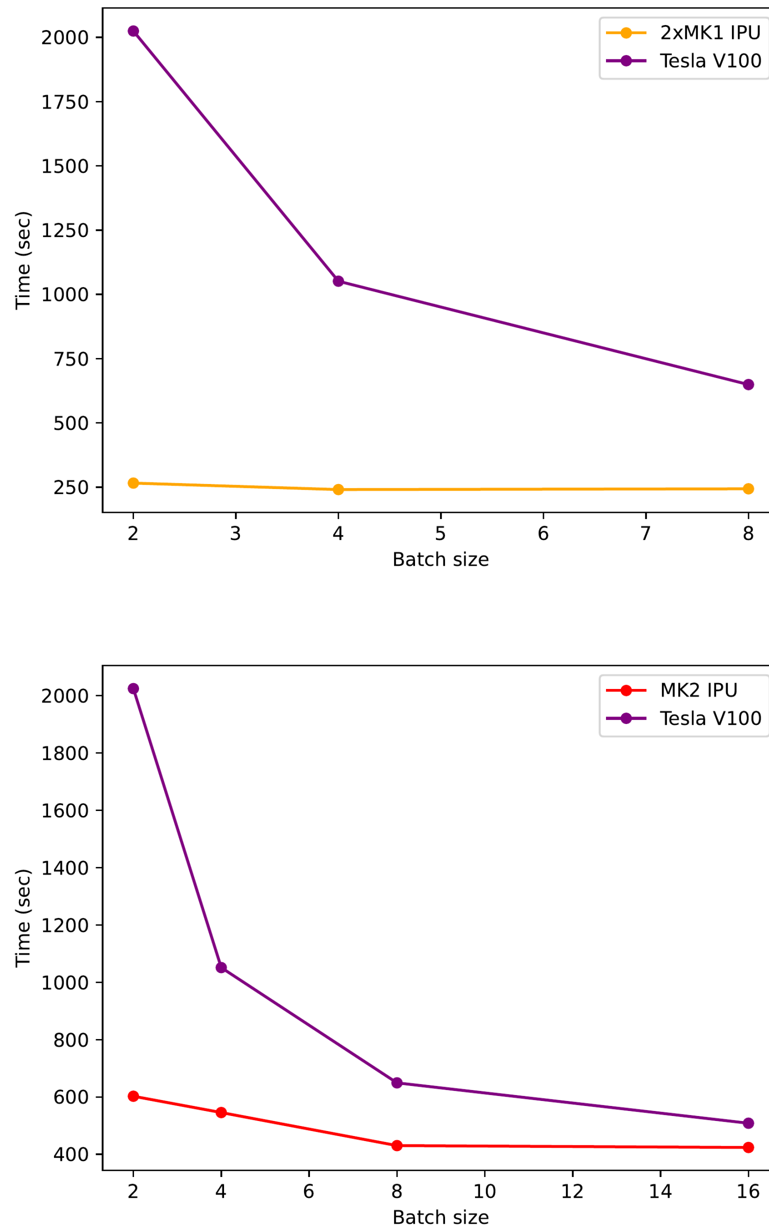



**Figure 7.3.** *Training time of BlazeFace model on the FDDB dataset. (top) The results for various batch sizes on V100 GPU and MK1 IPU chips. (bottom) The results for various batch sizes on V100 GPU and MK2 IPU chips.*

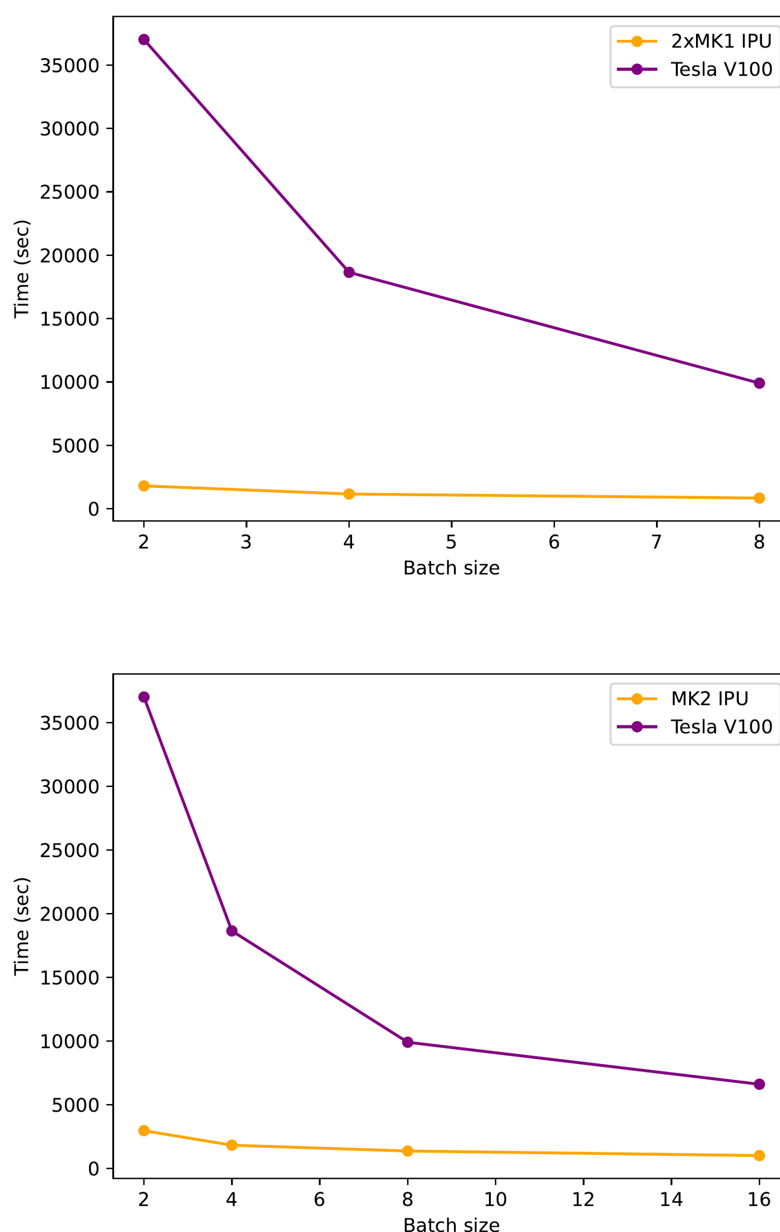



**Figure 7.4.** *Training time of BlazeFace model on the 300W-LP dataset. (top) The results for various batch sizes on V100 GPU and MK1 IPU chips. (bottom) The results for various batch sizes on V100 GPU and MK2 IPU chips.*

The model was trained for 150 epochs on the FDDB dataset, consisting of 2845 images, and for 100 epochs on the 300W-LP dataset, which was considerably larger with a total of 61,225 training samples. We used a C2 PCIe card with two MK1 IPUs attached, with a TDP value of 250 W that matched the rest of our hardware accelerators.

The results, presented in Figure 7.3 and Figure 7.4 for FDDB and 300W-LP respectively, show that both MK1 and MK2 IPUs outperform the V100 GPU in training speed. Even for greater batch sizes, the network was trained almost twice as fast compared to the GPU. Additionally, for very small batch sizes, the IPUs achieved superior performance.

In our experiments, we also conducted tests with larger batch sizes for training our model on the Tesla V100 GPU, as it offers a higher amount of total memory compared to the IPUs. For training on the FDDB database, we employed batch sizes of up to 64 and up to 128 for the 300W-LP database. A detailed presentation of the training results can be found in Appendix Tables A.1 and A.2. These tables contain some additional information about the overall training speed times (time per epoch, time per step) as well as the achieved loss and validation loss for each run.

The choice of batch size should be carefully considered based on the specific characteristics of the dataset, the model architecture, and the available computational resources when training a model. Using a larger batch size can lead to more stable gradients and faster convergence [49] [50]. This is because a larger batch size allows the optimization algorithm to use more information from the training data to compute the gradient of the loss function with respect to the model parameters. With more information, the gradients computed are more representative of the true underlying distribution of the data and less sensitive to random fluctuations, which can result in more stable updates. However, there are also some trade-offs when using larger batch sizes, such as increased memory usage and computational resources required for training. Additionally, large batches may generalize less well than smaller batches, especially for models with a high capacity.

On the other hand, smaller batch sizes result in more frequent weight updates, incorporating more diverse information into the model, but may result in more noisy gradients as each batch represents only a small sample of the overall dataset and can be subject to more sampling variability. Furthermore, a smaller batch size may result in slower convergence, requiring more careful tuning of the learning rate and other hyperparameters, as smaller batches can result in more drastic changes to the gradient and may require more frequent adjustments to the optimization algorithm. For example, some optimization algorithms, such as SGD [51], perform better with small batch sizes, while others, such as Adam [52], can handle larger batch sizes.

It is important to note that apart from optimizing the input tf.Data.Dataset and compiling with XLA, we did not fine-tune any other hyperparameters in our training experiments (i.e. learning rate, optimizer, etc.). The goal of the conducted experiments is to compare the training performance on each platform in terms of execution speed.

Using larger batch sizes resulted in much better GPU performance. GPUs are optimized for parallelism and high throughput, making them well-suited for processing large batches of data quickly. However, as the batch size decreases, the overhead associated with data transfer and memory management on GPUs becomes more pronounced, leading to lower performance.

IPUs, on the other hand, perform better at small batch sizes. We argue that this is due to their architecture, which is optimized for matrix operations and high-speed communication between processing elements (i.e., I/O tiles, IPU exchange communication

fabric). This allows the IPUs to effectively utilize their resources and process small batches of data efficiently, while reducing the memory overhead associated with larger batch sizes.

Chapter 8

# Conclusions

This chapter presents the conclusion of the thesis work. The contributions of this research are discussed in Section 8.1. Section 8.2 provides a discussion on the final results of the inference and training experiments conducted in this study, highlighting the pros and cons of each hardware platform used. Lastly, Section 8.3 outlines possible directions for future work in this area of research.

## 8.1 Contribution

The initial goal of this thesis was to construct and deploy a Machine Learning application on IPU-based and GPU-based systems in order to evaluate their performance in inference tasks. The goal of this application is to detect the amount of human eyelid closure in video data in a real-time processing speed of 500 frames per second. The selection of the appropriate algorithms was of key importance for the whole project since we had to maintain a balance between speed and accuracy. The execution time constrain for this project was also a crucial factor, and so IPU and GPU hardware device accelerators were used to accelerate the selected algorithms to fulfill the requirements. Creating an online implementation would not only alleviate the need for large off-line data storage, but also enable the neuroscientists to dynamically adjust eyeblink-conditioning experiments based on immediately available feedback on the subject's performance.

The detection process is divided into two major stages: the first stage detects the position of the human face in an image, while the second stage uses this face detection to determine the amount of eyelid closure. The selection of the appropriate face detector was based on a set of requirements that specify under which circumstances the detector must be able to detect a face. Three face detectors, one classical ML approach and two deep learning CNN-based models, were evaluated on a subset of images of the Annotated Facial Landmarks in the Wild database that meets these requirements. The MTCNN model was the most accurate detector but also ten times slower compared to the other methods. Both the Histogram of Oriented Gradients (HOG) algorithm and BlazeFace model are much faster alternatives, with the former being faster and the latter being more accurate. Although both detectors were considered suitable for this project we decided to proceed with BlazeFace as it was more accurate than HOG and had similar speed performance.

In addition, BlazeFace is a DNN model and enabled us to test the inference performance of different hardware accelerators, such as GPU and IPU (a dedicated AI-chip designed for Machine Learning and Deep Neural Network workloads).

BlazeFace follows the Single-Shot Detection approach which is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of faces in those boxes. It is implemented with TensorFlow and Keras libraries and accelerated with the use of three different hardware accelerators (MK1-IPU, MK2-IPU and V100 Tesla GPU) to exploit its data-level parallelism potential. To maximize the utilization of IPU devices, multiple images were processed in batches concurrently while further optimizations were applied in HOST-DEVICE and DEVICE-HOST communications. A final face-detection speed of 0.107 (ms) and 0.079 (ms) per image is achieved on MK1 and MK2 IPU chips respectively. Furthermore a face-detection speed of 0.012 (ms) per image is achieved when the V100 GPU is used. The V100 GPU manages to outperform both IPUs in terms of image processing speed as the larger available memory (32 GB vs 300 Mib vs 900 Mib) allows larger batch sizes to be used during inference.

Once the face detection step is finished and the location of the face is acquired, we can proceed with the second step of the eyelid closure detection. The followed landmark detection approach to determine the closure of the eye directly from eyelid landmarks, and therefore an algorithm for landmark detection was used. This approach is simple and less computationally expensive compared to other available approaches in the literature. The original landmark detection algorithm, an Ensemble of Regression Trees (ERT), detects 68 facial landmarks, 6 of which are on the eyelid (per eye), which can be used directly to to compute the amount of eyelid closure. A second version of the same algorithm was also investigated which detects only the 6 landmarks on each eye. This 12-point landmark detector is lighter and slightly faster compared to the original 68-point detector as its responsible for predicting only a subset of the total number of facial landmarks. However, the 68-point detector has the advantage that the other landmarks can be used to analyze the movement of other face muscles during the eyeblink-conditioning experiment. Although the ERT algorithm is not well-suited for data-level parallelism acceleration because of its iterative nature, it is already quite fast sequentially (around 1 ms per image - 1000 FPS). The algorithm was accelerated with the multiprocessing python library by exploiting the task-level parallelism. The total number of frames of a video are split up between multiple processes that work concurrently. This resulted in a landmark detection of 0.06 ms per image with 64 processes, and therefore a speedup of 16.6× compared to the original sequential implementation.

The final implementation of detecting the eyelid closure also includes steps such as image loading-decoding from memory (approximately 0.7 ms per image by utilizing multiple processes), data pre-processing and post-processing for face/landmark detection steps. Combining the times for all the above steps resulted in a total eyeblink-response detection time of roughly 1441 FPS on MK2-IPU, 1367 FPS on MK1-IPU and 1658 FPS on

Tesla V100 GPU. The achieved results satisfy the required processing speed of 500 FPS.

An additional aim of this thesis was to evaluate the performance of IPUs and the V100 Tesla GPU on training an image-based CNN face detection model. The training procedure is a complex task which consists of multiple steps that relies on efficient use of both CPUs and device accelerators in order to work properly. The tf.data.Dataset API was used to construct our training datasets. A number of data transformations and optimizations were applied to efficiently utilize available host resources and load batches of data into our accelerator devices. Both IPU-based systems achieve a superior performance compared to the Tesla V100 GPU, especially for small batch-sizes.

Concisely, the following contributions have been made by this thesis:

- Three different face detectors have been evaluated on a subset of the "Annotated Facial Landmarks in the Wild" (AFLW) dataset and on the BioID database. The pre-trained BlazeFace CNN-based model was selected as best suited for this project's requirements.
- The BlazeFace model was accelerated on three hardware accelerators by making use of data-level parallelism. We achieved a speedup of 306× on MK1-IPU chip, a speedup of 414× on MK2-IPU chip and a speedup of 2576× on a V100 Tesla GPU.
- The Ensemble of Regression Trees (ERT) algorithm was selected for landmark-detection from which the amount of eyelid closure is also estimated.
- The ERT landmark detector was accelerated with the multiprocessing python library by making use of multiple processes (maximum number of processes for each CPU). This resulted in a speedup of 15× on the GPU Host, a speedup of 13.6× on the MK1 HOST and a maximum speedup of ×17.7 on the MK2 Host.
- The accelerated face and landmark-detection algorithms were combined and the final eyeblink-response detection algorithm achieves a detection speed of 1658 FPS with Tesla V100 GPU and 32 CPU processes, 1441 FPS with MK2-IPU and 64 processes and 1367 FPS with MK1-IPU and 32 processes. Clearly, all the above satisfies the original requirement of 500 FPS.
- An experimental open-source implementation of the BlazeFace face detector was built from scratch to benchmark the performance of IPU and GPU hardware accelerators on the computationally intensive task of training. Both IPUs have superior performance compared to the GPU.

## 8.2 Discussion

In this study, we compared the performance of the Tesla V100 GPU and the MK1/MK2 IPU chips for machine learning workloads. Our results showed that while the IPU was faster than the GPU for training the BlazeFace network, the GPU was better suited for

serving the trained models in inference scenarios.

The IPU's high MIMD parallelism and ability to efficiently process small batches of data allowed it to achieve faster training times than the GPU. However, the limited on-chip memory of the IPUs is a factor that developers must consider carefully when deploying training or inference experiments. If the size of a model exceeds the available memory of a single IPU chip, developers must partition the model across multiple IPU chips. This process involves dividing the model into smaller sub-models that fit into the memory of each IPU chip, which can be trained in parallel across multiple IPUs. However, partitioning a model is a complex task that requires considerable expertise in parallel programming and distributed systems. Note that, we did not use any partitioning approaches in our case. This is because the BlazeFace detector is a CNN-based model with a small memory footprint due to its lightweight backbone network (inspired by MobileNetV1/V2 [78]) and so it can be compiled and deployed in a single IPU chip.

Our results have important implications for the field of machine learning, image-based applications and neuroscience, highlighting the need for careful selection of hardware platforms for different types of workloads. Ultimately, the choice of hardware platform should be based on careful consideration of factors such as model size, memory requirements, latency requirements and available expertise in parallel programming and distributed systems. Our study showed that while GPUs are still a popular and convenient choice for deploying ML/DL image based workloads, the IPU can offer significant performance advantages in certain scenarios.

One of the main architectural differences between the IPU and GPU is the way they handle parallel processing. IPUs use a MIMD (multiple instruction, multiple data) architecture, which enables them to efficiently process small batches of data in parallel. GPUs, on the other hand, use a SIMD (single instruction, multiple data) architecture, which is better suited for processing large batches of data in parallel.

In addition to the architectural differences, there are other factors that should be considered when selecting a hardware platform for ML/DL workloads. These include factors such as cost, power consumption, and ease of use. In terms of ease of use, GPUs are generally more widely available and well-supported in the machine learning community, making them a more convenient choice for deployment and serving but also more expensive. Many popular machine learning frameworks, such as TensorFlow and PyTorch, have well-established support for GPU acceleration, making it relatively easy to train and serve models on GPU-based hardware.

In contrast, the IPU is a relatively new and specialized architecture which is less widely supported by machine learning frameworks and tools. This could make it more challenging to deploy and serve models on IPU-based hardware, and may require additional engineering effort to optimize performance. Graphcore provides a software development kit (SDK) for their IPU accelerators, which includes libraries and tools for developing and optimizing machine learning applications. The Graphcore SDK is generally user-friendly

with extensive documentation and examples provided to help developers get started. The Graphcore IPU integrates with popular deep learning frameworks like TensorFlow and PyTorch through the high-level APIs provided by the SDK.

In summary, we identify the following advantages and disadvantages of each hardware platform:

**Pros of using GPU-Based systems:**

- Parallelism: GPUs have a SIMD (Single Instruction, Multiple Data) architecture adept at performing large-scale matrix operations commonly found in ML tasks. This makes them especially suitable for deep learning tasks, such as convolutional neural networks
- Ecosystem Maturity: The GPU ecosystem is mature and well-established, offering an abundance of tools, libraries, and community support. Developers can access a vast array of resources, facilitating the discovery of solutions and optimization of ML models for GPU execution.
- Hardware Availability: GPUs are widely available and encompass various performance levels and price points, accommodating different project sizes and budgets.

**Cons of using GPU-Based systems:**

- Memory Hierarchy: The hierarchical memory structure of GPUs necessitates careful management to optimize performance, which can be challenging for developers and may result in suboptimal performance if not appropriately addressed, especially for training ML/DL models.

**Pros of using IPU-based systems:**

- Specialized Architecture: IPUs feature a fine-grained MIMD (Multiple Instruction, Multiple Data) architecture, facilitating the execution of diverse instructions on distinct data elements. This flexibility enables IPUs to achieve faster training times and efficient processing of small batches of data.
- Scalability: IPUs are designed for easy scaling, both within a single device and across multiple devices, allowing developers to build large-scale ML systems with high computational capacity.
- Programmability: IPUs are designed to be more easily programmable, allowing developers to use popular ML frameworks like TensorFlow and PyTorch without the need for extensive low-level optimization. This can lead to faster development cycles and easier adaptation of existing models to the IPU platform.
- Monitor Tools, extensive documentation and code examples in a dedicated GitHub repository.

**Cons of using IPU-based systems:**

- Limited on-chip memory, which requires careful consideration of memory requirements and partitioning of models.
- Ecosystem Maturity: Relatively new and specialized architecture, which may require additional engineering effort to optimize performance.
- Learning Curve: Since IPUs have a unique architecture and programming model, there might be a steeper learning curve when adapting existing code or developing new models for the IPU platform, especially for developers who are already familiar with GPU programming.

## 8.3 Future work

The work conducted for this thesis investigated the combination of a deep learning CNN face detection model and a machine learning landmark detector to create an accurate and fast eyeblink response detection system. The current application showed that with the chosen algorithms, the desired speed of 500 FPS and a satisfactory detection accuracy can be achieved. However, accuracy is a critical factor in blink detection. More challenging recording settings may result in more false positive or false negative detections, leading to inaccurate conclusions about a person's behavior or state. Consequently, one of the directions for future work is to investigate more sophisticated deep learning algorithms for both the face and landmark detection steps to achieve better accuracy. Deep learning models can be trained on large datasets of eye images to learn the patterns and features that distinguish between eye blinks and other eye movements.

In addition, the current setup requires specialized hardware, such as cameras, to be effective. Future work can focus on developing eye blink detection solutions that can be implemented on portable devices, such as smartphones or smart glasses. This would make eye blink detection more accessible and usable in everyday life and provide additional data for research on eye blink patterns and their relation to attention, fatigue, and stress.

Furthermore, the intermediate stages of face and landmark detection used to achieve eyeblink response detection provide intriguing new opportunities for further research. The neuroscientists have expressed an interest in seeing how facial muscles in the face are working during the experiment. A total of 68 facial landmarks are detected using the landmark detection method, which can be used to track the movement of other facial features. In addition, many landmark detectors claim to be able to find up to 192 landmarks, and they can be used if a more accurate analysis of the face is needed for this line of future research.

Another important direction for future work is to evaluate the different hardware

platforms in various real-world scenarios to gain insights into their relative strengths and weaknesses. Experiments can be conducted to compare their performance and suitability in object detection and recognition tasks, medical image analysis, and natural language processing. This knowledge can then be used to develop more efficient and effective software and hardware solutions, ultimately advancing the field of machine learning and improving its practical applications.

# Παραρτήματα

# Appendix A

# Appendix

**FDDB Dataset**

| Device | batch_Size | time/epoch (sec) | time/step (ms) | total_training_time | loss | val_loss |
|---|---|---|---|---|---|---|
| V100 | 2 | 13-14 | 14-14 | 2024.504 | 0.1468 | 0.2553 |
| V100 | 4 | 6-7 | 14-15 | 1051.215 | 0.1599 | 0.2676 |
| V100 | 8 | 4 | 17-18 | 649.135 | 0.1298 | 0.2625 |
| V100 | 16 | 3-4 | 26-32 | 508.300 | 0.1789 | 0.3201 |
| V100 | 32 | 3 | 42-45 | 421.285 | 0.2157 | 0.4142 |
| V100 | 64 | 2 | 70-72 | 348.398 | 0.3046 | 0.6090 |
| 2xMK1 | 2x1 | 2-3 | 2-3 | 265,823 | 0.1826 | 0.2671 |
| 2xMK1 | 2x2 | 2-3 | 3-4 | 240,676 | 0.1805 | 0.2679 |
| 2xMK1 | 2x4 | 2-3 | 6-8 | 243,615 | 0.1971 | 0.3362 |
| 2xMK1 | 2x8 | OOM | - | - | - | - |
| MK2 | 2 | 5-6 | 5-6 | 602.478 | 0.1698 | 0.2649 |
| MK2 | 4 | 3-4 | 3-5 | 545.570 | 0.1512 | 0.2694 |
| MK2 | 8 | 3 | 7-8 | 430,255 | 0.1306 | 0.2764 |
| MK2 | 16 | 3 | 12-13 | 423.668 | 0.1459 | 0.3184 |
| MK2 | 32 | OOM | - | - | - | - |

**Table A.1.** *Training BlazeFace on FDDB dataset, epochs = 150, steps_per_epoch = (num_-of_samples // batch_size), Out of Memory (OOM) declares that the model could not fit inside the local memory of the IPU*

### 300W-LP Dataset

| Device | Batch_Size | time/epoch (sec) | time/step (ms) | total_training_time | loss | val_loss |
|---|---|---|---|---|---|---|
| V100 | 8 | 96-112 | 15-17 | 9908.442 | 0.0529 | 0.0293 |
| V100 | 16 | 64-69 | 21-22 | 6610.478 | 0.0577 | 0.0357 |
| V100 | 32 | 36-38 | 24-26 | 3746.590 | 0.0520 | 0.0272 |
| V100 | 64 | 25-27 | 33-37 | 2670.356 | 0.0502 | 0.0264 |
| V100 | 128 | 19-21 | 51-56 | 2059.075 | 0.0420 | 0.0291 |
| 2xMK1 | 2x1 | 18 | 0.6-0.7 | 1803,62 | 0.0777 | 0.0422 |
| 2xMK1 | 2x2 | 11-12 | 0.8-0.9 | 1163,5 | 0.0594 | 0.0328 |
| 2xMK1 | 2x4 | 8 | 1 | 843,959 | 0.0543 | 0.0261 |
| 2xMK1 | 2x8 | OOM | - | - | - | - |
| MK2 | 2 | 29 | 1 | 2965.834 | 0.0503 | 0.0413 |
| MK2 | 4 | 18 | 1 | 1824,855 | 0.0601 | 0.0361 |
| MK2 | 8 | 13 | 2 | 1368.401 | 0.0592 | 0.0304 |
| MK2 | 16 | 10 | 3 | 1011.710 | 0.0497 | 0.0271 |
| MK2 | 32 | OOM | - | - | - | - |

**Table A.2.** *Training BlazeFace on 300w-LP dataset for 100 epochs, steps_per_epoch = (num_of_samples // batch_size), Out of Memory (OOM) declares that the model could not fit inside the local memory of the IPU*

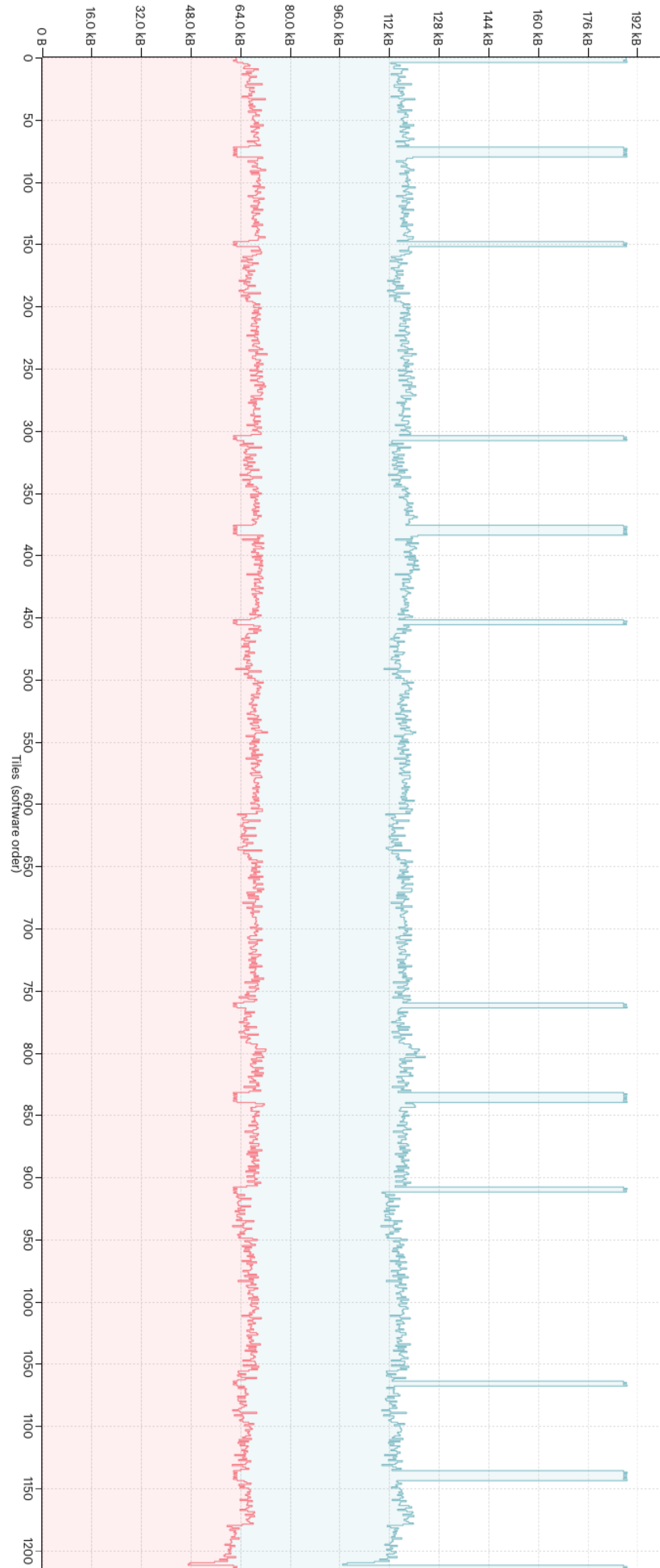



**Figure A.1.** *The (maximum) memory consumption (vertical axis) across the 1216 tiles (horizontal axis) of a MK1 IPU for a batch size of 32. The horizontal line denotes the maximum available tile memory. The pink area represents always live memory, whereas the blue indicates maximum transient memory use on a tile level.*

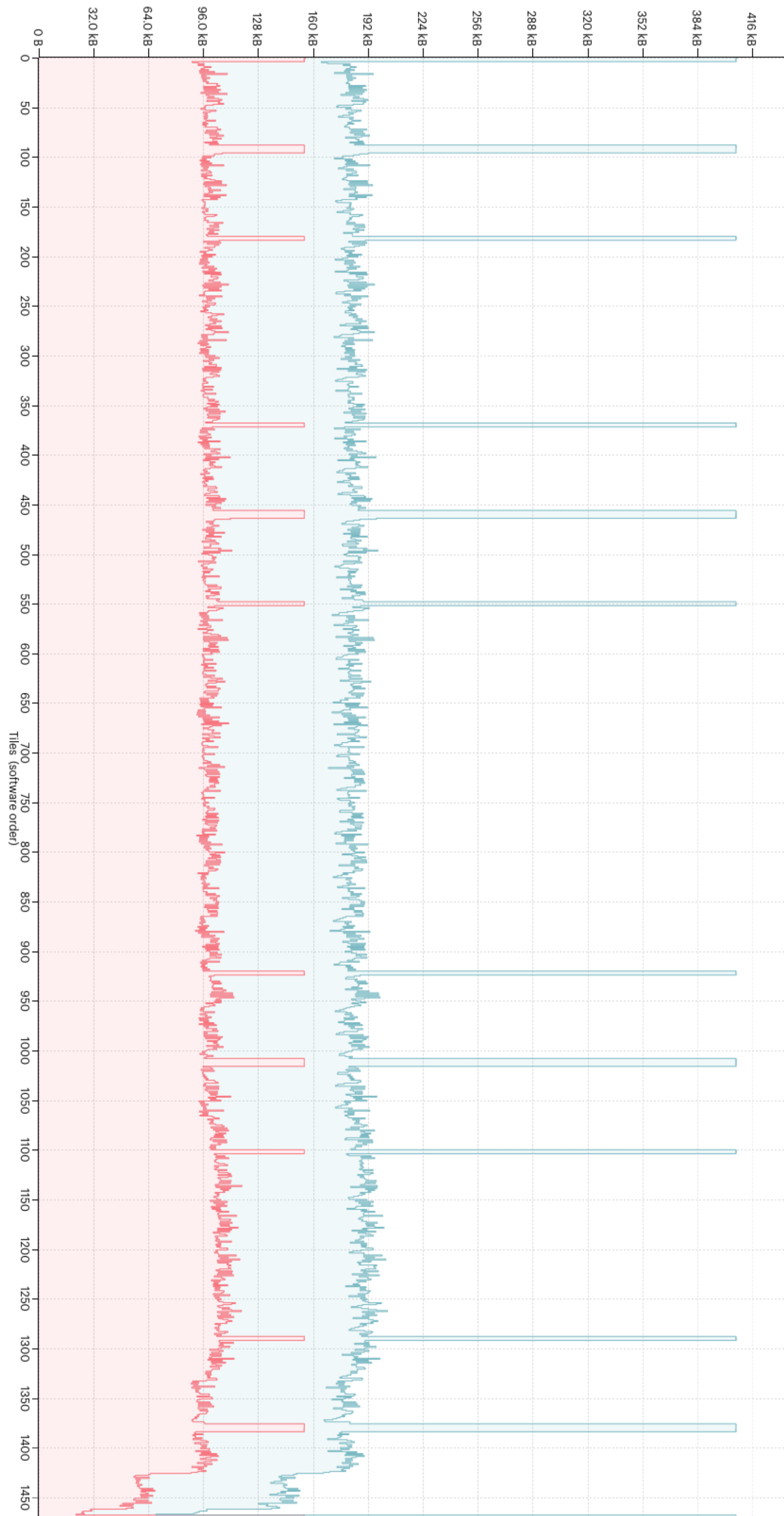



**Figure A.2.** *The (maximum) memory consumption (vertical axis) across the 1472 tiles (horizontal axis) of a MK2 IPU for a batch size of 64. The horizontal line denotes the maximum available tile memory. The pink area represents always live memory, whereas the blue indicates maximum transient memory use on a tile level.*

# Bibliography

[1] Henk Jan Boele. *Neural Mechanisms underlying Motor Learning*. Διδακτορική Διατριβή, 2014.

[2] John P. Welsh και Jeffrey T. Oristaglio. *Autism and Classical Eyeblink Conditioning: Performance Changes of the Conditioned Response Related to Autism Spectrum Disorder Diagnosis*. *Frontiers in Psychiatry*, 7, 2016.

[3] Charles Laidi, Carole Levenes, Alex Suarez-Perez, Caroline Février, Florence Durand, Noomane Bouaziz και Dominique Januel. *Cognitive Impact of Cerebellar Non-invasive Stimulation in a Patient With Schizophrenia*. *Frontiers in Psychiatry*, 11, 2020.

[4] J.P. Lewis. *Fast Template Matching*. *Vis. Interface*, 95, 1994.

[5] Yann LeCun, Y. Bengio και Geoffrey Hinton. *Deep Learning*. *Nature*, 521:436–44, 2015.

[6] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg και Li Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge*, 2015.

[7] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn και A. Zisserman. *The Pascal Visual Object Classes Challenge: A Retrospective*. *International Journal of Computer Vision*, 111(1):98–136, 2015.

[8] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu και Matti Pietikäinen. *Deep Learning for Generic Object Detection: A Survey*, 2019.

[9] Shervin Minaee, Yuri Y. Boykov, Fatih Porikli, Antonio J Plaza, Nasser Kehtarnavaz και Demetri Terzopoulos. *Image Segmentation Using Deep Learning: A Survey*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, σελίδες 1–1, 2021.

[10] Valentin Bazarevsky, Yury Kartynnik, Andrey Vakunov, Karthik Raveendran και Matthias Grundmann. *BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs*, 2019.

[11] Navneet Dalal και Bill Triggs. *Histograms of oriented gradients for human detection*. *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, τόμος 1, σελίδες 886–893. IEEE, 2005.

[12] Paul Viola και Michael J Jones. *Robust real-time face detection*. *International journal of computer vision*, 57(2):137–154, 2004.

[13] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu και Alexander C. Berg. *SSD: Single Shot MultiBox Detector*. *Lecture Notes in Computer Science*, σελίδα 21–37, 2016.

[14] Joseph Redmon, Santosh Divvala, Ross Girshick και Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*, 2016.

[15] Shaoqing Ren, Kaiming He, Ross Girshick και Jian Sun. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, 2016.

[16] Jan Hosang, Rodrigo Benenson και Bernt Schiele. *Learning non-maximum suppression*, 2017.

[17] Sparsh Mittal, Poonam Rajput και Sreenivas Subramoney. *A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations*. *IEEE Transactions on Neural Networks and Learning Systems*, σελίδες 1–21, 2021.

[18] *NVIDIA CUDA C programming guide, Release: 11.7.0*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2022.

[19] Leslie G. Valiant. *A bridging model for parallel computation*. *Commun. ACM*, 33:103–111, 1990.

[20] Farhana Sultana, Abu Sufian και Paramartha Dutta. *Advancements in Image Classification using Convolutional Neural Network*. *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, 2018.

[21] Florian Schroff, Dmitry Kalenichenko και James Philbin. *FaceNet: A unified embedding for face recognition and clustering*. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[22] Yash Srivastava, Vaishnav Murali και Shiv Ram Dubey. *Hard-Mining Loss based Convolutional Neural Network for Face Recognition*, 2020.

[23] Shervin Minaee, Ping Luo, Zhe Lin και Kevin Bowyer. *Going Deeper Into Face Detection: A Survey*, 2021.

[24] Shifeng Zhang, Xiangyu Zhu, Zhen Lei, Hailin Shi, Xiaobo Wang και Stan Z. Li. *$S^3$FD: Single Shot Scale-invariant Face Detector*, 2017.

[25] Jiankang Deng, Jia Guo, Yuxiang Zhou, Jinke Yu, Irene Kotsia και Stefanos Zafeiriou. *RetinaFace: Single-stage Dense Face Localisation in the Wild*, 2019.

[26] Ross Girshick, Jeff Donahue, Trevor Darrell και Jitendra Malik. *Rich feature hierarchies for accurate object detection and semantic segmentation*, 2014.

[27] Jasper Uijlings, K. Sande, T. Gevers και A.W.M. Smeulders. *Selective Search for Object Recognition*. *International Journal of Computer Vision*, 104:154–171, 2013.

[28] Chenchen Zhu, Yutong Zheng, Khoa Luu και Marios Savvides. *CMS-RCNN: Contextual Multi-Scale Region-based CNN for Unconstrained Face Detection*, 2016.

[29] Jifeng Dai, Yi Li, Kaiming He και Jian Sun. *R-FCN: Object Detection via Region-based Fully Convolutional Networks*, 2016.

[30] Changzheng Zhang, Xiang Xu και Dandan Tu. *Face Detection Using Improved Faster RCNN*, 2018.

[31] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng και Rong Qu. *A Survey of Deep Learning-Based Object Detection*. *IEEE Access*, 7:128837–128868, 2019.

[32] Yizhou Wang. *Binary Neural Networks for Object Detection*, 2019.

[33] Davis E. King. *Dlib-ml: A Machine Learning Toolkit*. *Journal of Machine Learning Research*, 10:1755–1758, 2009.

[34] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li και Yu Qiao. *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*. *IEEE Signal Processing Letters*, 23(10):1499–1503, 2016.

[35] Peter M. Roth Martin Koestinger, Paul Wohlhart και Horst Bischof. *Annotated Facial Landmarks in the Wild: A Large-scale, Real-world Database for Facial Landmark Localization*. *Proc. First IEEE International Workshop on Benchmarking Facial Image Analysis Technologies*, 2011.

[36] Oliver Jesorsky, Klaus J Kirchberg και Robert W Frischholz. *Robust face detection using the hausdorff distance*. *International Conference on Audio-and Video-Based Biometric Person Authentication*, σελίδες 90–95. Springer, 2001.

[37] Paul Bakker, Henk Jan Boele, Zaid Al-Ars και Christos Strydis. *Real-Time Face and Landmark Localization for Eyeblink Detection*, 2020.

[38] Tereza Soukupova και Jan Cech. *Real-Time Eye Blink Detection using Facial Landmarks*. *21st Computer Vision Winter Workshop*, 2016.

[39] Yue Wu και Qiang Ji. *Facial Landmark Detection: A Literature Survey*. *International Journal of Computer Vision*, 127(2):115–142, 2018.

[40] Vahid Kazemi και Josephine Sullivan. *One millisecond face alignment with an ensemble of regression trees*. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, σελίδες 1867–1874, 2014.

[41] Kostiantyn Khabarlak και Larysa Koriashkina. *Fast Facial Landmark Detection and Applications: A Survey*. *Journal of Computer Science and Technology*, 22(1):e02, 2022.

[42] Xin Jin και Xiaoyang Tan. *Face alignment in-the-wild: a survey*. *arXiv preprint arXiv:1608.04188*, 2016.

[43] *Live ML anywhere*. https://mediapipe.dev/. Accessed: 2022-07-03.

[44] *Graphcore Documentation*. https://docs.graphcore.ai/en/latest/. Accessed: 2022-07-05.

[45] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan και Zhenyao Zhu. *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*, 2015.

[46] Stefanos Zafeiriou, Cha Zhang και Zhengyou Zhang. *A survey on face detection in the wild: past, present and future*. *Computer Vision and Image Understanding*, 138:1–24, 2015.

[47] Vidit Jain και Erik Learned-Miller. *FDDB: A Benchmark for Face Detection in Unconstrained Settings*. Τεχνική Αναφορά με αριθμό UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[48] Xiangyu Zhu, Xiaoming Liu, Zhen Lei και Stan Z. Li. *Face Alignment in Full Pose Range: A 3D Total Solution*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(1):78–92, 2019.

[49] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy και Ping Tak Peter Tang. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*, 2017.

[50] Xiaoxin He, Fuzhao Xue, Xiaozhe Ren και Yang You. *Large-Scale Deep Learning Optimizations: A Comprehensive Survey*, 2021.

[51] Yanli Liu, Yuan Gao και Wotao Yin. *An Improved Analysis of Stochastic Gradient Descent with Momentum*. *Advances in Neural Information Processing Systems*H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan και H. Lin, επιμελητές, τόμος 33, σελίδες 18261–18271. Curran Associates, Inc., 2020.

[52] Diederik P. Kingma και Jimmy Ba. *Adam: A Method for Stochastic Optimization*, 2017.

[53] Ivan Petrovich Pavlov και Gleb Vasīlevīch Anrep. *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex*. Oxford University Press: Humphrey Milford, 1927.

[54] D. Amodei και D. Hernandez. *AI and compute*. https://openai.com/blog/ai-and-compute/, 2018.

[55] V Bracha, L Zhao, DA Wunderlich, SJ Morrissy και JR Bloedel. *Patients with cerebellar lesions cannot acquire but are able to retain conditioned eyeblink reflexes*. *Brain: a journal of neurology*, 120(8):1401–1413, 1997.

[56] E James Kehoe, Elliot A Ludvig, Joanne E Dudeney, James Neufeld και Richard S Sutton. *Magnitude and timing of nictitating membrane movements during classical conditioning of the rabbit (Oryctolagus cuniculus)*. *Behavioral Neuroscience*, 122(2):471, 2008.

[57] Musyarofah , Valentina Schmidt και Martin Kada. *Object detection of aerial image using mask-region convolutional neural network (mask R-CNN)*. *IOP Conference Series: Earth and Environmental Science*, 500:012090, 2020.

[58] Y. Lecun, L. Bottou, Y. Bengio και P. Haffner. *Gradient-based learning applied to document recognition*. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[59] Ning Qian. *On the momentum term in gradient descent learning algorithms*. *Neural Networks*, 12(1):145–151, 1999.

[60] Moises Hernandez Fernandez, Ginés Guerrero, José Cecilia, José García, Alberto Inuggi, Saad Jbabdi, Timothy Behrens και Stamatios Sotiropoulos. *Erratum: Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using GPUs (PLoS ONE (2015) 10:6 (e0130915) 10.1371/journal.pone.0130915)*. *PLoS ONE*, 10, 2015.

[61] Xinxin Mei και Xiaowen Chu. *Dissecting GPU Memory Hierarchy Through Microbenchmarking*. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[62] Shin Ying Lee και Carole Jean Wu. *Characterizing the latency hiding ability of GPUs*. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, σελίδες 145–146, 2014.

[63] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu και Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, 2016.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai και Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, 2019.

[65] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama και Trevor Darrell. *Caffe: Convolutional Architecture for Fast Feature Embedding*, 2014.

[66] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro και Evan Shelhamer. *cuDNN: Efficient Primitives for Deep Learning*, 2014.

[67] *NVIDIA CUDA-X*. https://developer.nvidia.com/gpu-accelerated-libraries. Accessed: 2022-02-15.

[68] François Chollet και others. *Keras*. https://github.com/fchollet/keras, 2015.

[69] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt και Gang Hua. *A convolutional neural network cascade for face detection*. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, σελίδες 5325–5334, 2015.

[70] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester και Deva Ramanan. *Object Detection with Discriminatively Trained Part-Based Models*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[71] Gary B. Huang, Manu Ramesh, Tamara Berg και Erik Learned-Miller. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*. Τεχνική Αναφορά με αριθμό 07-49, University of Massachusetts, Amherst, 2007.

[72] *Python Package Index - PyPI*. https://pypi.org/.

[73] Dilpreet Singh Brar, Amit Kumar, Pallavi, Usha Mittal και Pooja Rana. *Face Detection for Real World Application*. *2021 2nd International Conference on Intelligent Engineering and Management (ICIEM)*, σελίδες 239–242, 2021.

[74] Kaiming He, Xiangyu Zhang, Shaoqing Ren και Jian Sun. *Deep Residual Learning for Image Recognition*, 2015.

[75] Fei Yang, Qian Zhang, Miaohui Wang και Guoping Qiu. *Quality Classified Image Analysis with Application to Face Detection and Recognition*, 2018.

[76] Vuong Le, Jonathan Brandt, Zhe Lin, Lubomir Bourdev και Thomas S Huang. *Interactive facial feature localization*. *European Conference on Computer Vision*, σελίδες 679–692. Springer, 2012.

[77] Xiaojie Guo, Siyuan Li, Jinke Yu, Jiawan Zhang, Jiayi Ma, Lin Ma, Wei Liu και Haibin Ling. *PFLD: A Practical Facial Landmark Detector*, 2019.

[78] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov και Liang Chieh Chen. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*, 2019.

[79] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto και Hartwig Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, 2017.

[80] Christos Sagonas, Georgios Tzimiropoulos, Stefanos Zafeiriou και Maja Pantic. *300 faces in-the-wild challenge: The first facial landmark localization challenge*. *Proceedings of the IEEE International Conference on Computer Vision Workshops*, σελίδες 397–403, 2013.

[81] Christos Sagonas, Epameinondas Antonakos, Georgios Tzimiropoulos, Stefanos Zafeiriou και Maja Pantic. *300 Faces In-The-Wild Challenge: database and results*. *Image and Vision Computing*, 47:3–18, 2016. 300-W, the First Automatic Facial Landmark Detection in-the-Wild Challenge.

[82] P Umesh. *Image Processing in Python*. *CSI Communications*, 23, 2012.

[83] Davis E. King. *A Global Optimization Algorithm Worth Using*. http://blog.dlib.net/2017/12/a-global-optimization-algorithm-worth.html. Accessed: 2022-07-05.

[84] Cédric Malherbe και Nicolas Vayatis. *Global optimization of Lipschitz functions*, 2017.

[85] YoungJoon Yoo, Dongyoon Han και Sangdoo Yun. *EXTD: Extremely Tiny Face Detector via Iterative Filter Reuse*, 2019.

[86] Ross Girshick. *Fast R-CNN*, 2015.

[87] Luis Perez και Jason Wang. *The Effectiveness of Data Augmentation in Image Classification using Deep Learning*, 2017.

[88] Agnieszka Mikołajczyk και Michał Grochowski. *Data augmentation for improving deep learning in image classification problem*. *2018 International Interdisciplinary PhD Workshop (IIPhDW)*, σελίδες 117–122, 2018.

[89] Joseph Lemley, Shabab Bazrafkan και Peter Corcoran. *Smart Augmentation Learning an Optimal Data Augmentation Strategy*. *IEEE Access*, 5:5858–5869, 2017.

[90] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala και Vijay Chidambaram. *Analyzing and Mitigating Data Stalls in DNN Training*, 2021.

[91] Xiang Wang, Kai Wang και Shiguo Lian. *A survey on face data augmentation for the training of deep neural networks*. *Neural Computing and Applications*, 32(19):15503–15531, 2020.