ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

FPGA Acceleration of Generative Adversarial Networks for Image Reconstruction

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνωστόπουλος Χ. Κωνσταντίνος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Αθήνα, Σεπτέμβριος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

FPGA Acceleration of Generative Adversarial Networks for Image Reconstruction

---

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνωστόπουλος Χ.
Κωνσταντίνος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Σεπτεμβίου 2020.

.....	.....	.....
Σούντρης Δημήτριος Καθηγητής	Τσανάκας Παναγιώτης Καθηγητής	Διονύσιος Πνευματικάτος Καθηγητής

Αθήνα, Σεπτέμβριος 2020.

.....
Αναγνωστόπουλος Χ. Κωνσταντίνος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα τελευταία χρόνια, οι εφαρμογές της μηχανικής μάθησης έρχονται ολοένα και περισσότερο στο προσκήνιο. Απο τις διάφορες τεχνικές που εφαρμόζονται, προκειμένου οι υπολογιστές να δύνανται να μαθαίνουν από εμπειρίες, τα νευρωνικά δίκτυα αποτελούν μία από τις σημαντικότερες. Εμπνευσμένα από τους βιολογικούς εγκεφάλους έμβιων όντων, τα νευρωνικά δίκτυα διαθέτουν μεγάλο βαθμό πολυπλοκότητας και, ακόμα και για μοντέρνους επεξεργαστές, η εκπαίδευση τους καθώς και ο υπολογισμός των εξόδων τους αποτελούν σημαντική πρόκληση. Για την επιτάχυνση αυτών των διαδικασιών χρησιμοποιείται ειδικό hardware (GPUs, TPUs, NPUs, FPGAs) που επιτρέπει την παραλληλοποίηση συγκεκριμένων υπολογισμών με στόχο την ταχύτερη περάτωση τους.
Στο πρώτο μέρος της παρούσας εργασίας χρησιμοποιείται μία νέα αρχιτεκτονική νευρωνικών δικτύων, η οποία προτάθηκε από τον Ian Goodfellow και τους συνεργάτες του το 2015 και η οποία καλείται Generative Adversarial Networks για την ανακατασκευή ημιτελών εικόνων από το MNIST dataset. Τα Generative Adversarial Networks είναι δύο νευρωνικά δίκτυα τα οποία «παίζουν» μεταξύ τους ένα παιχνίδι όπου η βελτίωση του ενός στο ρόλο που έχει αναλάβει στο συγκεκριμένο παιχνίδι συνδράμει και στη βελτίωση του άλλου στο δικό του ρόλο. Με την τεχνική αυτή, γεννητικές διαδικασίες όπως αυτή της ανακατασκευής μίας εικόνας μπορούν να αντιμετωπιστούν ως διακριτικές διαδικασίες όπως αυτή της αναγνώρισης ενός ψηφίου. Πριν την ανάπτυξη του μοντέλου που επιτυγχάνει την ανακατασκευή, γίνεται πειραματισμός με την τεχνική των Generative Adversarial Networks με την ανάπτυξη μοντέλου ικανού να συνθέτει εκ νέου εικόνες σαν αυτές του MNIST dataset, ακολουθώντας τα βήματα που περιγράφονται στην εργασία του Goodfellow.
Στο δεύτερο μέρος, επιχειρείται η επιτάχυνση του forward propagation του τελικού μοντέλου που φέρει τη δυνατότητα ανακατασκευής εικόνων με τη χρήση FPGA, του οποίου ο προγραμματισμός γίνεται με High-level Synthesis περιγραφή. Η εκπαιδευτική πλατφόρμα που χρησιμοποιείται φέρει το chip Z-7010 της Xilinx, και καθότι εκπαιδευτική, διαθέτει περιορισμένους υλικούς πόρους για την επίτευξη της επιτάχυνσης. Το γεγονός αυτό αποτέλεσε σημαντικότατη πρόκληση καθώς η χρήση των διαθέσιμων υλικών πόρων του FPGA ήταν απαραίτητο να γίνει με ευλάβεια. Επιπλέον, αφού το τελικό δεσιγν ολοκληρωθεί, ακολουθείται σύγκριση της ταχύτητας περάτωσης των υπολογισμών με διάφορους επεξεργαστές πέρα από τον ARM cortex-A9 που διατίθεται στην εκπαιδευτική πλατφόρμα. Τέλος, παρουσιάζονται διεξοδικά οι επιπτώσεις της επιλογής διαφορετικού πλήθους bit για την αναπαράσταση των δεκαδικών αριθμών εντός του FPGA στην ακρίβεια των αποτελεσμάτων καθώς και στην χρήση των διαθέσιμων υλικών πόρων που διαθέτει το chip Z-7010.

Λέξεις κλειδιά : μηχανική μάθηση, νευρωνικά δίκτυα, βαθιά μάθηση, γεννητικά ανταγωνιστικά δίκτυα, ανακατασκευή εικόνας, FPGA, σύνθεση υψηλού επιπέδου, επιτάχυνση υλικού

Abstract

In the recent years, machine learning applications become increasingly popular. From the various techniques that are used to enable computers to learn from experience, neural networks are one of the most important. Inspired by the biological brains of living entities, neural networks posse great complexity and even modern CPUs struggle to deal with their training and the computation of their outputs. For the acceleration of these operations, dedicated hardware is used (GPUs, TPUs, NPUs, FPGAs) that enables the parallelization of certain computations in order to achieve faster completion of them.

In the first part of the present thesis, a new architecture of neural networks is used, proposed by Ian Goodfellow and his colleagues at 2015, which is called Generative Adversarial Networks and is used for the reconstruction of half-cut MNIST images. Generative Adversarial Networks consist of two neural networks that contest each other in a game. The improvement of the one network in its roll in the game contributes to the improvement of the other network in its own roll. With this technique, generative processes like image reconstruction can be dealt as discrimination processes like the recognition of a handwritten digit. Before the development of the network that achieves the image reconstruction, experimentation with Generative Adversarial Network technique takes place with the development of a model capable of generating new images of handwritten digits like the those in the MNIST dataset, following the steps that are described in Goodfellow's paper.

In the second part, the acceleration of the forward propagation of the final model, that can reconstruct images, is attempted with the use of an FPGA which is programmed using High-level Synthesis description. The educational platform that is used has the Z-7010 Xilinx chip. Being educational, the chip possesses limited amount of hardware recourses for the acceleration to take place. This fact was a serious challenge as the use of the available hardware resources had to be done carefully. Furthermore, after the final design is completed, a performance comparison between the FPGA and other CPUs, apart from the ARM cortex-A9 which is available on the educational platform, is done. Finally, the consequences at the results' precision and the hardware resources utilization, of different bit-width choices for representing decimal numbers in the programmable logic are presented thoroughly.

Keywords : machine learning, neural networks, deep learning, generative adversarial networks, image reconstruction, FPGA, high-level synthesis, hardware acceleration

Ευχαριστίες

Ξεκινώντας, θα ήθελα να ευχαριστήσω τον κύριο Δημήτριο Σούντρη που η καθοδήγηση του οδήγησε στην επιλογή της παρούσας διπλωματικής εργασίας. Επιπλέον, θα ήθελα να ευχαριστήσω θερμά τον Διδακτορικό ερευνητή Δημήτριο Δανόπουλο για τη συνεχή βοήθεια του και την απλόχερη προσφορά γνώσεων και πληροφοριών που συνέβαλαν σημαντικά στην εκπόνηση της παρούσας εργασίας.

Τέλος, καθότι η μελέτη και ο πειραματισμός για την εκπόνηση αυτής της εργασίας πραγματοποιήθηκαν σε μία περίοδο που χαρακτηρίστηκε έντονα από τη μη κανονικότητα της, με όλες τις ψυχολογικές επιπτώσεις που αυτό συνεπάγεται, θέλω να εκφράσω την αμέριστη ευγνωμοσύνη μου αρχικά στην οικογένεια μου και ύστερα στους φίλους μου οι οποίοι στάθηκαν στο πλευρό μου.

Περιεχόμενα

Μέρος 1

Εύρεση & Ανάπτυξη Μοντέλου

1.1 Machine Learning

1.1.1 Ορισμός

Η μηχανική μάθηση αποτελεί κομμάτι της επιστήμης των υπολογιστών. Είναι η επιστήμη του προγραμματισμού υπολογιστών με τέτοιο τρόπο ώστε να μαθαίνουν από τα δεδομένα. Σύμφωνα με τον Arthur Samuel, Αμερικανό πρωτοπόρο σε θέματα τεχνητής νοημοσύνης [1]:

" Η μηχανική μάθηση είναι το πεδίο μελέτης που δίνει στους υπολογιστές τη δυνατότητα να μαθαίνουν χωρίς να έχουν ρητά προγραμματιστεί»

Ένας λιγότερο αφηρημένος ορισμός της μηχανικής μάθησης δίνεται από τον Tom Mitchell, Αμερικανό επιστήμονα υπολογιστών, όπου αναφέρει:[1]

«Ένα πρόγραμμα υπολογιστή λέγεται ότι μαθαίνει από μία εμπειρία E σχετικά με ένα έργο T και μία μετρική απόδοσης P, αν η απόδοση του στο T, όπως μετριέται από την μετρική P, βελτιώνεται από την εμπειρία E».

1.1.2 Κατηγοριοποίηση Μεθόδων Μηχανικής Μάθησης

Τα συστήματα μηχανικής μάθησης μπορούν σε γενικές γραμμές να κατηγοριοποιηθούν με τους εξής τρόπους[1]:

- Βάση του αν η εκμάθηση τους γίνεται με επίβλεψη από τον άνθρωπο (Επιβλεπόμενη μάθηση, Μη επιβλεπόμενη μάθηση, Ημί-επιβλεπόμενη μάθηση, Ενισχυτική μάθηση).
- Βάση του αν η εκμάθηση γίνεται σταδιακά ή επί τόπου (Batch learning ή Online learning).
- Βάση του τρόπου λειτουργίας τους, δηλαδή είτε συγκρίνοντας νέα δεδομένα με ήδη γνωστά είτε ανακαλύπτοντας μοτίβα σε δεδομένα εκμάθησης, χτίζοντας έτσι ένα μοντέλο πρόβλεψης. (Instance based ή model based learning).

Όσον αφορά τον πρώτο τρόπο κατηγοριοποίησης, παρουσιάζονται οι βασικές μέθοδοι εκμάθησης με περισσότερη λεπτομέρεια:

- Online Machine Learning: Η μέθοδος μηχανικής μάθησης κατά την οποία τα δεδομένα γίνονται διαθέσιμα σειριακά και χρησιμοποιούνται για τη σταδιακή βελτίωση του εκάστοτε μοντέλου.
- Batch Machine Learning: Η μέθοδος κατά την οποία η εκμάθηση του εκάστοτε μοντέλου γίνεται πάνω σε ολόκληρο το σετ δεδομένων.

Τέλος, για τη διαφοροποίηση βάση του τρόπου λειτουργίας οι κατηγορίες αναλυτικότερα είναι:

- Instance Based Learning: Στην κατηγορία αυτή ανήκουν οι αλγόριθμοι μηχανικής μάθησης, οι οποίοι προκειμένου να παράγουν κάποια έξοδο, συγκρίνουν τα δεδομένα στην είσοδο τους με ήδη υπάρχοντα δεδομένα που βρίσκονται στο σετ εκπαίδευσης.

- Model Based Learning: Σε αντίθεση με την προηγούμενη περίπτωση, δημιουργείται ένα μοντέλο με στόχο τη γενίκευση του συνόλου των δοσμένων δεδομένων-παραδειγμάτων προκειμένου να γίνονται προβλέψεις με νέα δεδομένα ως είσοδο.

1.1.3 Χρησιμότητα

Οι τεχνικές μηχανικής μάθησης αναδεικνύουν τη χρησιμότητα τους σε περιπτώσεις όπου τα δεδομένα ακολουθούν κάποιο μοτίβο, το οποίο διαφοροποιείται με την πάροδο του χρόνου ή είναι ιδιαίτερα πολύπλοκο με αποτέλεσμα να χρειάζεται πολυάριθμους υπολογισμούς πέρα από την αντίληψη μας. Σε τέτοιες περιπτώσεις, είναι αδύνατο για τους προγραμματιστές να προγραμματίσουν ρητά κάποια λύση ενώ οι τεχνικές μηχανικής μάθησης με την ικανότητα τους να «μαθαίνουν» από την εμπειρία μπορούν να προσφέρουν τις λύσεις αυτές.

1.2 Θεωρητικό Υπόβαθρο Νευρωνικών Δικτύων

1.2.1 Νευρωνικά Δίκτυα

Τα τεχνητά νευρωνικά δίκτυα, αποτελούν υπολογιστικά συστήματα εμπνευσμένα από τα βιολογικά νευρωνικά δίκτυα, τα οποία συνθέτουν τους εγκεφάλους των ζώων. [2]
Ώθηση στη μελέτη των τεχνιτών νευρωνικών δικτύων δόθηκε από την αναγνώριση ότι ο ανθρώπινος εγκέφαλος πραγματοποιεί υπολογισμούς με τρόπο εντελώς διαφορετικό σε σχέση με έναν ηλεκτρονικό υπολογιστή. Με βασική διαφορά την παράλληλη επεξεργασία πληροφορίας, έχει την δυνατότητα να οργανώνει τη δομή του, δηλαδή τους νευρώνες που τον αποτελούν, με τέτοιο τρόπο ώστε να επιτυγχάνει την ταχύτατη ολοκλήρωση συγκεκριμένων υπολογισμών. Οι υπολογισμοί αυτοί αφορούν διεργασίες όπως αυτή της όρασης και της ακοής καθώς και κάθε άλλης νοητικής λειτουργίας. [3]
Όπως τα φυσικά νευρωνικά δίκτυα αποτελούνται από ένας πλήθος διασυνδεδεμένων νευρώνων, οι οποίοι λειτουργούν ως τη δομική μονάδα επεξεργασίας πληροφορίας, έτσι και τα τεχνητά νευρωνικά δίκτυα αποτελούνται από ένα σύνολο τεχνητών νευρώνων, ενώ είναι σχεδιασμένα έτσι ώστε να μοντελοποιούν τον τρόπο που ο εγκέφαλος ολοκληρώνει μία συγκεκριμένη διεργασία. Επιπλέον, όπως τα φυσικά νευρωνικά δίκτυα έχουν τη δυνατότητα αναδιοργάνωσης της δομής τους με στόχο την ολοκλήρωση μίας διεργασίας, όπως προαναφέρθηκε, έτσι και τα τεχνητά νευρωνικά δίκτυα δύνανται να περάσουν από μία διαδικασία εκμάθησης όπου ύστερα από αυτή θα έχουν πραγματοποιηθεί σε αυτά οι κατάλληλες τροποποιήσεις με στόχο την ορθή ολοκλήρωση κάποιας διεργασίας. [3]
Βλέποντας τα νευρωνικά δίκτυα ως μία μηχανή με δυνατότητα προσαρμογής, ο εξής ορισμός μπορεί να δοθεί: «Ένα νευρωνικό δίκτυο αποτελεί έναν μεγάλης κλίμακας παράλληλα κατανεμημένο επεξεργαστή, ο οποίος αποτελείται από απλές επεξεργαστικές μονάδες, τους νευρώνες, οι οποίες έχουν τη φυσική ικανότητα να αποθηκεύουν εμπειρική γνώση και να την καθιστούν διαθέσιμη για χρήση. Ομοιάζουν με τον εγκέφαλο με τους εξής δύο τρόπους [3]:

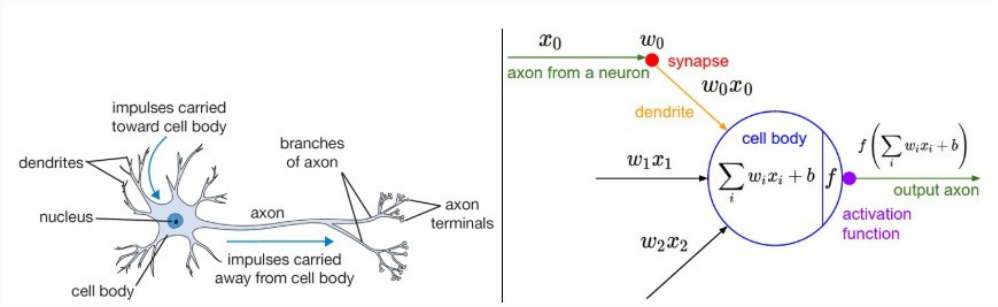
1. Η γνώση αποκτάται από το περιβάλλον μέσω μίας διαδικασίας μάθησης.

2. Οι δυνάμεις των συνδέσεων μεταξύ των νευρώνων, γνωστές και ως συναπτικά βάρη, χρησιμοποιούνται για την αποθήκευση της αποκτώμενης γνώσης. »

1.2.2 Δομή & Τρόπος Λειτουργίας

Όπως προαναφέρθηκε, η θεμελιώδης υπολογιστική μονάδα ενός νευρωνικού δικτύου είναι ο νευρώνας. Η λειτουργία ενός τεχνητού νευρώνα μπορεί άμεσα να παρομοιαστεί με τη λειτουργία ενός βιολογικού νευρώνα. Στην περίπτωση του βιολογικού μοντέλου, κάθε νευρώνας δέχεται σήματα εισόδου από τους δενδρίτες και παράγει την έξοδό του κατά μήκος του άξονά του. Αυτός ο άξονας εν τέλει διακλαδίζεται και συνδέεται μέσω των συνάψεων με άλλους νευρώνες. Όσον αφορά το τεχνητό μοντέλο, τα σήματα ταξιδεύουν κατά μήκος των αξόνων (x_0, x_1,

...) και έρχονται σε αλληλεπίδραση με τους υπόλοιπους νευρώνες ($w_0 * x_0$, ...) μέσω του βάρους (w_0, ...) που χαρακτηρίζει την εκάστοτε σύναψη. Οι τιμές των βαρών χαρακτηρίζουν την ισχύ της επιρροής που έχει κάθε είσοδος στο νευρώνα. Στη συνέχεια, όλες οι σταθμισμένες πλέον είσοδοι αθροίζονται και αν το άθροισμα τους ξεπεράσει ένα συγκεκριμένο κατώφλι ο νευρώνας πυροδοτείται. Στο τεχνητό μοντέλο, η πυροδότηση μοντελοποιείται μέσω μίας συνάρτησης ενεργοποίησης, η οποία δίνει την τιμή του σήματος εξόδου του νευρώνα βάση του αθροίσματος των σταθμισμένων εισόδων και του κατωφλιού. Η παραπάνω αναλογία παρουσιάζεται στην παρακάτω εικόνα:



Σχήμα 1.1: Σύγκριση βιολογικού και τεχνητού νευρώνα [4]

Συνοψίζοντας, ένας τεχνητός νευρώνας αποτελείται από τα τρία εξής στοιχεία:

1. Ένα σετ βαρών, όπου κάθε βάρος αντιστοιχεί σε μία είσοδο και υποδηλώνει την ισχύ της επιρροής της εκάστοτε εισόδου.

2. Έναν αθροιστή, ο οποίος αθροίζει τις σταθμισμένες πλέον από τα βάρη εισόδους.

3. Μία συνάρτηση ενεργοποίησης που καθορίζει την τιμή της εξόδου του νευρώνα συγκρίνοντας το αποτέλεσμα του αθροιστή με ένα κατώφλι b.

Βάση των παραπάνω, για έναν νευρώνα k, m εισόδων, η έξοδος y_k δίνεται από τη σχέση:

$$y_k = f\left(\sum_{i=1}^{m} w_{ki} x_i + b_k\right)$$

Όπου x_i η εκάστοτε είσοδος, w_{ki} το βάρος της εκάστοτε εισόδου, b_k το κατώφλι και f η συνάρτηση ενεργοποίησης.

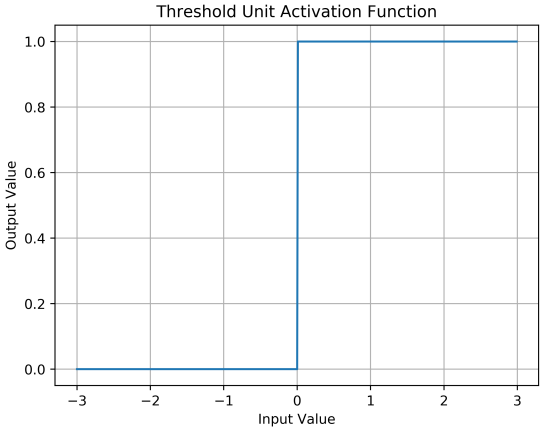
Μερικές συχνά χρησιμοποιούμενες συναρτήσεις ενεργοποίησης νευρώνων είναι οι παρακάτω:

- Threshold Unit

  Η συνάρτηση αυτή αποτελεί την απλούστερη μορφή συνάρτησης ενεργοποίησης και πρακτικά μοντελοποιεί το αν ο νευρώνας θα ενεργοποιηθεί ή θα μείνει απενεργοποιημένος δεδομένης μίας εισόδου. Η έξοδος της y για μία δεδομένη είσοδο x δίνεται από τη σχέση: [3]

  $$y = \begin{cases} 1 & if \quad x \geq 0 \\ 0 & if \quad x < 0 \end{cases}$$

  Το γράφημα εισόδου-εξόδου της Threshold Unit παρουσιάζεται στην εικόνα 1.2.



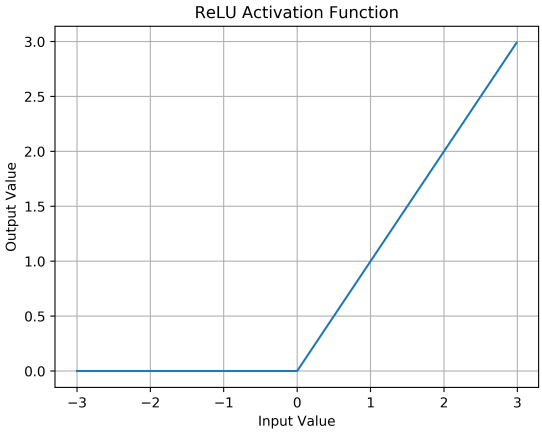
Σχήμα 1.2: Threshold Unit Activation Function

- Rectifier Linear Unit (ReLU)

  Η συνάρτηση αυτή αποτελεί συχνή επιλογή λόγω της απλότητας της και της καλής απόδοσης που παρουσιάζει σε πλήθος εφαρμογών. Για δεδομένη τιμή εισόδου x, η έξοδος της ReLU δίνεται από τη σχέση: [5]

$$ReLU(x) = max(x, 0)$$

  Το γράφημα εισόδου-εξόδου της ReLU παρουσιάζεται στην εικόνα 1.3.



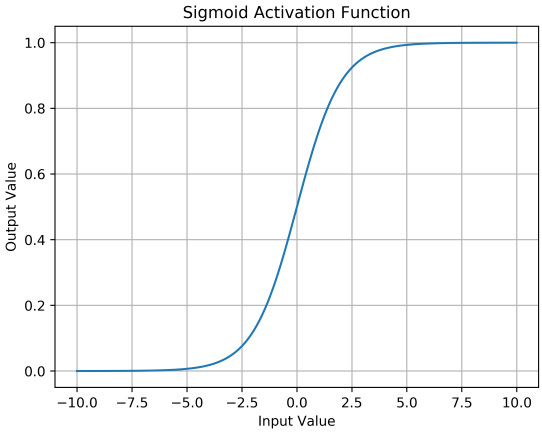
Σχήμα 1.3: ReLU Activation Function

- Simgoid Function

  Η σιγμοειδής συνάρτηση χρησιμοποιήθηκε ως συνάρτηση ενεργοποίησης εξαιτίας της λείας φύσης της, που την καθιστά παραγωγίσιμη προσέγγιση της συνάρτησης threshold unit. Η σιγμοειδής συνάρτηση βρίσκει ευρεία χρήση σε περιπτώσεις όπου είναι επιθυμητή η ερμηνεία της εξόδου ενός νευρώνα ως πιθανότητα σε προβλήματα ταξινόμησης. Για δεδομένη τιμή εισόδου x, η έξοδος της σιγμοειδούς συνάρτησης δίνεται από τη σχέση: [5]

$$Sigmoid\,(x) = \frac{1}{1 + e^{-x}}$$

Το γράφημα εισόδου-εξόδου της σιγμοειδούς συνάρτησης παρουσιάζεται στην εικόνα 1.4.



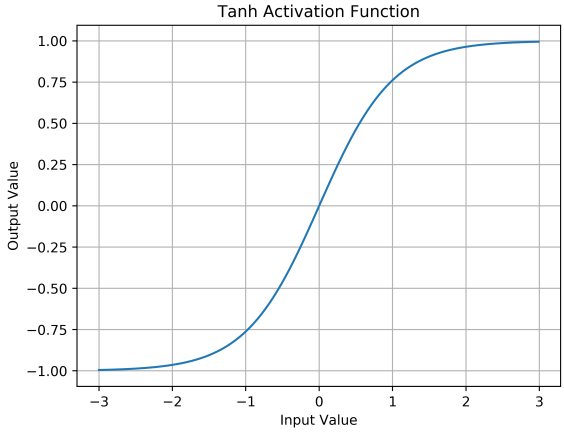
Σχήμα 1.4: Sigmoid Activation Function

- Tanh Function

  Όπως η σιγμοειδής συνάρτηση έτσι και η υπερβολική εφαπτομένη επιτυγχάνει να «συνθλίβει» την είσοδο, αυτή τη φορά μεταξύ των τιμών -1 και 1. Για δεδομένη τιμή εισόδου x, η έξοδος δίνεται από τη σχέση: [5]

$$Tanh\,(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

  Το γράφημα εισόδου-εξόδου της υπερβολικής εφαπτομένης παρουσιάζεται στην εικόνα 1.5.



Σχήμα 1.5: Tanh Activation Function

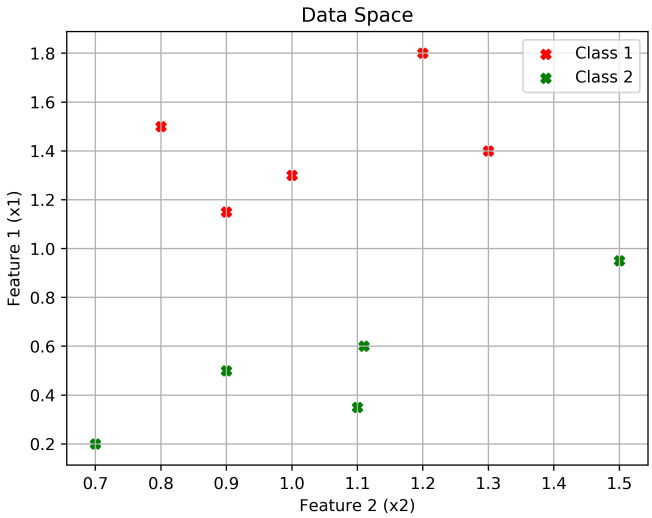
1.2.3 Γεωμετρική Ερμηνεία Λειτουργίας Νευρώνα

Για απλούστευση της οπτικοποίησης των παρακάτω, γίνεται υπόθεση ενός νευρώνα δύο εισόδων. Κάθε μία από τις δύο εισόδους αναφέρεται σε ένα χαρακτηριστικό, ή διαφορετικά σε μία συνιστώσα, ενός διανύσματος εισόδου $\mathbf{X} = [x1, x2]$. Όπως αναφέρθηκε και παραπάνω, σε κάθε μία από τις εισόδους αναθέτεται και ένα βάρος w_i.

Επομένως, στην προκειμένη περίπτωση έχουμε ένα σετ βαρών $[w1, w2]$. Ο αθροιστής του νευρώνα παράγει το αποτέλεσμα :

$$u_k = x_1 * \ w_{k1} + x_2 * \ w_{k2} + b_k$$

Στην περίπτωση αυτή, τα δεδομένα εισόδου βρίσκονται σε έναν χώρο δύο διαστάσεων (μία διάσταση για κάθε χαρακτηριστικό της συνολικής εισόδου) και επομένως μπορούν να τοποθετηθούν σε ένα επίπεδο. Υποθέτοντας πως τα δεδομένα χωρίζονται σε δύο διαφορετικές κλάσεις, ένα παράδειγμα ενός τέτοιου χώρου παρουσιάζεται στην εικόνα 1.6.



Σχήμα 1.6: Χώρος δεδομένων δύο διαστάσεων αποτελούμενος από δύο γραμμικά διαχωρίσιμες κλάσεις

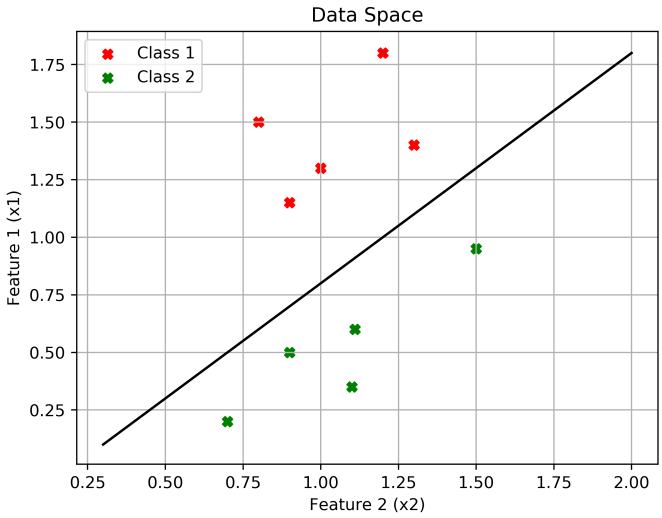
Παρατηρώντας την έξοδο του αθροιστή του νευρώνα (όπου έχει τη μορφή εξίσωσης ευθείας), γίνεται αντιληπτό ότι με κατάλληλη επιλογή των βαρών $w1$ και $w2$ και του κατωφλιού b_k, είναι δυνατή η χάραξη μίας ευθείας, η οποία θα διαχωρίζει το επίπεδο σε δύο ημι-επίπεδα, κάθε ένα εκ των οποίων θα περιέχει δεδομένα μόνο μίας κλάσης. Μία τέτοια ευθεία παρουσιάζεται στην εικόνα 1.7.

Πλέον είναι αντιληπτό ότι στις περιπτώσεις όπου το u_k ξεπερνάει την τιμή 0, δηλαδή όταν κάποιο δεδομένο εισόδου βρίσκεται πάνω από την χαραγμένη ευθεία, ο νευρώνας θα οδηγηθεί σε ενεργοποίηση ενώ θα μείνει απενεργοποιημένος (δηλαδή η έξοδος του θα λάβει μηδενική η αρνητική τιμή), για αρνητικές τιμές του u_k. Είναι αντιληπτό, ότι ο νευρώνας έχει την ικανότητα της κατηγοριοποίησης γραμμικά διαχωρίσιμων δεδομένων εισόδου και προφανώς τα παραπάνω μπορούν να γενικευτούν σε περισσότερες από δύο διαστάσεις εισόδου, ωστόσο η οπτικοποίηση θα ήταν δυσκολότερη. Ωστόσο, η σωστή κατηγοριοποίηση θα ήταν αδύνατη στην περίπτωση μη γραμμικά διαχωρίσιμων δεδομένων καθώς δεν θα υπήρχαν τιμές για τα w_1, w_2 και b_k όπου θα έδιναν την επιθυμητή ευθεία. Μία τέτοια περίπτωση δεδομένων παρουσιάζεται στην εικόνα 1.8.

Είναι διαισθητικά αντιληπτό ότι η εύρεση ευθείας, η οποία να δύναται να διαχωρίσει πλήρως τα δεδομένων των δύο διαφορετικών κλάσεων είναι αδύνατη.

1.2.4 Εκμάθηση Νευρώνα

Στην παραπάνω παράγραφο έγινε αναφορά στη γεωμετρική ερμηνεία της λειτουργίας του νευρώνα και θεωρήθηκε δεδομένη η επιλογή των παραμέτρων w_1, w_2 και b_k, οι οποίες μας έδιναν την επιθυμητή ευθεία που επιτύγχανε τον διαχωρισμό των κλάσεων. Σε αυτή την παράγραφο θα αναλυθεί ο αλγόριθμος εύρεσης των παραμέτρων αυτών για την περίπτωση της επιβλεπόμενης μάθησης.



Σχήμα 1.7: Η ευθεία διαχωρισμού του νευρώνα

Ο αλγόριθμος Perceptron Convergence [3]

Έστω δεδομένα εισόδου τα οποία προέρχονται από δύο γραμμικά διαχωρίσιμες κλάσεις C1 και C2 και ότι τα δεδομένα βρίσκονται στη μορφή διανύσματος m συνιστωσών, όπου m το σύνολο των διαφορετικών χαρακτηριστικών τους. Επιπλέον, θεωρείται ότι ο νευρώνας χαρακτηρίζεται από m διαφορετικά βάρη και το σταθερό όρο b_k.
Ορίζουμε το διάνυσμα βαρών $W(n)$ ως $W(n) = [b, w1(n), w2(n), \ldots, w_m(n)]$ και την είσοδο ως $X(n) = [1, x_1(n), x_2(n), \ldots, x_m(n)]^T$. Για λόγους σύμβασης, έχει συμπεριληφθεί στο διάνυσμα και ο σταθερός όρος, προσθέτοντας απλά στο διάνυσμα εισόδου ένα σταθερό χαρακτηριστικό που λαμβάνει την τιμή 1. Τότε η έξοδος του αθροιστή του νευρώνα για τη n-οστή είσοδο κατά τη n-οστή επανάληψη του αλγορίθμου δίνεται απλά από τη σχέση $W(n) * X(n)$. Η τελική έξοδος του νευρώνα y υποθέτοντας συνάρτηση ενεργοποίησης sgn όπου :

$$sgn(i) = \begin{cases} 1 & if \quad i > 0 \\ -1 & if \quad i \leq 0 \end{cases}$$

Θα δίνεται από τη σχέση:

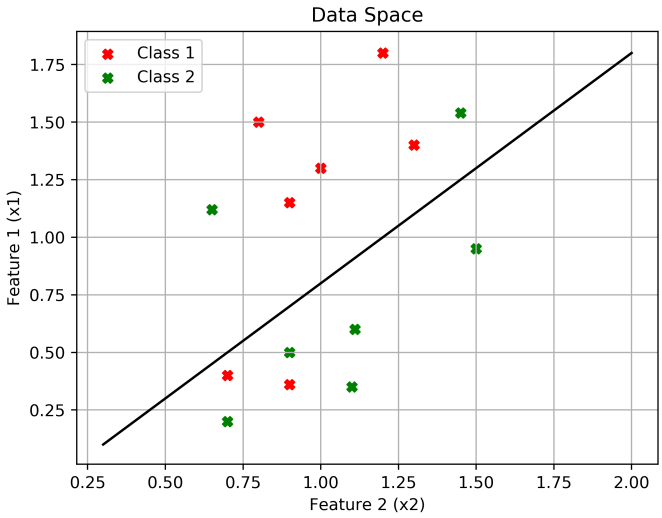
$$y\,(n) = sgn\,(W\,(n) * X\,(n))$$

ενώ ορίζεται και η επιθυμητή αντίδραση του νευρώνα για τη δεδομένη είσοδο ώς $d(n)$.
Εφόσον οι κλάσεις των δεδομένων είναι γραμμικά διαχωρίσιμες, υπάρχουν τιμές των παραμέτρων του διανύσματος $W(n)$ έτσι ώστε:

- $W(n) * X(n) > 0$ για κάθε διάνυσμα εισόδου που ανήκει στην κλάση $C1$.
- $W(n) * X(n) \leq s0$ για κάθε διάνυσμα εισόδου που ανήκει στην κλάση $C2$.

Για την εύρεση των τιμών αυτών, ακολουθούνται τα εξής βήματα:

1. **Αρχικοποίηση**: Θέτονται οι τιμές του διανύσματος βαρών στο $W(0) = 0$.
   Για τα βήματα n = 1,2,...
2. **Ενεργοποίηση**: Κατά τη n-οστή επανάληψη, ο νευρώνας ενεργοποιείται με την είσοδο $X(n)$ και αναμένεται να δώσει την επιθυμητή έξοδο $d(n)$.
3. **Υπολογισμός απόκρισης νευρώνα**: Γίνεται ο υπολογισμός της εξόδου ως $y(n) = sgn(W(n) * X(n))$.



Σχήμα 1.8: Μη γραμμικά διαχωρίσιμα δεδομένα

4. **Προσαρμογή του διανύσματος βαρών**: Το διάνυσμα βαρών ενημερώνεται ως εξής:

$$W(n+1) = W(n) + \eta * [d(n) - y(n)]\, x(n)$$

Όπου:

$$d(n) = \begin{cases} 1 & if \quad x(n) \in C1 \\ -1 & if \quad x(n) \in C2 \end{cases}$$

5. **Επανάληψη**: Συνέχιση στο βήμα $n+1$

Αποδεικνύεται ότι ο παραπάνω αλγόριθμος, οποίος έχει το όνομα «αλγόριθμος σύγκλισης νευρώνα» («perceptron convergence algorithm»), οδηγεί στην εύρεση των παραμέτρων που προσδιορίζουν το υπερεπίπεδο (ή την ευθεία στην περίπτωση που έχουμε δισδιάστατα δεδομένα), το οποίο επιτυγχάνει τον διαχωρισμό των δύο κλάσεων. [8]

The Batch Perceptron Algorithm [3]

Ο αλγόριθμος perceptron convergence που παρουσιάστηκε παραπάνω επικεντρώθηκε στη διόρθωση του διανύσματος βαρών χρησιμοποιώντας για κάθε βήμα μόνο ένα δείγμα από το σύνολο των δεδομένων εισόδου. Ο παρών αλγόριθμος, χρησιμοποιώντας μία συνάρτηση κόστους (cost function), επιτυγχάνει τη διόρθωση του διανύσματος βαρών, χρησιμοποιώντας ένα σύνολο δειγμάτων (batch) από τα δεδομένα εισόδου. Υπό μία έννοια, ο αλγόριθμος perceptron convergence αποτελεί ειδική περίπτωση του αλγορίθμου batch perceptron, όπου το μέγεθος του batch είναι ίσο με ένα. [8]
Η συνάρτηση κόστους, αποτελεί ένα δείκτη για το πόσο διαφέρουν οι πραγματικές τιμές από τις προβλεπόμενες που δίνει στην έξοδό του ο νευρώνας για ένα διάνυσμα εισόδου $\mathbf{X}$. Λαμβάνει τιμές συνήθως μη αρνητικές και στόχος είναι η ελαχιστοποίηση της. Επομένως, μικρότερες τιμές της συνάρτησης κόστους μεταφράζονται ως καλύτερες ενώ τέλειες προβλέψεις του νευρώνα δίνουν μηδενική τιμή στη συνάρτηση κόστους. [3]
Υποθέτουμε ένα υποσύνολο του training set αποτελούμενο από k δείγματα, y την έξοδο του νευρώνα και $\hat{y}$ την επιθυμητή έξοδο. Δύο συχνά χρησιμοποιημένες συναρτήσεις κόστους είναι οι παρακάτω:

- Mean Square Error:

$$L(W) = \frac{\sum_{i=1}^{k} (y_i(W) - \widehat{y_i})^2}{n}$$

- Binary Cross Entropy:

$$L(W) = -\sum_{i=1}^{k} y_i(W)\log(\widehat{y_i}) + (1 - y_i(W))\log(1 - \widehat{y_i})$$

Η binary cross entropy συνάρτηση κόστους χρησιμοποιείται για περιπτώσεις όπου υπάρχουν δεδομένα μόνο δύο κλάσεων προς ταξινόμηση. Για ταξινόμηση δεδομένων σε περισσότερες από δύο κλάσεις, χρησιμοποιείται μια πιο γενικευμένη μορφή της (Cross Entropy Loss Function).

Για την διόρθωση του διανύσματος βαρών $W = [b, w1, \ldots, w_m]$ είναι απαραίτητος ο υπολογισμός του gradient της συνάρτησης κόστους ως προς τις παραμέτρους προς διόρθωση, δηλαδή του $\nabla L(W)$ όπου $\nabla = \left[\frac{\partial}{\partial b}, \frac{\partial}{\partial w_1}, \ldots, \frac{\partial}{\partial w_m}\right]$. Επιθυμούμε να τροποποιήσουμε το διάνυσμα βαρών με τρόπο τέτοιο ώστε να ελαχιστοποιήσουμε τη συνάρτηση κόστους, επομένως ακολουθούμε το μονοπάτι της πιο απότομης κατάβασης(steepest descent), μεταβάλλοντας το διάνυσμα βαρών ως εξής: [3],[8]

$$W(n+1) = W(n) - \eta(n)\nabla L(W(n))$$

Με αυτόν τον τρόπο ο αλγόριθμος perceptron convergence έχει γενικευτεί προσφέροντας κατάλληλη διόρθωση του διανύσματος βαρών βάση ενός συνόλου δεδομένων και όχι απλά για κάθε ένα από αυτά ξεχωριστά.

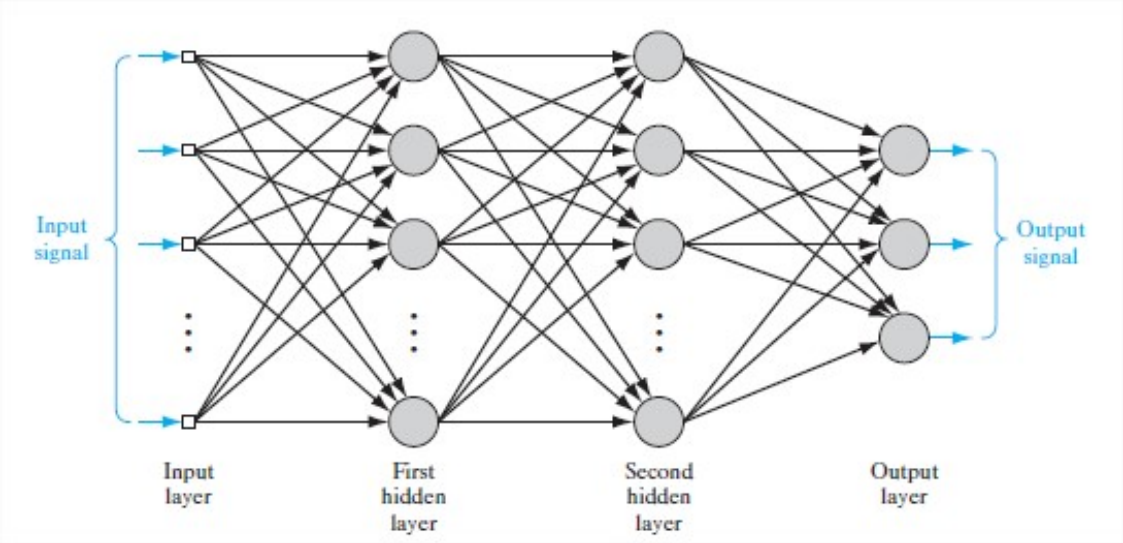
1.2.5 Πολυεπίπεδα Νευρωνικά Δίκτυα

Στην προηγούμενη παράγραφο αναλύθηκε ο τρόπος λειτουργίας ενός νευρώνα καθώς και η διαδικασία επιλογής των παραμέτρων του για τον ορθό διαχωρισμό δεδομένων, τα οποία ανήκουν σε δύο διαφορετικές γραμμικά διαχωρίσιμες κλάσεις. Σε αυτήν την παράγραφο θα αναλυθούν δίκτυα αποτελούμενα από τέτοιους νευρώνες, τα οποία διαθέτουν περισσότερα από ένα επίπεδο.
Ο νευρώνας που μελετήθηκε παραπάνω μπορεί να θεωρηθεί ως νευρωνικό δίκτυο ενός επιπέδου και είναι ικανός για την κατηγοριοποίηση γραμμικά διαχωρίσιμων δεδομένων. Για να ξεπεράσουμε αυτόν τον περιορισμό χρησιμοποιούμε μία αρχιτεκτονική διασυνδεδεμένων νευρώνων γνωστή και ως «πολυεπίπεδο νευρωνικό δίκτυο» (multilayer perceptron). Συχνά, τα πολυεπίπεδα νευρωνικά δίκτυα χαρακτηρίζονται και ως "vanilla neural networks", ειδικά όταν διαθέτουν μόνο ένα κρυμμένο επίπεδο νευρώνων. Η αρχιτεκτονική αυτή διαθέτει τα εξής χαρακτηριστικά: [3]

- Κάθε νευρώνας διαθέτει μία μη γραμμική συνάρτηση ενεργοποίησης.
- Το δίκτυο διαθέτει ένα ή περισσότερα επίπεδα νευρώνων, τα οποία είναι κρυμμένα και από τους κόμβους εισόδου αλλά και από τους κόμβους εξόδου.
- Το δίκτυο διαθέτει μεγάλο βαθμό συνδεσιμότητας μεταξύ των νευρώνων.

Κάθε επίπεδο του δικτύου αποτελείται από έναν ή περισσότερους νευρώνες. Οι νευρώνες εξόδου αποτελούν το επίπεδο εξόδου του νευρωνικού δικτύου ενώ οι υπόλοιποι νευρώνες συνθέτουν τα επιμέρους κρυμμένα επίπεδα. Η έξοδος κάθε νευρώνα ενός επιπέδου αποτελεί είσοδο για όλους τους νευρώνες του αμέσως επόμενου επιπέδου.

Παράδειγμα μίας τέτοιας αρχιτεκτονικής νευρωνικού δικτύου παρουσιάζεται στην εικόνα 1.9.:



Σχήμα 1.9: Παράδειγμα πολυεπίπεδου νευρωνικού δικτύου [3]

Ιδιαίτερη είναι η σημασία των κρυμμένων νευρώνων, καθώς εξαιτίας τους γίνεται επιτεύξιμη η κατηγοριοποίηση μη γραμμικά διαχωρίσιμων δεδομένων. Οι κρυμμένοι νευρώνες στην πραγματικότητα λειτουργούν ως ανιχνευτές χαρακτηριστικών (feature detectors). Κατά τη διαδικασία της εκμάθησης μαθαίνουν να αναγνωρίζουν κρυμμένα χαρακτηριστικά των δεδομένων. Κάθε επίπεδο εφαρμόζει στην πραγματικότητα ένα μη γραμμικό μετασχηματισμό, με τον οποίο περνάμε από το χώρο των δεδομένων στο χώρο των χαρακτηριστικών (feature space) . Σε αυτό το χώρο τα δεδομένα μπορούν να κατηγοριοποιηθούν ευκολότερα σε σύγκριση με τον αρχικό χώρο δεδομένων.

1.2.6 Εκμάθηση Πολυεπίπεδων Νευρωνικών Δικτύων

Κατά την εκμάθηση, όπως και στην περίπτωση τους ενός νευρώνα, γίνεται σύγκριση της εξόδου του δικτύου με την επιθυμητή έξοδο. Το αποτέλεσμα της σύγκρισης οδηγεί σε τροποποίηση των βαρών και του κατωφλιού κάθε νευρώνα ενώ η διαδικασία τροποποίησης διαδίδεται από το επίπεδο εξόδου προς το πρώτο κρυμμένο επίπεδο σταδιακά. Ο αλγόριθμος εκμάθησης για τα πολυεπίπεδα νευρωνικά δίκτυα, για την περίπτωση της επιβλεπόμενης μάθησης, ονομάζεται back-propagation και ο τρόπος με τον οποίο γίνεται η τροποποίηση των παραμέτρων των νευρώνων κάθε επιπέδου του δικτύου παρουσιάζεται παρακάτω.
Θεωρείται πολυεπίπεδο νευρωνικό δίκτυο, το οποίο αποτελείται από τουλάχιστον ένα κρυφό επίπεδο νευρώνων και προφανώς, το επίπεδο εξόδου. Κάθε ένας από τους νευρώνες του επιπέδου εξόδου παράγει μία έξοδο $y_j(n)$ ενώ υπάρχει η επιθυμητή έξοδος του εκάστοτε νευρώνα, $d_j(n)$. Το σφάλμα της εξόδου του j-οστού νευρώνα του επιπέδου εξόδου δίνεται από τη σχέση $e_j(n) = d_j(n) - y_j(n)$. Ορίζουμε ως στιγμιαία ενέργεια σφάλματος του νευρώνα j την ποσότητα $E_j(n) = \frac{1}{2}e_j^2(n)$. Η συνολική στιγμιαία ενέργεια σφάλματος του επιπέδου εξόδου του δικτύου είναι:

$$E\left(n\right) = \sum_{j \in C} E_j\left(n\right) = \frac{1}{2} \sum_{j \in C} e_j^2\left(n\right)$$

Για την περίπτωση της εκπαίδευσης με τη χρήση N δειγμάτων από τα δεδομένα, γίνεται χρήση της μέσης στιγμιαίας ενέργειας σφάλματος, που δεν αποτελεί τίποτα άλλο από το μέσο όρο των ενεργειών σφάλματος των επιμέρους δειγμάτων.

$$E_{av}\left(N\right) = \frac{1}{N} \sum_{n=1}^{N} E\left(n\right)$$

Όπως και στην περίπτωση του ενός νευρώνα, η στιγμιαία ενέργεια σφάλματος χρησιμοποιείται για την τροποποίηση των παραμέτρων των νευρώνων μετά τον υπολογισμό της εξόδου για ένα δείγμα δεδομένων (online

learning), ενώ, η μέση ενέργεια σφάλματος για την τροποποίηση ύστερα από τον υπολογισμό της εξόδου για ένα ολόκληρο σετ δεδομένων (batch learning). Σε κάθε περίπτωση, στόχος είναι η τροποποίηση των παραμέτρων των νευρώνων με τέτοιο τρόπο ώστε να επιτυγχάνεται η ελαχιστοποίηση της ενέργειας σφάλματος, δηλαδή της συνάρτησης κόστους.
Όπως έχει ήδη αναφερθεί, η έξοδος του αθροιστή ενός νευρώνα m εισόδων δίνεται από τη σχέση:

$$v_j = \sum_{i=1}^{m} w_{ji}(n) * x_i(n)$$

Ενώ η τελική έξοδος του νευρώνα είναι:

$$y_j(n) = \varphi_j(v_j(n))$$

Όπου ϕ_j η συνάρτηση ενεργοποίησης.
Εφόσον η έξοδος των νευρώνων είναι συνάρτηση των παραμέτρων του νευρώνα, μπορεί να υπολογιστεί η διόρθωση $\Delta w_{ji}(n)$ που πρέπει να γίνει στις παραμέτρους για την ελαχιστοποίηση της ενέργειας σφάλματος, η οποία διόρθωση είναι ανάλογη της μερικής παραγώγου $\frac{\partial E(n)}{\partial w_{ji}(n)}$. Σύμφωνα με τον κανόνα της αλυσίδας έχουμε:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Οι τελικοί υπολογισμοί των επιμέρους μερικών παραγώγων παρουσιάζονται παρακάτω:

- $\frac{\partial E(n)}{\partial e_j(n)} = e_j(n)$
- $\frac{\partial e_j(n)}{\partial y_j(n)} = -1$
- $\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$
- $\frac{\partial v_j(n)}{\partial w_{ji}(n)} = x_i(n)$

Αντικαθιστώντας λαμβάνεται η εξής μαθηματική παράσταση για την παράγωγο $\frac{\partial E(n)}{\partial w_{ji}(n)}$:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))x_i(n)$$

Η τελική διόρθωση που πρέπει να γίνει στην παράμετρο w_{ji} για την ελαχιστοποίηση της ενέργειας σφάλματος είναι:

$$\Delta w_{ji}(n) = -n\frac{\partial E(n)}{\partial w_{ji}(n)}$$

Όπου n παράμετρος του ρυθμού μάθησης, ενώ το αρνητικό πρόσημο υπάρχει προκειμένου η τροποποίηση που θα γίνει στα βάρη να γίνει προς τη φορά που ελαχιστοποιείται η συνάρτηση κόστους (gradient descent). Με τελική αντικατάσταση, η τροποποίηση των βαρών του j-οστού νευρώνα του επιπέδου εξόδου είναι:

$$\Delta w_{ji}(n) = n\delta_j(n)x_i(n)$$

Όπου $\delta_j(n)$ ορίζεται ως η τοπική κλίση και ισούται με:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = e_j(n)\varphi_j'(v_j(n))$$

Τα παραπάνω ισχύουν όπως αναφέρθηκε για νευρώνες του επιπέδου εξόδου καθώς ήταν δυνατός ο προσδιορισμός του σφάλματος $e_j(n)$ της εξόδου του εκάστοτε νευρώνα. Ωστόσο, το ίδιο δεν ισχύει για τους νευρώνες των κρυφών επιπέδων καθώς για αυτούς δεν υπάρχει κάποια επιθυμητή έξοδος, γεγονός που περιπλέκει τον αλγόριθμο back-propagation.
Για τους νευρώνες των κρυφών επιπέδων, επαναπροσδιορίζεται η τοπική κλίση δ ως εξής:

Για τον νευρώνα j ενός κρυμμένου επιπέδου πριν το επίπεδο εξόδου με έξοδο y_j και για νευρώνες του επιπέδου εξόδου που χαρακτηρίζονται από το δείκτη k:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} \varphi_j'(v_j(n))$$

Για τον όρο $\frac{\partial E(n)}{\partial y_j(n)}$ έχουμε:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

Επειδή $e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n)) \Rightarrow \frac{\partial e_k(n)}{\partial v_k(n)} = \varphi_k'(v_k(n))$

Επιπλέον $v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \Rightarrow \frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$

Άρα τελικά ισχύει ότι:

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_k e_k(n) \varphi_k'(v_k(n)) w_{kj}(n) = -\sum_k \delta_k(n) w_{kj}(n)$$

Επομένως, η τοπική κλίση για έναν νευρώνα του κρυμμένου επιπέδου πριν το επίπεδο εξόδου δίνεται από τη σχέση:

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$

Συνοψίζοντας τις παραπάνω σχέσεις, για τη διόρθωση των παραμέτρων ενός νευρώνα j ισχύει:

$$\begin{pmatrix} \text{Διόρθωση Παραμέτρων} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \text{Ρυθμός Μάθησης} \\ \eta \end{pmatrix} * \begin{pmatrix} \text{Τοπική Κλίση} \\ \delta_j(n) \end{pmatrix} * \begin{pmatrix} \text{Είσοδος} \\ x_i(n) \end{pmatrix}$$

Όπου για την τοπική κλήση διακρίνονται οι εξής δύο περιπτώσεις:

1. Αν ο νευρώνας j ανήκει στο επίπεδο εξόδου τότε:

$$\delta_j(n) = e_j(n) \varphi_j'(v_j(n))$$

2. Αν ο νευρώνας j ανήκει σε κάποιο κρυφό επίπεδο τότε:

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$

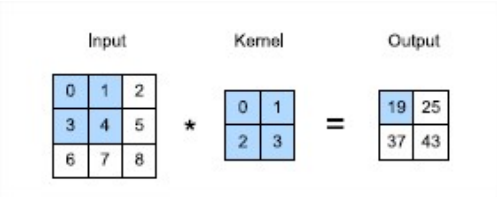
Όπου δ_k και w_{kj} οι τοπικές κλίσεις και οι παράμετροι των νευρώνων των αμέσως επόμενων επιπέδων που είναι συνδεδεμένοι στο νευρώνα j.

1.2.7 Είδη Επιπέδων

Τα απλά πολυεπίπεδα νευρωνικά δίκτυα, των οποίων η διεξοδική παρουσίαση έγινε παραπάνω, αποτελούνται από πολλαπλά επίπεδα νευρώνων με κάθε νευρώνα ενός επιπέδου πλήρως διασυνδεδεμένο με όλους τους νευρώνες του αμέσως επόμενου και του αμέσως προηγούμενου επιπέδου. Ωστόσο, υπάρχουν μοντέλα νευρωνικών δικτύων τα οποία δεν περιλαμβάνουν μόνο απλά επίπεδα αποτελούμενα από νευρώνες. Στην παρούσα παράγραφο γίνεται μία συνοπτική παρουσίαση των διάφορων ειδών επιπέδων που χρησιμοποιούνται συχνότερα.

- Convolutional Layers

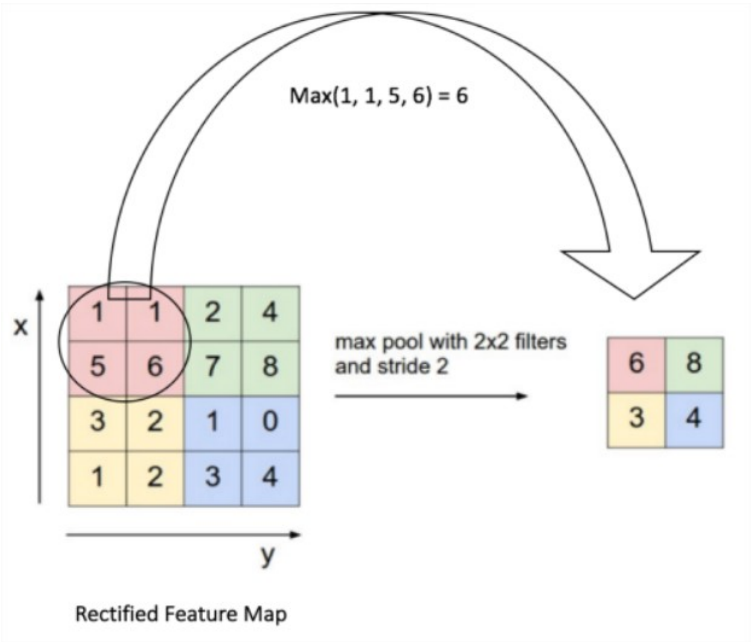
  Επί της ουσίας αυτό το είδος επιπέδου αποτελεί ένα φίλτρο, το οποίο εφαρμόζεται στα δεδομένα εισόδου με σκοπό την ανίχνευση κάποιου χαρακτηριστικού σε αυτά. Η χρήση του συναντάται συχνά σε εφαρμογές όπου τα δεδομένα εισόδου αποτελούν κάποια εικόνα. Το φίλτρο εφαρμόζεται σε όλη την επιφάνεια της εικόνας και έχει ως έξοδο ένα χάρτη χαρακτηριστικών (feature map), ο οποίος επεξεργάζεται από τα αμέσως επόμενα επίπεδα. Ένα παράδειγμα για την εφαρμογή ενός τέτοιου φίλτρου διαστάσεων $2x2$ σε μία είσοδο διαστάσεων $3x3$ παρουσιάζεται στην εικόνα 1.10.



Σχήμα 1.10: Παράδειγμα συνέλιξης [5]

- Pooling Layers

  Τα pooling layers βρίσκονται συχνά μετά από τα συνελικτικά επίπεδα. Όπως και στην περίπτωση των συνελικτικών επιπέδων, αποτελούν φίλτρα, τα οποία εφαρμόζονται στα δεδομένα εισόδου, ωστόσο η λειτουργία τους σχετίζεται όχι με την ανίχνευση χαρακτηριστικών αλλά με την υποδειγματοληψία του feature map. Η υποδειγματοληψία γίνεται με την εφαρμογή κάποιου κανόνα στο τμήμα των δεδομένων όπου γίνεται η επεξεργασία. Για παράδειγμα, για ένα φίλτρο $2x2$ που εφαρμόζει Max Pooling, και εφαρμόζεται σε δεδομένα εισόδου διαστάσεων $4x4$ κινούμενο κάθε φορά με βήμα 2, έχουμε:



Σχήμα 1.11: Παράδειγμα Max Pooling

Διαφορετικός κανόνας από αυτόν του Max Pooling αποτελεί το Average Pooling όπου στην έξοδο κάθε φορά δίνεται ο μέσος όρος των δεδομένων εισόδου που υπόκεινται στην επεξεργασία του φίλτρου.

- Recurrent Layers

  Τα recurrent layers αποτελούνται από νευρώνες στους οποίους η έξοδος ανατροφοδοτείται στην είσοδο. Με αυτόν τον τρόπο η είσοδος του εκάστοτε νευρώνα αποτελείται από τα νέα δεδομένα που

υπεισέρχονται για επεξεργασία αλλά και από προηγούμενους υπολογισμούς δημιουργώντας έτσι ένα είδος μνήμης. Τέτοιου είδους επίπεδα βρίσκουν χρήση σε εφαρμογές επεξεργασίας φυσικής γλώσσας.

- Dropout Layers

  Τα dropout layers χρησιμοποιούνται για την αποφυγή του overfitting μεγάλων μοντέλων όταν αυτά εκπαιδεύονται σε μικρά datasets. Τα layers αυτά λειτουργούν θέτοντας τυχαία εισόδους στο 0 με συγκεκριμένη συχνότητα R κατά τη διάρκεια της εκπαίδευσης. Οι είσοδοι που δεν τέθηκαν στο 0 πολλαπλασιάζονται με τον όρο $1/(1-R)$ προκειμένου το συνολικό άθροισμα των εισόδων να μένει αμετάβλητο.

1.3 Generative Adversarial Networks

Στην παραπάνω ανάλυση της λειτουργίας του νευρώνα καθώς και της λειτουργίας των πολυεπίπεδων νευρωνικών δικτύων η λειτουργία των μοντέλων αφορούσε διαδικασίες κατηγοριοποίησης των δεδομένων. Ωστόσο, οι τεχνικές μηχανικής μάθησης βρίσκουν εφαρμογή όχι μόνο σε διαδικασίες κατηγοριοποίησης αλλά και σε διαδικασίες δημιουργίας δεδομένων (γεννητικές διαδικασίες) τα οποία φέρουν συγκεκριμένα χαρακτηριστικά. Μία τέτοια διαδικασία για παράδειγμα μπορεί να αποτελέσει η δημιουργία νέων ρεαλιστικών εικόνων προσώπων δεδομένου ενός ήδη υπάρχοντος συνόλου αντίστοιχων εικόνων.
Η ανάπτυξη των βαθιών νευρωνικών δικτύων και η μεγάλη επιτυχία τους στις διαδικασίες διάκρισης οδήγησε και στη βελτίωση των γεννητικών διαδικασιών εφόσον έδωσαν τη δυνατότητα να αντιμετωπίζονται τέτοιου είδους προβλήματα ως προβλήματα επιβλεπόμενης μάθησης. Το 2014, ένα πρωτοποριακό paper του Ian J. Goodfellow [6] εισήγαγε την έννοια των Generative Adversarial Networks (Γεννητικών Ανταγωνιστικών Δικτύων). Τα generative adversarial networks αποτελούνται από δύο δίκτυα, ένα εκ των οποίων αναλαμβάνει τις γεννητικές διαδικασίες και ένα που αναλαμβάνει τις διακριτικές διαδικασίες. Όπως πολύ κομψά παρομοιάζεται και στην ίδια την εργασία του Goodfellow,τα δύο αυτά δίκτυα παίζουν μεταξύ τους ένα παιχνίδι. Το γεννητικό μοντέλο κατέχει το ρόλο του πλαστογράφου ενώ το διακριτικό μοντέλο το ρόλο του αστυνομικού. Ο πλαστογράφος έχει στόχο την παραγωγή δεδομένων, τα οποία είναι ικανά να μπερδέψουν τον αστυνομικό σχετικά με την γνησιότητα τους. Ο αστυνομικός από την άλλη έχει ως στόχο την ορθή διάκριση πραγματικών δεδομένων από δεδομένα που έχουν παραχθεί από τον πλαστογράφο. Ο ανταγωνισμός των δύο μοντέλων οδηγεί στην από κοινού βελτίωσή τους με τελικό αποτέλεσμα τα πλαστά δεδομένα να είναι δυσδιάκριτα από τα πραγματικά.
Η ανάλυση στην εργασία του Goodfellow αφορά δύο πολυεπιπέδα νευρωνικά δίκτυα. Το γεννητικό δίκτυο G δέχεται στην είσοδο του τυχαίο θόρυβο z και παράγει στην έξοδο του δείγματα $G(z)$ τα οποία το διακριτικό δίκτυο D κρίνει αν είναι πραγματικά η παραγμένα από το γεννητικό δίκτυο. Η εκπαίδευση του δικτύου D γίνεται με τρόπο τέτοιο ώστε να μεγιστοποιείται η πιθανότητα σωστής διάκρισης των παραγόμενων από το G δειγμάτων από τα πραγματικά παραδείγματα. Συμβολίζεται με $D(x)$ η πιθανότητα το δείγμα x να προέρχεται από το σύνολο δεδομένων και να μην αποτελεί δείγμα παραγμένο από το G. Ταυτόχρονα, το δίκτυο G εκπαιδεύεται με τέτοιο τρόπο ώστε να ελαχιστοποιείται η ποσότητα $\log\left(1-D\left(G\left(z\right)\right)\right)$. Με άλλα λόγια τα δίκτυα D και G παίζουν το εξής παιχνίδι μεγίστου-ελαχίστου με συνάρτηση τιμών $V(G,D)$:

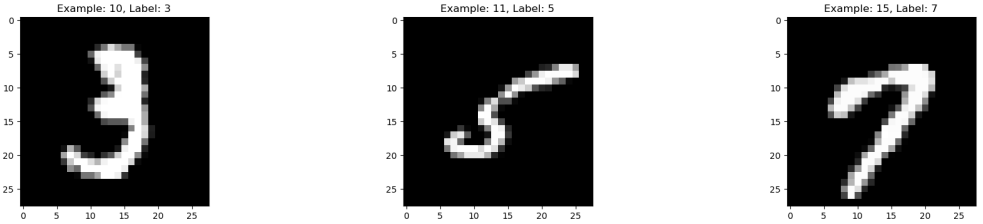
$$\underset{G}{min}\,\underset{D}{max}\,V\left(G,D\right)=E_{x\sim p_{data}(x)}[\log\left(D\left(x\right)\right]+E_{z\sim p_z(z)}\left[\log\left(1-D\left(G\left(z\right)\right)\right)\right]$$

Τα παραπάνω θα επιβεβαιωθούν μετέπειτα και πρακτικώς με τη δημιουργία ρεαλιστικών εικόνων χειρόγραφων αριθμητικών ψηφίων, με τη χρήση κώδικα Python και των βιβλιοθηκών TensorFlow & Keras για τη δημιουργία και την εκπαίδευση των νευρωνικών δικτύων, αφού πρώτα γίνει παρουσίαση του προβλήματος προς επίλυση καθώς και όλων των εργαλείων που χρησιμοποιήθηκαν.

1.4 Παρουσίαση Προβήματος

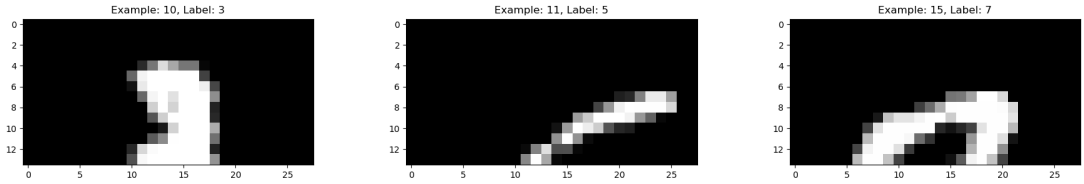
Στόχος του πρώτου μέρους της παρούσας εργασίας είναι η χρήση νευρωνικών δικτύων και συγκεκριμένα της ιδέας των Generative Adversarial Networks για την ανακατασκευή ημιτελών εικόνων. Συγκεκριμένα, οι εικόνες προς ανακατασκευή ανήκουν στο MNIST dataset.

Το MNIST dataset (Modified National Institute of Standards and Technology dataset) αποτελεί ένα σύνολο δεδομένων αποτελούμενο από εικόνες χειρόγραφων αριθμητικών ψηφίων. Η κατασκευή του αποτελεί μία αναδιοργάνωση των δειγμάτων των αρχικών dataset του NIST. Τα δεδομένα του αρχικού dataset που προορίζονταν για την εκπαίδευση μοντέλων (training set) πάρθηκαν από υπαλλήλους του αμερικανικού γραφείου απογραφής ενώ τα δεδομένα που προορίζονταν για το testing των μοντέλων (test set) πάρθηκαν από μαθητές λυκείου. Για αυτό τον λόγο το αρχικό dataset κρίθηκε ακατάλληλο για πειράματα στον κλάδο της μηχανικής μάθησης. Οι εικόνες του MNIST dataset έχουν διαστάσεις 28 επί 28 pixel ενώ η τιμή κάθε pixel λαμβάνει ακέραια τιμή από 0 έως 255, με το 0 να αντιστοιχεί στο απόλυτο μαύρο και το 255 στο απόλυτο λευκό. Οι ενδιάμεσες τιμές προφανώς αντιστοιχούν σε όλες τις ενδιάμεσες αποχρώσεις. [7]
Παρακάτω παρουσιάζονται μερικές εικόνες από το training set του MNIST dataset:



Σχήμα 1.12: Παραδείγματα εικόνων από το MNIST dataset σε απεικόνιση reverse Grayscale (255 για απόλυτο μάυρο, 0 για απόλυτο λευκό)

Οι παραπάνω εικόνες είναι πλήρεις. Όπως αναφέρθηκε, στόχος είναι η ανακατασκευή ημιτελών εικόνων και συγκεκριμένα εικόνων όπου το κάτω μισό μέρος απουσιάζει. Οι αντίστοιχες εικόνες που θα πρέπει να ανακατασκευαστούν παρουσιάζονται παρακάτω:



Σχήμα 1.13: Παραδείγματα ημιτελών εικόνων από το MNIST dataset σε απεικόνιση reverse Grayscale (255 για απόλυτο μάυρο, 0 για απόλυτο λευκό)

Είναι προφανές, ότι στις ημιτελείς εικόνες, ακόμα και μία διαδικασία διάκρισης του τι ψηφίου αναπαριστούν θα ήταν ιδιαίτερα δύσκολη ακόμα και για τον άνθρωπο. Πριν την ανάπτυξη μοντέλου ικανού για την ανακατασκευή των εικόνων του dataset, προηγείται πειραματισμός με την ανάπτυξη generative adversarial networks για τη σύνθεση νέων εικόνων σαν αυτών του dataset όπως περιγράφεται και στην εργασία του Goodfellow. Παρακάτω θα παρουσιαστούν όλα τα μοντέλα και ο τρόπος ανάπτυξης τους καθώς και τα αποτελέσματα που παράγουν, αρχικά για την περίπτωση της εκ νέου σύνθεσης και ύστερα για την περίπτωση της ανακατασκευής.

1.5 Παρουσίαση Εργαλείων

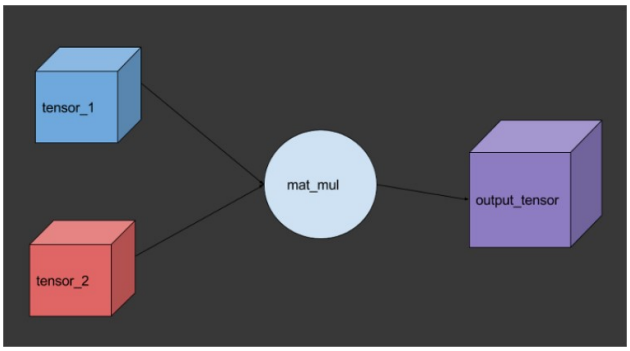
Για την δημιουργία και την εκπαίδευση των μοντέλων που επιτυγχάνουν την επίλυση του προβλήματος χρησιμοποιήθηκε η γλώσσα προγραμματισμού Python 3.7 καθώς και οι βιβλιοθήκες της, TensorFlow 2 και Keras. Επιπλέον, για την επιτάχυνση των διαδικασιών της εκπαίδευσης των μοντέλων έγινε χρήση του Google Colaboratory όπου προσφέρεται σε κάθε χρήστη η δυνατότητα απομακρυσμένης χρήσης GPUs, ενώ για τοπική εκτέλεση του κώδικα γίνεται ευρεία χρήση του Jupyter Notebook. Παρακάτω παρουσιάζονται περεταίρω λεπτομέρειες για τα εργαλεία που χρησιμοποιήθηκαν.

1.5.1 Python programming language

Η γλώσσα προγραμματισμού Python δημιουργήθηκε από το Guido van Rossum και κυκλοφόρησε για το κοινό για πρώτη φορά το 1991. Αποτελεί διερμηνευόμενη (interpreted) γλώσσα υψηλού επιπέδου και γενικού σκοπού. Υποστηρίζει πολλαπλά προγραμματιστικά υποδείγματα όπως δομημένο προγραμματισμό, προστακτικό προγραμματισμό, αντικειμενοστραφές προγραμματισμό και συναρτησιακό προγραμματισμό. Η έκδοση 2.0 της Python κυκλοφόρησε το 2000 ενώ η 3.0 έκδοση, το 2008. Βασικοί πυλώνες στους οποίους βασίστηκε ο σχεδιασμός της γλώσσας αποτελούν η εύκολη ανάγνωση του κώδικά της καθώς και η ευκολία της χρήσης της. Η Python αποτελεί βασική επιλογή για την ανάπτυξη εφαρμογών μηχανικής μάθησης καθώς και ανάλυσης δεδομένων. [8], [9]

1.5.2 TensorFlow

Το TensorFlow αποτελεί δωρεάν μαθηματική βιβλιοθήκη ανοιχτού, η οποία αναπτύχθηκε από την ομάδα Google Brain της Google. Σχεδιάστηκε για την διευκόλυνση διαδικασιών όπως ο προγραμματισμός ροής δεδομένων (dataflow programming) και ο διαφορικός προγραμματισμός (differentiable programming) και χρησιμοποιείται για την ανάπτυξη εφαρμογών μηχανικής μάθησης όπως τα νευρωνικά δίκτυα. Είναι γραμμένη σε Python, C++ αλλά και CUDA επιτρέποντας την ανάπτυξη προγραμμάτων σε GPUs (graphical processing units) και TPUs (tensor processing units). Έγινε διαθέσιμη στο κοινό για πρώτη φορά το 2015 ενώ οι εκδόσεις 1.0 και 2.0 έγιναν διαθέσιμες το 2017 και το 2019 αντίστοιχα. Συνοπτικά, η λειτουργία της στηρίζεται στην αναπαράσταση πράξεων μεταξύ τανυστών ως γράφους. Οι κόμβοι του εκάστοτε γράφου αναπαριστούν τις πράξεις ενώ οι ακμές αναπαριστούν τους τανυστές που είτε δίνονται ως είσοδος είτε προκύπτουν ως αποτέλεσμα κάποιας πράξης. [10], [11]

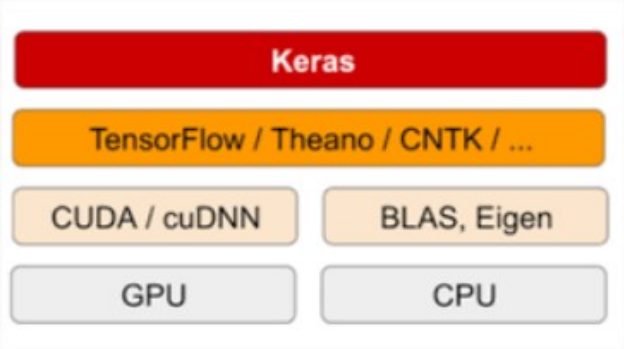


Σχήμα 1.14: Απεικόνιση λειτουργίας TensorFlow

1.5.3 Keras

Το Keras αποτελεί βιβλιοθήκη ανοιχτού κώδικα για την ανάπτυξη νευρωνικών δικτύων γραμμένη σε Python. Δεν αποτελεί αυτόνομη βιβλιοθήκη καθώς είναι σχεδιασμένη να τρέχει χρησιμοποιώντας ως backend κάποια από τις βιβλιοθήκες TensorFlow, Theano, CNTK, PlaidML, οι οποίες με τη σειρά τους είναι γραμμένες με βιβλιοθήκες γραμμικής άλγεβρας χαμηλού επιπέδου, οι οποίες υποστηρίζουν εκτέλεση είτε σε CPU είτε σε GPU.

Αναπτύχθηκε ως μέρος της ερευνητικής προσπάθειας του project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System) με κύριο δημιουργό και συντηρητή τον François Chollet, μηχανικό της Google. Η πρώτη έκδοση της βιβλιοθήκης έγινε διαθέσιμη το Μάρτιο του 2015. Στην πραγματικότητα αποτελεί μία διεπαφή υψηλού επιπέδου για την γρήγορη και φιλική προς το χρήστη δημιουργία και ανάπτυξη νευρωνικών δικτύων χωρίς την ανάμιξη του χρήστη με μαθηματικές βιβλιοθήκες. Το 2017, η ομάδα της Google υπεύθυνη για την ανάπτυξη του TensorFlow αποφάσισε να υποστηρίξει τη χρήση του Keras μέσα από τη βασική βιβλιοθήκη του TensorFlow. [12], [13]



Σχήμα 1.15: Επιμέρους αφαιρετικά επίπεδα λειτουργίας της βιβλιοθήκς Keras

1.5.4 Jupyter Notebook

Το Jupyter Notebook αποτελεί web-based, διαδραστικό υπολογιστικό περιβάλλον για τη δημιουργία εγγράφων Jupyter Notebook ανεπτυγμένο από τον μη κερδοσκοπικό οργανισμό Project Jupyter. Ένα έγγραφο Jupyter Notebook είναι στην πραγματικότητα ένα έγγραφο JSON, το οποίο περιέχει μία διατεταγμένη λίστα από κελιά εισόδου/εξόδου τα οποία περιέχουν κώδικα, κείμενο, μαθηματικές παραστάσεις, γραφήματα και ποικίλα μέσα όπως εικόνες. Το περιβάλλον Jupyter Notebook δύναται να συνδεθεί με πλήθος πυρήνων (kernels) επιτρέποντας τον προγραμματισμό σε διάφορες γλώσσες προγραμματισμού όπως η Python, η R και η Haskell. [14], [15]

1.5.5 Google Colaboratory

Το Google Colaboratory αποτελεί διαδικτυακή πλατφόρμα της Google που επιτρέπει τη συγγραφή και απομακρυσμένη εκτέλεση κώδικα μέσω του browser. Αρμόζει ειδικά για εφαρμογές μηχανικής μάθησης, ανάλυσης δεδομένων και εκπαίδευσης. Στην πραγματικότητα η πλατφόρμα φιλοξενεί μία υπηρεσία Jupyter Notebook προσφέροντας δωρεάν υπολογιστικούς πόρους στους χρήστες συμπεριλαμβανομένων και GPUs. Καθώς η εκπαίδευση μοντέλων μηχανικής μάθησης αποτελεί υπολογιστικά επίπονη διαδικασία, με τη χρήση του Google Colaboratory είναι δυνατή η αποφόρτιση του προσωπικού υπολογιστή από επίπονες διαδικασίες μεταφέροντας το υπολογιστικό φορτίο σε κάποιο απομακρυσμένο σύστημα, το οποίο χρησιμοποιώντας GPU μπορεί να ολοκληρώσει τη διαδικασία ταχύτερα. [16]

1.6 Generative Adversarial Networks για την εκ νέου σύνθεση εικόνων

Αρχικά, γίνεται μία απόπειρα εξοικείωσης με την ιδέα των Generative Adversarial Networks, και συγκεκριμένα, με οδηγό την εργασία του Goodfellow γίνεται ανάπτυξη ενός generative μοντέλου, το οποίο με είσοδο τυχαίο θόρυβο επιτυγχάνει την εκ νέου σύνθεση εικόνων χειρόγραφων αριθμητικών ψηφίων, σαν αυτών του MNIST dataset. Στην παρούσα παράγραφο παρουσιάζονται με τη σειρά η προ επεξεργασία των δεδομένων, ο κώδικας για τη δημιουργία των μοντέλων καθώς και ο κώδικας για την εκπαίδευση τους. Ο πλήρης κώδικας παραθέτεται στο παράρτημα.

1.6.1 Προεπεξεργασία δεδομένων

Όπως έχει ήδη αναφερθεί παραπάνω, το MNIST dataset περιέχει 60000 εικόνες στο training set και 10000 εικόνες στο test set κάθε μία από τις οποίες έχει διαστάσεις 28*28 pixel ενώ η τιμή κάθε pixel λαμβάνει τιμή από το 0 έως το 255. Επειδή στόχος είναι η κατασκευή μοντέλου που στην έξοδο του παράγει μία εικόνα σαν αυτές του dataset, γίνεται κανονικοποίηση των τιμών των πιξελ όλων των εικόνων στο εύρος από -1 έως 1, εφόσον η πλειοψηφία των συναρτήσεων ενεργοποίησης των νευρώνων, και συγκεκριμένα αυτών του επιπέδου εξόδου, έχουν σαν έξοδο τιμές σε αυτό το εύρος. Επιπλέον, παρατηρείται ότι μία τέτοια κανονικοποίηση των δεδομένων επιτρέπει την εκπαίδευση αποδοτικότερων μοντέλων.

Η προ επεξεργασία αυτή είναι ιδιαίτερα απλή και γίνεται με τον εξής τρόπο:

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5)/127.5
X_test = (X_test.astype(np.float32) - 127.5)/127.5
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
```

Listing 1.1: Data Preprocessing

Αφαιρώντας από κάθε pixel του dataset την τιμή 127.5 επιτυγχάνεται εύρος τιμών από -127.5 έως 127.5 για κάθε pixel κάθε εικόνας. Ύστερα, η διαίρεση με 127.5, φέρνει τα δεδομένα στην τελική επιθυμητή μορφή με εύρος τιμών από -1 έως 1. Επιπλέον, επειδή κάθε εικόνα και στο training set αλλά και στο test set είναι σε μορφή πίνακα $28x28$, γίνεται reshaping ώστε να μετατραπεί σε μορφή διανύσματος 784 στοιχείων. Το reshaping αυτό είναι απαραίτητο για την τροφοδότηση της εικόνας στα dense layers των μοντέλων τα οποία θα παρουσιαστούν παρακάτω.

1.6.2 Δημιουργία Μοντέλων

Για την επίτευξη την εκ νέου σύνθεσης εικόνων, χρησιμοποιούνται τα εξής δύο μοντέλα:

- Ως generator χρησιμοποιείται ένα multilayer perceptron αποτελούμενο από 3 hidden layers και προφανώς το output layer. Τα επιμέρους hidden layers αποτελούνται με τη σειρά από 256, 512 και 1024 νευρώνες ενώ το output layer από 794, ένας νευρώνας δηλαδή για κάθε pixel της εικόνας που θα παράξει το μοντέλο. Επιπλέον, κάθε νευρώνας όλων των hidden layer χρησιμοποιεί την ReLU συνάρτηση ενεργοποίησης.

  Οι νευρώνες του output layer χρησιμοποιούν ως συνάρτηση ενεργοποίησης την tanh η οποία έχει αναλυθεί παραπάνω. Επιλέγεται η tanh έναντι της ReLU διότι στην έξοδο είναι επιθυμητός ο περιορισμός των τιμών από -1 έως 1 καθώς, όπως έγινε και στην προ επεξεργασία δεδομένων, αναπαριστούμε τις τιμές των pixel κάθε εικόνας με αριθμούς εντός αυτού του εύρους.

- Ως discriminator χρησιμοποιείται επίσης ένα multilayer perceptron αποτελούμενο από 3 hidden layers κάθε ένα από τα οποία ακολουθείται από ένα dropout layer, και φυσικά το output layer. Τα hidden layers αποτελούνται με τη σειρά από 1024, 512 και 256 νευρώνες και κάθε νευρώνας τους χρησιμοποιεί τη LeakyReLU ως συνάρτηση ενεργοποίησης, η οποία προσδιορίζεται από την εξής σχέση:

  $$LeakyReLU(x) = \begin{cases} x & if \quad x \geq 0 \\ ax & if \quad x < 0 \end{cases}$$

  , όπου $a \geq 0$ παράμετρος της επιλογής μας.

  Το επίπεδο εξόδου αποτελείται από ένα νευρώνα ο οποίος χρησιμοποιεί ως συνάρτηση ενεργοποίησης τη sigmoid function. Η επιλογή του ενός νευρώνα δεν είναι τυχαία και γίνεται εφόσον ο ρόλος του discriminator είναι να διακρίνει αν τα δεδομένα εισόδου ανήκουν σε πραγματικές εικόνες ή είναι δημιουργήματα του generator. Για binary classification ένας νευρώνας με τιμές εξόδου στο εύρος από 0 έως 1 αρκεί. Η χρησιμότητα των dropout layers έχει αναλυθεί παραπάνω στην παράγραφο 1.2.7.

Το τμήμα του κώδικα που είναι υπεύθυνο για την κατασκευή αυτών των νευρωνικών δικτύων είναι το εξής:

```
adam = Adam(lr=0.0002, beta_1=0.5)

generator = Sequential()
generator.add(Dense(256, input_dim=randomDim, kernel_initializer=RandomNormal(stddev=0.4)))
generator.add(ReLU())
generator.add(Dense(512))
generator.add(ReLU())
generator.add(Dense(1024))
generator.add(ReLU())
generator.add(Dense(784))
generator.add(Activation(tanh))
generator.compile(loss='binary_crossentropy', optimizer=adam)

```

```
discriminator = Sequential()
discriminator.add(Dense(1024, input_dim=784, kernel_initializer=RandomNormal(stddev=0.02)))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(512))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(256))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(1))
discriminator.add(Activation(sigmoid))
discriminator.compile(loss='binary_crossentropy', optimizer=adam)

# Combined network
discriminator.trainable = False
ganInput = tf.keras.Input(shape=(randomDim,))
x = generator(ganInput)
ganOutput = discriminator(x)
gan = Model(inputs=ganInput, outputs=ganOutput)
gan.compile(loss='binary_crossentropy', optimizer=adam)
```

Listing 1.2: Network Initilization

Το combined network που φαίνεται στις τελευταίες γραμμές του κώδικα χρησιμοποιείται για την εκπαίδευση του generator. Περεταίρω επεξήγηση επί αυτού θα γίνει στην επόμενη παράγραφο.
Παρακάτω παρουσιάζονται συνοπτικά οι λεπτομέρειες για κάθε μοντέλο.

Generator Model				
Layer	**Neurons**	**Inputs**	**Activation Function**	**Trainable Parameters**
Hidden Dense Layer 1	256	100	ReLU	100*256 + 256 = 25856
Hidden Dense Layer 2	512	256	ReLU	256*512 + 512 = 131584
Hidden Dense Layer 3	1024	512	ReLU	512 * 1024 + 1024 = 525312
Output Layer	784	1024	Tanh	1024 * 784 + 784 = 803600
				Total : 1486352

Discriminator Model				
Layer	**Neurons**	**Inputs**	**Activation Function**	**Trainable Parameters**
Hidden Dense Layer 1	1024	784	LeakyReLU	1024*784 + 1024 = 803840
Dropout Layer 1	-	1024	-	-
Hidden Dense Layer 2	512	1024	LeakyReLU	512*1024 + 512 = 524800
Dropout Layer 2	-	512	-	-
Hidden Dense Layer 3	256	512	LeakyReLU	256*512 + 256 = 131328
Dropout Layer 3	-	256	-	-
Output Layer	1	256	Sigmoid	1*256 + 1 = 257
				Total : 1460225

Αξίζει να σημειωθεί η σημασία των kernel initializers όπου έγινε χρήση του RandomNormal initializer για τις αρχικές τιμές των παραμέτρων των πρώτων επιπέδων των μοντέλων. Δίχως τη χρήση του RandomNormal initializer ή ακόμα και με τη χρήση του αλλά με μικρή τιμή τυπικής απόκλισης ως παράμετρο, η εκπαίδευση των μοντέλων πολύ συχνά δεν οδηγούσε σε επιθυμητά αποτελέσματα καθώς «κολλούσε» εύκολα σε τοπικά ελάχιστα της συνάρτησης κόστους που γίνεται απόπειρα ελαχιστοποίησης της.

1.6.3 Εκπαίδευση Μοντέλων

Για την εκπαίδευση των μοντέλων δημιουργήθηκε η συνάρτηση train, η οποία δέχεται δύο παραμέτρους, epoch και batchSize και εκπαιδεύει τα μοντέλα για πλήθος εποχών ίσο με epoch ενώ σε κάθε εποχή, κάθε ενημέρωση των παραμέτρων των μοντέλων μέσω του gradient descent γίνεται για πλήθος εικόνων ίσο με batchSize. Αναλυτικά, ο κώδικας παρουσιάζεται παρακάτω:

```
def train(epochs=1, batchSize=128):
    batchCount = X_train.shape[0] / batchSize
    print('Epochs:', epochs)
    print('Batch size:', batchSize)
    print('Batches per epoch:', batchCount)

    for e in range(1, epochs+1):
        print('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in tqdm(range(int(batchCount))):
            # Get a random set of input noise and images
            noise = np.random.normal(0, 1, size=[batchSize, randomDim])
            imageBatch = X_train[np.random.randint(0, X_train.shape[0], size=batchSize)]

            # Generate fake MNIST images
            generatedImages = generator.predict(noise)
            # print np.shape(imageBatch), np.shape(generatedImages)
            X = np.concatenate([imageBatch, generatedImages])

            # Labels for generated and real data
            yDis = np.zeros(2*batchSize)
            # One-sided label smoothing
            yDis[:batchSize] = 0.9

            # Train discriminator
            discriminator.trainable = True
            dloss = discriminator.train_on_batch(X, yDis)

            # Train generator
            noise = np.random.normal(0, 1, size=[batchSize, randomDim])
            yGen = np.ones(batchSize)
            discriminator.trainable = False
            gloss = gan.train_on_batch(noise, yGen)

        # Store loss of most recent batch from this epoch
        dLosses.append(dloss)
        gLosses.append(gloss)

        if e < 10 or e % 50 == 0:
            plotGeneratedImages(e)

    # Plot losses from every epoch
    plotLoss(e)
    saveModels(e)
```

Listing 1.3: Network Training

Η διαδικασία που υλοποιεί η συνάρτηση train ακολουθεί τη λογική που περιέγραψε στην εργασία του ο Goodfellow και έχει ως στόχο την από κοινού βελτίωση των δύο μοντέλων. Για κάθε εποχή εκπαίδευσης και για κάθε batch εικόνων γίνονται με τη σειρά τα εξής:

1. Δημιουργία θορύβου για να χρησιμοποιηθεί ως είσοδος στον generator.

2. Δημιουργία batch εικόνων επιλέγοντας τυχαία εικόνες από το training set.

3. Παραγωγή πλαστών εικόνων χρησιμοποιώντας τον generator.

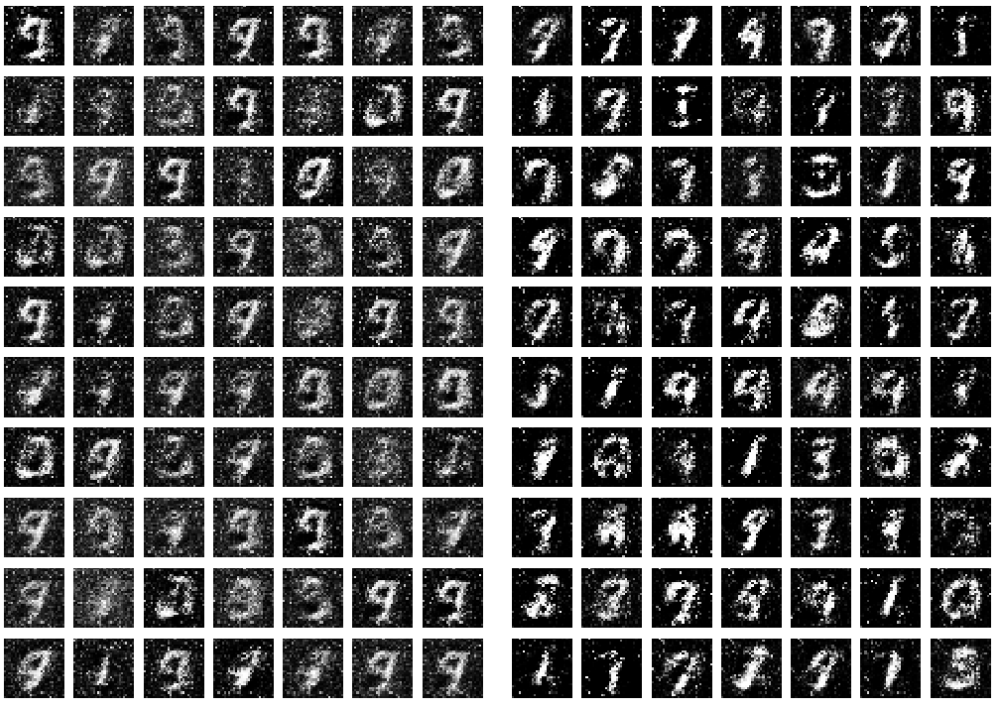
4. Σήμανση πλαστών εικόνων με το label 0 και αληθινών εικόνων με 0.9 (label smoothing).

5. Εκπαίδευση του discriminator με τις πλαστές εικόνες καθώς και με το batch αληθινών εικόνων χρησιμοποιώντας τα labels του βήματος 4.

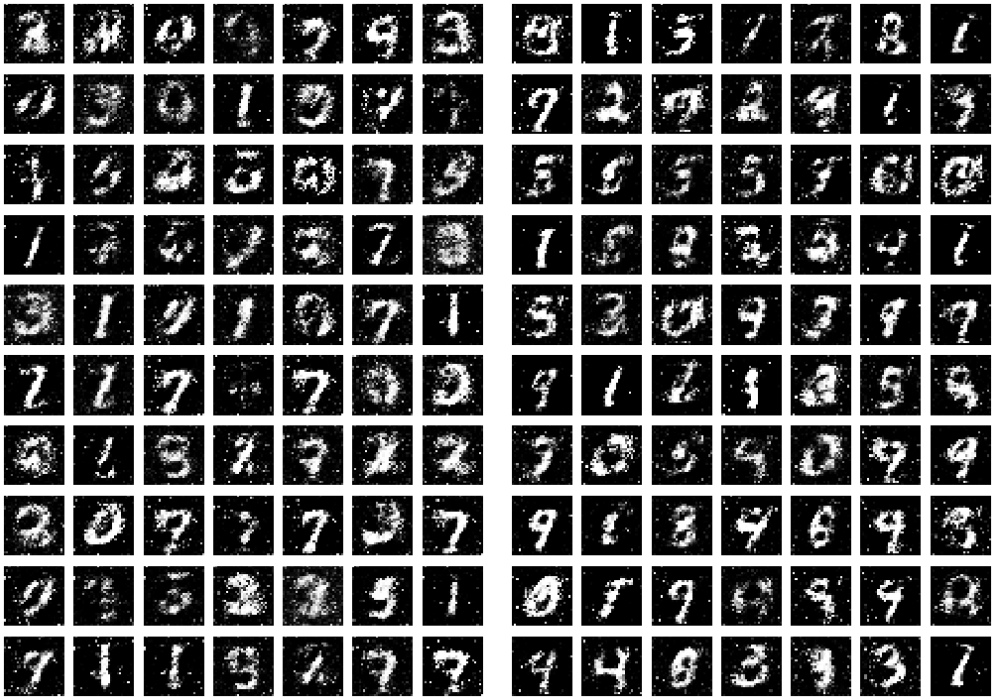
6. Εκπαίδευση του generator χρησιμοποιώντας το combined network. Συγκεκριμένα ο discriminator τίθεται ως non trainable και ο generator εκπαιδεύεται με τέτοιο τρόπο ώστε το combined network, που αποτελείται από τον discriminator σε σειρά με τον generator, να δίνει στην έξοδο του 1 (δηλαδή να αναγνωρίζει τις πλαστές εικόνες ως αληθινές).

1.6.4 Παρουσίαση Αποτελεσμάτων

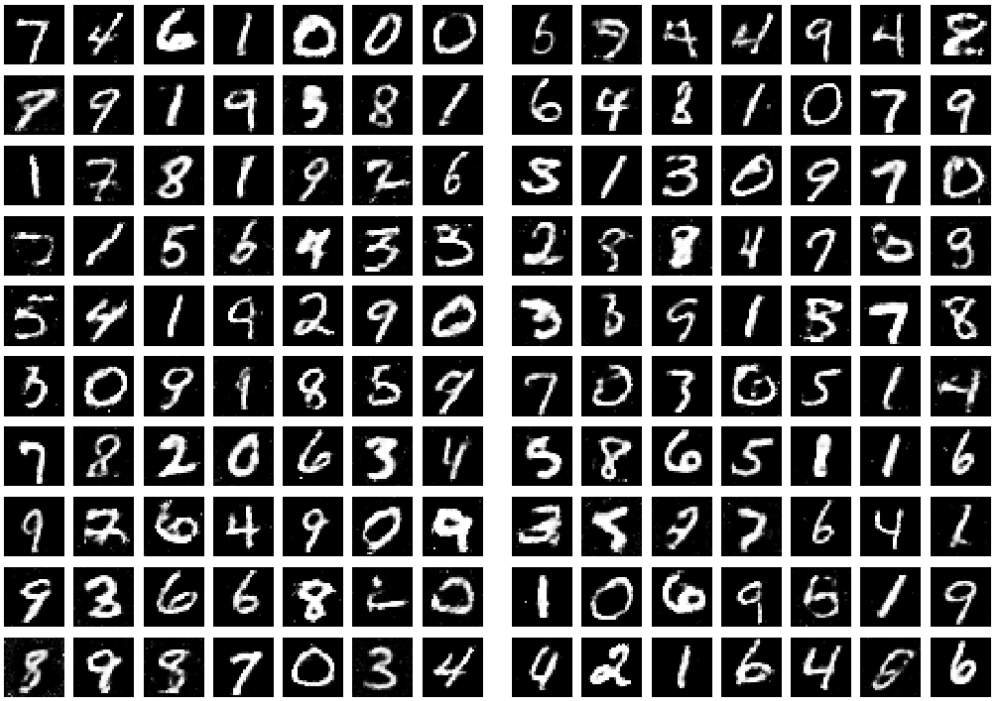
Η εκπαίδευση τον μοντέλων έγινε για 250 εποχές και για batchSize 128. Για τις πρώτες 9 εποχές εκπαίδευσης τυπώνεται μία συλλογή από 70 εικόνες παραγμένες από τον generator, για την σταδιακή επίβλεψη της βελτίωσης του μοντέλου. Ύστερα από την 9η εποχή τυπώνεται μία συλλογή κάθε 50 εποχές ενώ τα τελικά μοντέλα αποθηκεύονται για μετέπειτα χρήση. Η αποθήκευση των μοντέλων θα φανεί ιδιαίτερα χρήσιμη παρακάτω όπου θα ασχοληθούμε με την επιτάχυνση του generator σε ειδικό hardware. Τέλος, στο σχήμα 1.19 παρουσιάζεται διάγραμμα των απωλειών κάθε μοντέλου για την εκάστοτε εποχή εκπαίδευσης.



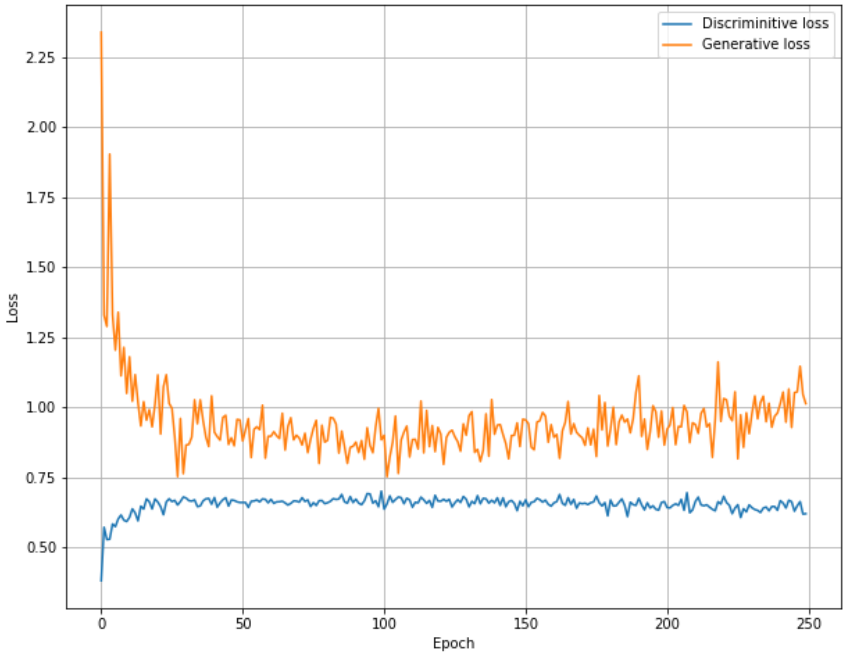
Σχήμα 1.16: Παραγόμενες από τον Generator εικόνες ύστερα από την πρώτη και τη δεύτερη εποχή εκπαίδευσης



Σχήμα 1.17: Παραγόμενες από τον Generator εικόνες ύστερα από την τρίτη και την τέταρτη εποχή εκπαίδευσης



Σχήμα 1.18: Παραγόμενες από τον Generator εικόνες ύστερα από την 100η και την 250η εποχή εκπαίδευσης



Σχήμα 1.19: Απώλειες μοντέλων για τις διάφορες εποχές εκπαίδευσης

1.7 Generative Adversarial Networks για την ανακατασκευή εικόνων

Έχοντας πειραματιστεί με την ιδέα των Generative Adversarial Networks σε ένα πρόβλημα σαν αυτό που περιέγραψε στην εργασία του ο Goodfellow, θα γίνει τροποποίηση των μοντέλων με τέτοιο τρόπο ώστε αυτή τη φορά να επιτυγχάνεται από το generative μοντέλο όχι η εκ νέου σύνθεση εικόνων βάση τυχαίου θορύβου στην είσοδο του, αλλά η ανακατασκευή ημιτελών εικόνων από το dataset. Με λίγα λόγια, η έξοδος του generator θα αποτελεί το απόν κάτω μισό μίας εικόνας από το MNIST dataset ενώ στην είσοδο του θα δέχεται το υπάρχον πάνω μισό της εικόνας, όπως στα παραδείγματα της εικόνας 1.13.

1.7.1 Προεπεξεργασία Δεδομένων

Όπως και στην προηγούμενη περίπτωση, όπου χρησιμοποιήθηκε η ιδέα των GANs για την εκ νέου σύνθεση εικόνων, το dataset κανονικοποιείται έτσι ώστε η τιμή κάθε pixel κάθε εικόνας να λαμβάνει τιμές από -1 έως 1 με τον κώδικα που υλοποιεί την κανονικοποίηση να είναι ίδιος με πριν. Ωστόσο, επειδή ο generator σε αυτήν την περίπτωση πρέπει να δέχεται ως είσοδο το πάνω μισό μέρος των εικόνων του dataset, δημιουργείται ένα ακόμα training set το οποίο περιέχει μόνο το πάνω μέρος από όλες τις εικόνες του αρχικού training set. Το ίδιο γίνεται και για το test set. Ταυτόχρονα, δημιουργείται και η συνάρτηση imageCombiner, η οποία λαμβάνει τα δύο επιμέρους κομμάτια μίας ημιτελούς εικόνας και τα ενώνει φέρνοντας την εικόνα στην τελική της μορφή. Τα παραπάνω υλοποιούνται με το εξής τμήμα κώδικα:

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5)/127.5
X_train = X_train.reshape(60000, 784)
X_test = (X_test.astype(np.float32) - 127.5)/127.5
```

```
X_test = X_test.reshape(10000, 784)

# Create set of half images

X_train_cut = X_train[:,0:int(784/2)]
X_test_cut = X_test[:,0:int(784/2)]

# Fuction to combine network output with the cut image

def image_combiner(cut, generated):

  if(len(cut.shape) == 2):
    full_image = np.concatenate((cut,generated), axis = 1)
  else:
    full_image = np.concatenate((cut,generated))
  return full_image
```

Listing 1.4: Data preprocessing

1.7.2 Δημιουργία Μοντέλων

Για την ανακατασκευή των εικόνων χρησιμοποιήθηκαν τα ίδια μοντέλα που χρησιμοποιήθηκαν και στην περίπτωση της εκ νέου σύνθεσης που παρουσιάστηκε στην προηγούμενη παράγραφο. Ωστόσο, μερικές μικρές τροποποιήσεις ήταν απαραίτητες. Συγκεκριμένα:

- Για το δίκτυο του generator οι είσοδοι του πρώτου κρυφού επιπέδου έγιναν 392 και πλέον δεν αποτελούνται από τυχαίο θόρυβο αλλά από τα pixel της μισής εικόνας του εκάστοτε αριθμητικού ψηφίου που δίνεται στην είσοδο του μοντέλου. Επιπλέον, επειδή το πρόβλημα των τοπικών ελαχίστων κατά την εκπαίδευση ήταν εντονότερο, οι αρχικές τιμές των παραμέτρων του πρώτου επιπέδου έγιναν initialize με τη χρήση του RandomNormal initializer όπως και πριν, αλλά αυτή τη φορά με μεγαλύτερη τιμή τυπικής απόκλισης.

- Το δίκτυο του discriminator είναι ολόιδιο με αυτό της προηγούμενης περίπτωσης, ωστόσο η είσοδος του είναι η ένωση των δύο τμημάτων της εικόνας, δηλαδή του πάνω μισού που δίνεται ως είσοδος στον generator, και του κάτω μισού που είναι το αποτέλεσμα που παράγει. Τέλος, όπως και στον generator, οι παράμετροι του πρώτου επιπέδου έγιναν initialize με χρήση του RandomNormal initializer αλλά με μεγαλύτερη τιμή τυπικής απόκλισης.

Η δημιουργία των μοντέλων γίνεται με το εξής τμήμα κώδικα:

```
#   Generative Network

generator = Sequential()
generator.add(Dense(256, input_dim = 392, kernel_initializer = RandomNormal(stddev=0.2)))
generator.add(ReLU())
generator.add(Dense(512))
generator.add(ReLU())
generator.add(Dense(1024))
generator.add(ReLU())
generator.add(Dense(392, activation = 'tanh'))
generator.compile(loss='binary_crossentropy', optimizer= optimizer)

# Discriminator Network

discriminator = Sequential()
discriminator.add(Dense(1024, input_dim = 784, kernel_initializer = RandomNormal(stddev=0.8)
    ))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(512))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(256))
discriminator.add(LeakyReLU(0.2))
```

```
discriminator.add(Dropout(0.3))
discriminator.add(Dense(1, activation = 'sigmoid'))
discriminator.compile(loss='binary_crossentropy', optimizer= optimizer)

# Combined Network - GAN

discriminator.trainable = False
ganInput = Input(shape=(392,))
x = generator(ganInput)
image = concatenate([ganInput,x])
ganOutput = discriminator(image)
gan = Model(inputs = ganInput, outputs = ganOutput)
gan.compile(loss='binary_crossentropy', optimizer= optimizer)
```

Listing 1.5: Data preprocessing

Στο combined network, φαίνεται η διαδικασία συνένωσης των δύο τμημάτων της τελικής εικόνας μέσω του concatenation layer, το οποίο δίνει την τελική εικόνα στον discriminator, ο οποίος κρίνει αν το τελικό προϊόν είναι αληθινό ή πλαστό.
Στους παρακάτω πίνακες παρουσιάζονται οι λεπτομέρειες κάθε δικτύου συνοπτικά:

Generator Model				
Layer	**Neurons**	**Inputs**	**Activation Function**	**Trainable Parameters**
Hidden Dense Layer 1	256	392	ReLU	392*256 + 256 = 100608
Hidden Dense Layer 2	512	256	ReLU	256*512 + 512 = 131584
Hidden Dense Layer 3	1024	512	ReLU	512 * 1024 + 1024 = 525312
Output Layer	392	1024	Tanh	1024 * 392 + 392 = 401800
				Total : 1159304

Discriminator Model				
Layer	**Neurons**	**Inputs**	**Activation Function**	**Trainable Parameters**
Hidden Dense Layer 1	1024	784	LeakyReLU	1024*784 + 1024 = 803840
Dropout Layer 1	-	1024	-	-
Hidden Dense Layer 2	512	1024	LeakyReLU	512*1024 + 512 = 524800
Dropout Layer 2	-	512	-	-
Hidden Dense Layer 3	256	512	LeakyReLU	256*512 + 256 = 131328
Dropout Layer 3	-	256	-	-
Output Layer	1	256	Sigmoid	1*256 + 1 = 257
				Total : 1460225

1.7.3 Εκπαίδευση Μοντέλων

Όπως και πριν, για την εκπαίδευση των μοντέλων δημιουργήθηκε η συνάρτηση train της οποίας ο κώδικας παρουσιάζεται παρακάτω:

```
def train(gan, generator, discriminator, path, epochs=1, batchSize=128):
    batchCount = X_train.shape[0] / batchSize
    print('Epochs:', epochs)
    print('Batch size:', batchSize)
    print('Batches per epoch:', batchCount)

    gLosses = []
    dLosses = []

    for e in range(1, epochs+1):
        print('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in tqdm(range(int(batchCount))):


```

```
15            imageCutBatch = X_train_cut[np.random.randint(0, X_train_cut.shape[0], size=
    batchSize)]
16            imageBatch = X_train[np.random.randint(0, X_train_cut.shape[0], size=batchSize)]
17
18            # Generate fake MNIST images
19            generatedHalfImages = generator.predict(imageCutBatch)
20            generatedImages = image_combiner(imageCutBatch, generatedHalfImages)
21            X = np.concatenate([imageBatch, generatedImages])
22
23            # Labels for generated and real data
24            yDis = np.zeros(2*batchSize)
25            # One-sided label smoothing
26            yDis[:batchSize] = 0.9
27
28            # Train discriminator
29            discriminator.trainable = True
30            dloss = discriminator.train_on_batch(X, yDis)
31
32            # Train generator
33            imageCutBatch = X_train_cut[np.random.randint(0, X_train_cut.shape[0], size=
    batchSize)]
34            yGen = np.ones(batchSize)
35            discriminator.trainable = False
36            gloss = gan.train_on_batch(imageCutBatch, yGen)
37
38        # Store loss of most recent batch from this epoch
39
40        dLosses.append(dloss)
41        gLosses.append(gloss)
42
43        if e < 10 or e % 50 == 0:
44            plotGeneratedImages(e, path)
45        if e == epochs:
46          saveModels(generator, discriminator, e, path = path)
47
48    # Plot losses from every epoch
49    plotLoss(e, gLosses, dLosses, path)
```

Listing 1.6: Data preprocessing

Η διαφορά της παραπάνω συνάρτησης train σε σχέση με αυτή της παραγράφου 1.6.3 έγκειται στην είσοδο του generator καθώς και στη διαχείριση της εξόδου του. Συγκεκριμένα, δεν δημιουργούνται δείγματα τυχαίου θορύβου όπως στην παράγραφο 1.6.3 αλλά επιλέγονται τυχαία δείγματα εικόνων από το training set που δημιουργήθηκε στην προεπεξεργασία των δεδομένων και περιλαμβάνει τις ημιτελείς εικόνες. Επιπλέον, οι έξοδοι του generator ενώνονται με τις ημιτελείς εικόνες δημιουργώντας μία πλήρη εικόνα και ύστερα δίνονται ως είσοδοι στο discriminator. Αναλυτικά, τα βήματα της εκπαίδευσης είναι τα παρακάτω:

1. Δημιουργία batch ημιτελών εικόνων επιλεγμένων τυχαία για να χρησιμοποιηθεί ως είσοδος στον generator.

2. Δημιουργία batch πλήρους εικόνων επιλεγμένων τυχαία από το training set.

3. Παραγωγή του απόντος μέρους των ημιτελών εικόνων χρησιμοποιώντας τον generator.

4. Συνένωση ημιτελών εικόνων με το πλαστό κάτω μέρος τους προς σύνθεση της πλήρους εικόνας.

5. Σήμανση πλαστών εικόνων με το label 0 και αληθινών εικόνων με 0.9 (label smoothing).

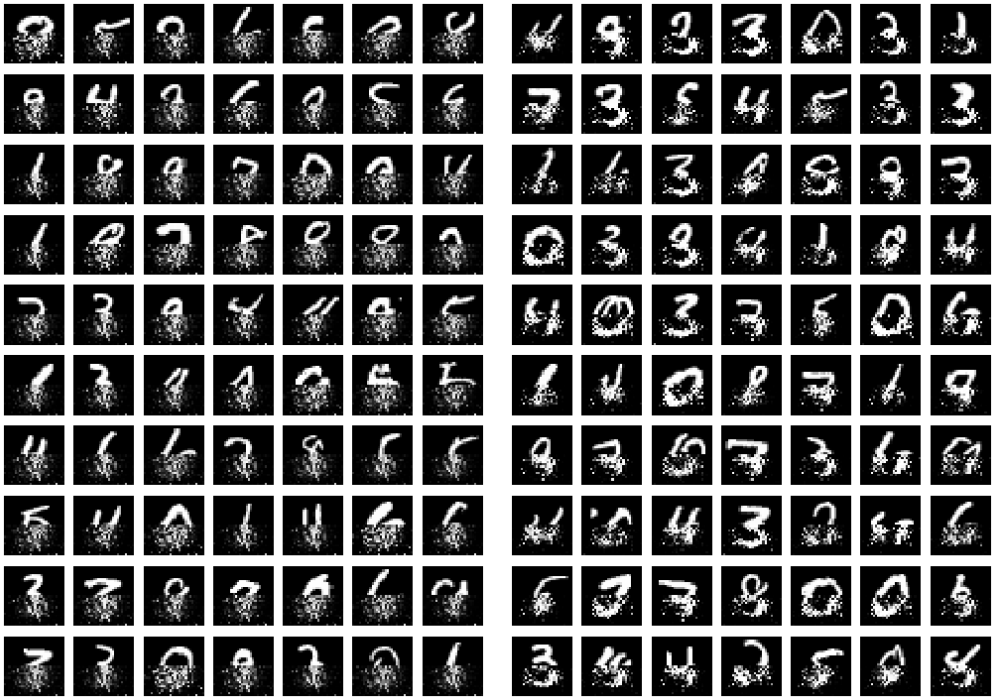
6. Εκπαίδευση του discriminator με τις πλαστές εικόνες καθώς και με το batch αληθινών εικόνων χρησιμοποιώντας τα labels που προαναφέρθηκαν.

7. Εκπαίδευση του generator χρησιμοποιώντας το combined network. Συγκεκριμένα ο discriminator τίθεται ως non trainable και ο generator εκπαιδεύεται με τέτοιο τρόπο ώστε το combined network, που

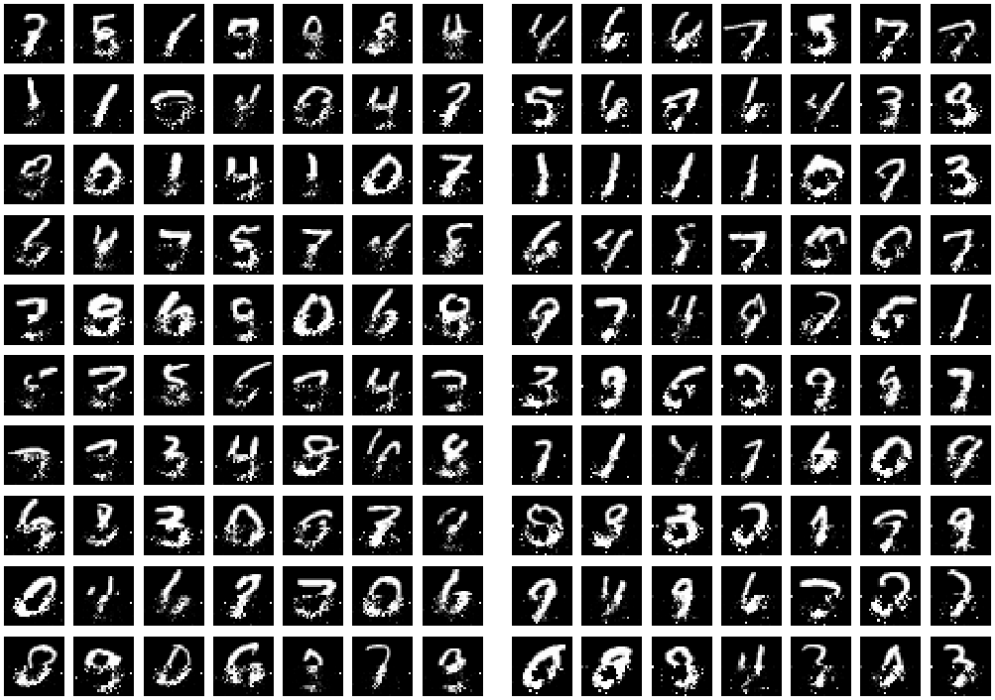
αποτελείται από τον discriminator σε σειρά με τον generator, να δίνει στην έξοδο του 1 (δηλαδή να αναγνωρίζει τις πλαστές εικόνες ως αληθινές).

1.7.4 Παρουσίαση Αποτελεσμάτων

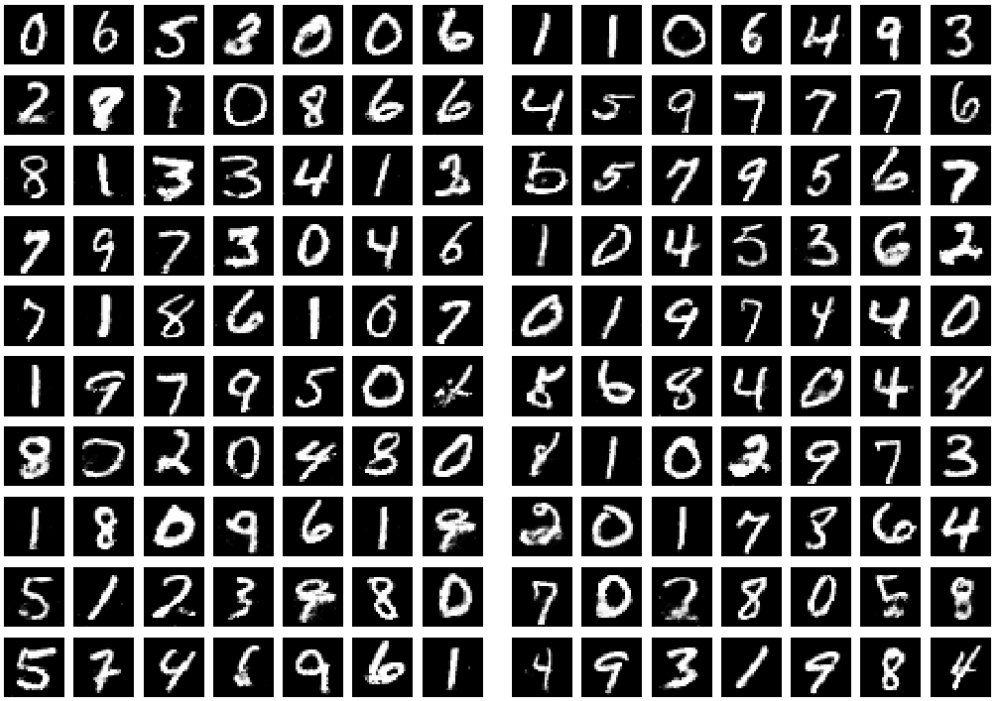
Όπως και στην παράγραφο 1.5.4 γίνεται εκπαίδευση των μοντέλων για 250 εποχές για batchSize ίσο με 128. Στα σχήματα 1.20 και 1.21 παρουσιάζονται συλλογές 70 ανακατασκευασμένων εικόνων για τις πρώτες 4 εποχές εκπαίδευσης, ενώ στο σχήμα 1.22 για τις εποχές 100 και 250. Τέλος, στο σχήμα 1.23 παρουσιάζονται οι απώλειες των δύο μοντέλων καθ' όλη τη διάρκεια της εκπαίδευσης τους.



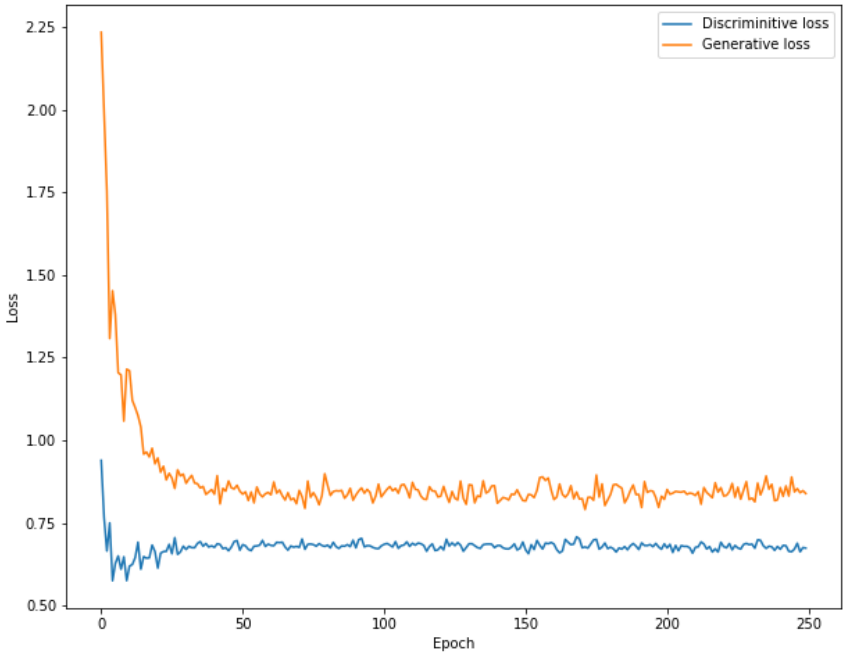
Σχήμα 1.20: Παραγόμενες από τον Generator εικόνες ύστερα από την πρώτη και τη δεύτερη εποχή εκπαίδευσης



Σχήμα 1.21: Παραγόμενες από τον Generator εικόνες ύστερα από την τρίτη και την τέταρτη εποχή εκπαίδευσης

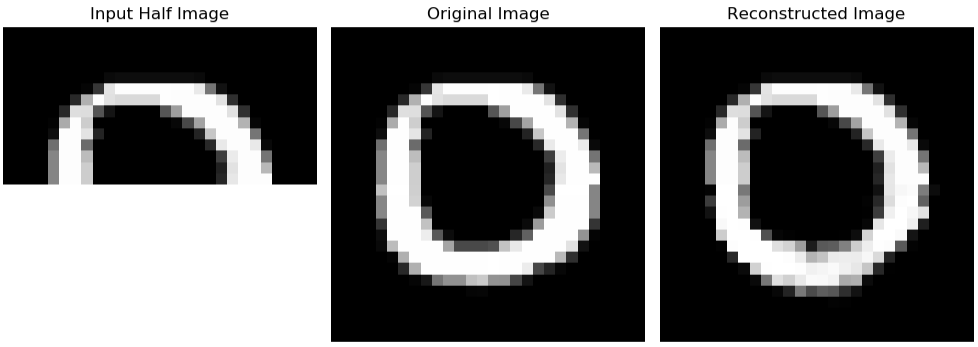


Σχήμα 1.22: Παραγόμενες από τον Generator εικόνες ύστερα από την 100η και την 250η εποχή εκπαίδευσης

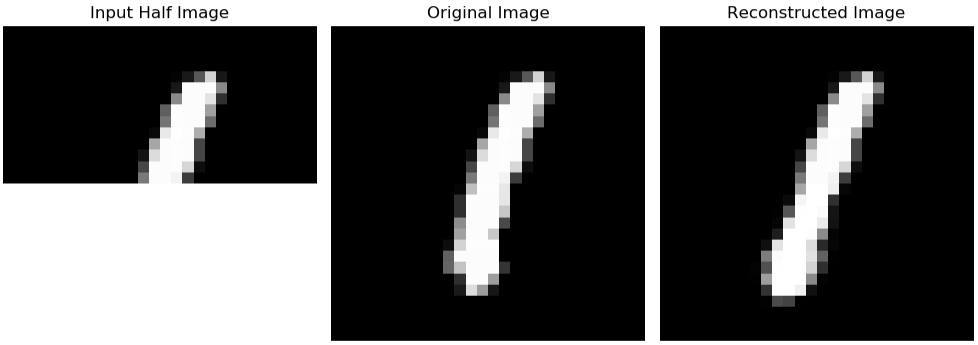


Σχήμα 1.23: Απώλειες μοντέλων για τις διάφορες εποχές εκπαίδευσης

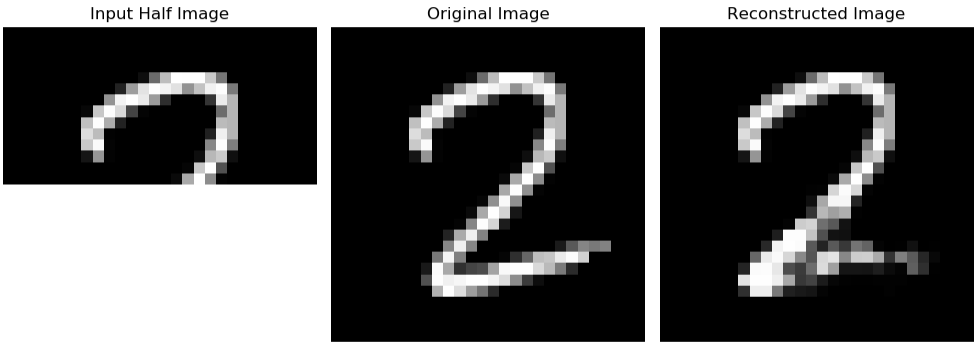
Μεγάλο ενδιαφέρον εγείρει η σύγκριση μεταξύ των αυθεντικών και τω ν ανακατασκευασμένων εικόνων. Στα παρακάτω σχήματα παρουσιάζονται αρχικά η ημιτελής εικόνα που ο generator καλείται να ανακατασκευάσει καθώς και η σύγκριση μεταξύ αυθεντικής και επιτυχημένα ανακατασκευασμένης εικόνας για διάφορα παραδείγματα από το test set.



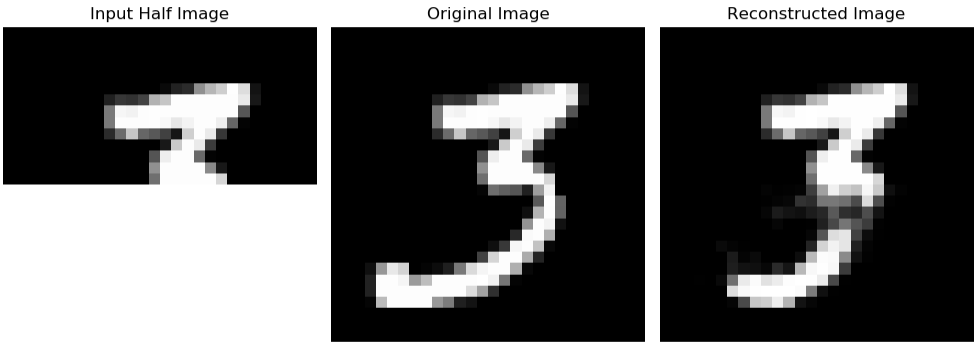
Σχήμα 1.24: Ανακατασκευή αριθμητικού ψηφίου 0



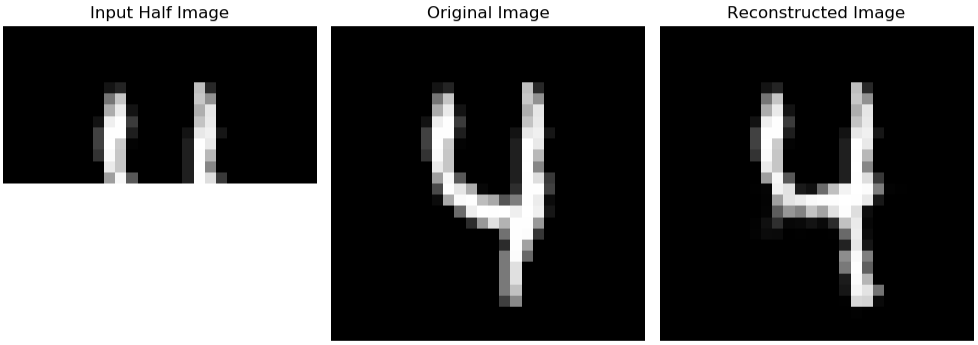
Σχήμα 1.25: Ανακατασκευή αριθμητικού ψηφίου 1



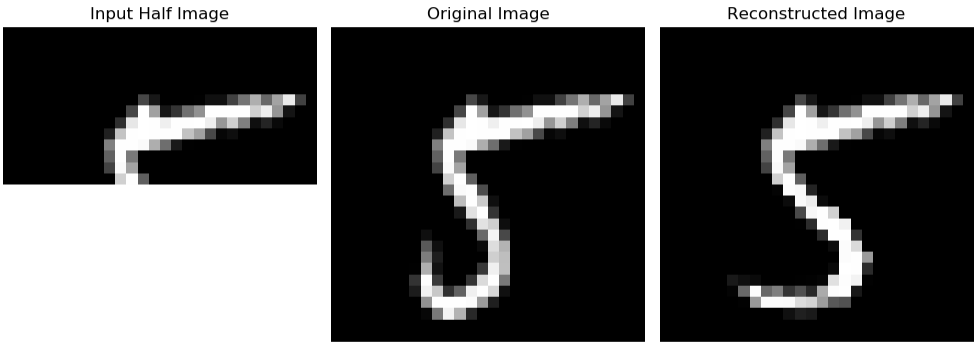
Σχήμα 1.26: Ανακατασκευή αριθμητικού ψηφίου 2



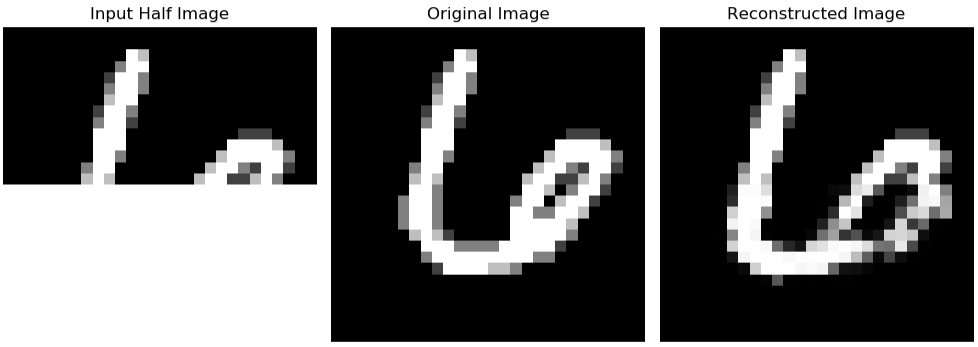
Σχήμα 1.27: Ανακατασκευή αριθμητικού ψηφίου 3



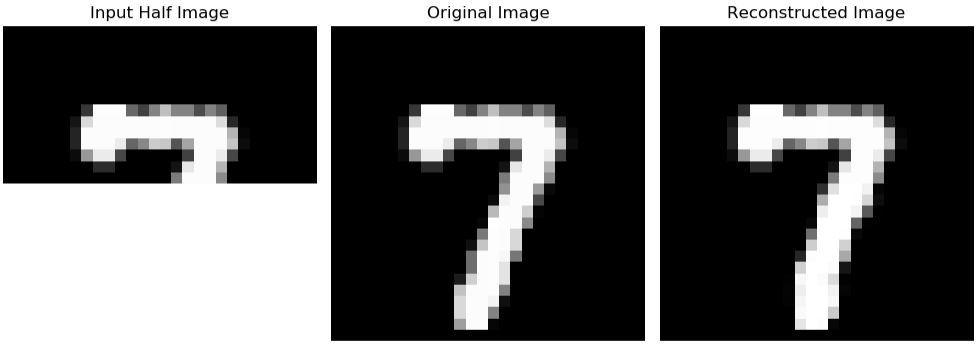
Σχήμα 1.28: Ανακατασκευή αριθμητικού ψηφίου 4



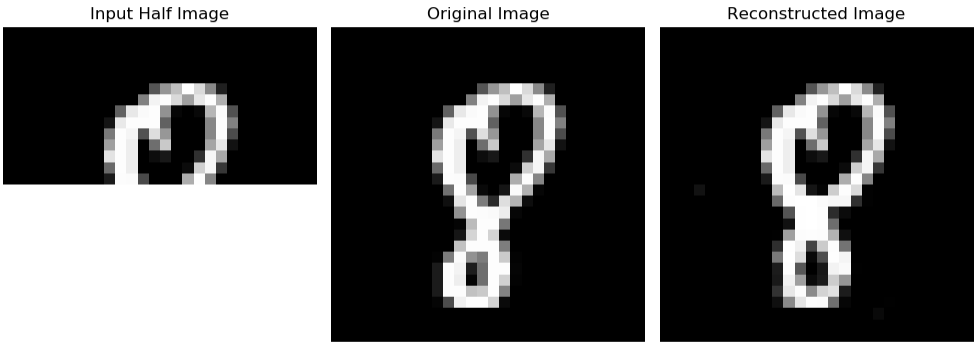
Σχήμα 1.29: Ανακατασκευή αριθμητικού ψηφίου 5



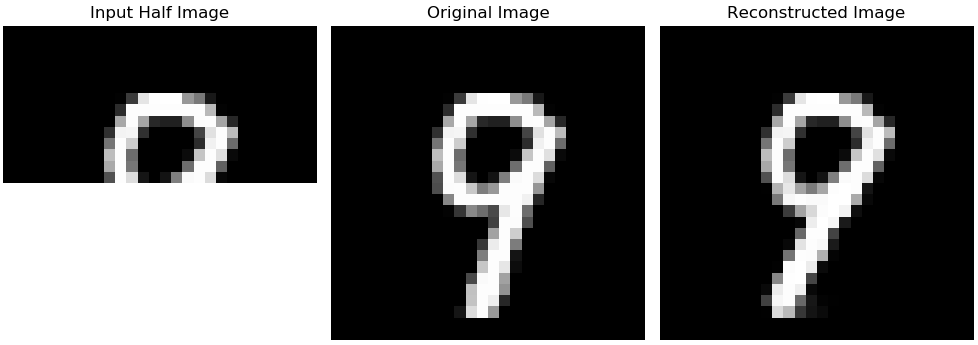
Σχήμα 1.30: Ανακατασκευή αριθμητικού ψηφίου 6



Σχήμα 1.31: Ανακατασκευή αριθμητικού ψηφίου 7

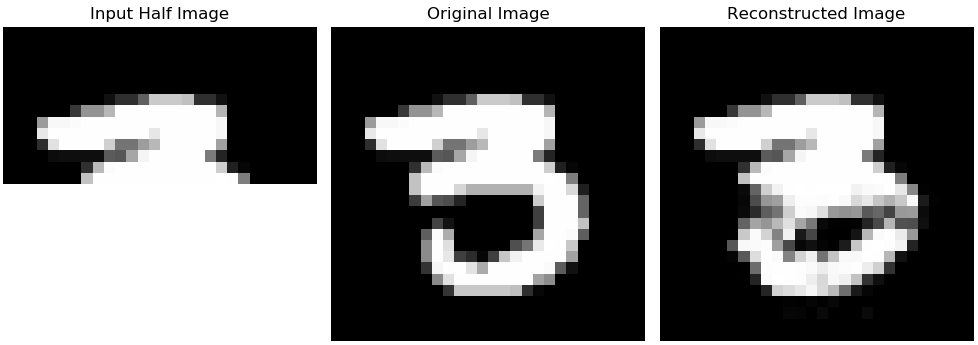


Σχήμα 1.32: Ανακατασκευή αριθμητικού ψηφίου 8

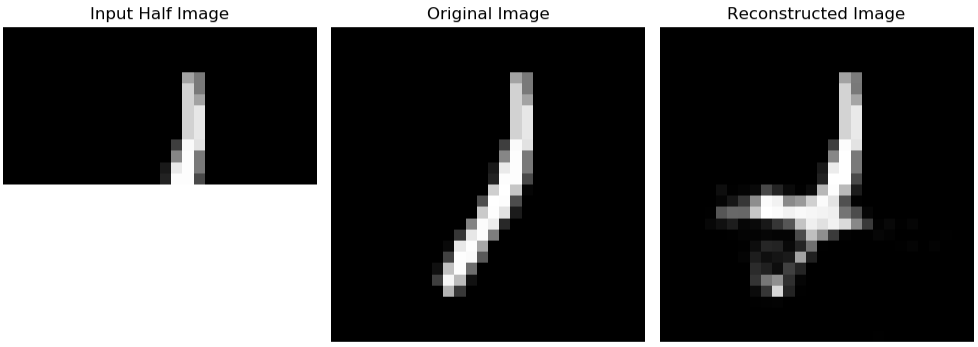


Σχήμα 1.33: Ανακατασκευή αριθμητικού ψηφίου 9

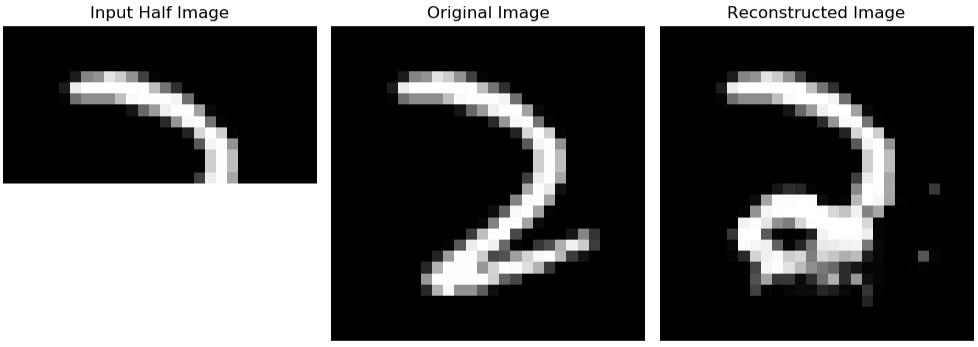
Ωστόσο, όπως φαίνεται και στις συλλογές εικόνων του σχήματος 1.22, δεν είναι επιτυχημένες όλες οι ανακατασκευές. Στα σχήματα 1.34 & 1.35 παρουσιάζονται μερικά παραδείγματα αποτυχημένων ανακατασκευών.



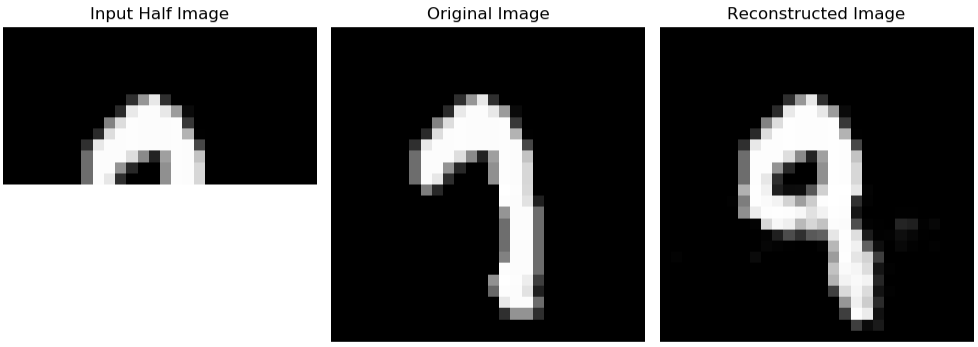
Σχήμα 1.34: Αδυναμία νακατασκευής αριθμητικού ψηφίου 3



Σχήμα 1.35: Αδυναμία ανακατασκευής αριθμητικού ψηφίου 1



Σχήμα 1.36: Αδυναμία ανακατασκευής αριθμητικού ψηφίου 2



Σχήμα 1.37: Αδυναμία ανακατασκευής αριθμητικού ψηφίου 1

Μέρος 2

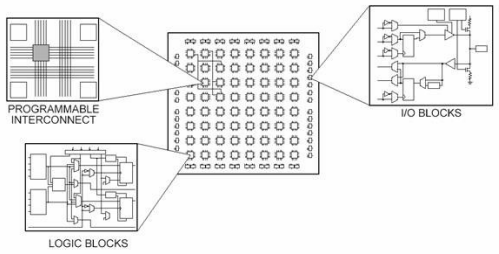
Επιτάχυνση Υπολογισμών Μοντέλου

2.1 Field Programmable Gate Arrays (FPGAs)

Ένα Field Programmable Gate Array (FPGA) αποτελεί ολοκληρωμένο κύκλωμα με δυνατότητα επαναπρογραμματισμού της εσωτερικής του διαμόρφωσης. Ο επαναπρογραμματισμός γίνεται με τη χρήση γλωσσών περιγραφής υλικού (Hardware Description Languages). Στο εσωτερικό του, ένα FPGA περιέχει πλήθος από διαμορφώσιμα λογικά blocks (configurable logic blocks) καθώς και ένα σύνολο από διαμορφώσιμες διασυνδέσεις μεταξύ τους. Τα λογικά blocks δύνανται να διαμορφωθούν έτσι ώστε να υλοποιούν απλές και σύνθετες λογικές συναρτήσεις συνδυαστικής λογικής. Ο δυναμικός χαρακτήρας του FPGA είναι η βασική διαφοροποίηση του από ένα ASIC (Application Specific Integrated Circuit), όπου η εσωτερική του διαμόρφωση παραμένει στατική υλοποιώντας μόνο μία λειτουργία. Η πρώτη συσκευή επαναπρογραμαμτιζόμενης λογικής (EP300) έγινε διαθέσιμη το 1984 από την εταιρεία Altera. Διέθετε ένα παράθυρο χαλαζία (quartz window) που επέτρεπε στους χρήστες να σβήσουν από τα κελιά EPROM τις πληροφορίες για τη διαμόρφωση της συσκευής απλά με την έκθεση σε υπεριώδες φως. Έκτοτε, ο επαναπρογραμματισμός έχει γίνει ευκολότερος με την εκμετάλλευση μνημών SRAM, οι οποίες αποθηκεύουν τις απαραίτητες πληροφορίες (bitstream) για την διαμόρφωση του FPGA και τη χρήση ειδικών κυκλωμάτων υπεύθυνων για την ανάγνωση του bitstream και την διαμόρφωση του FPGA σύμφωνα με αυτό.

2.1.1 Εσωτερική Δομή

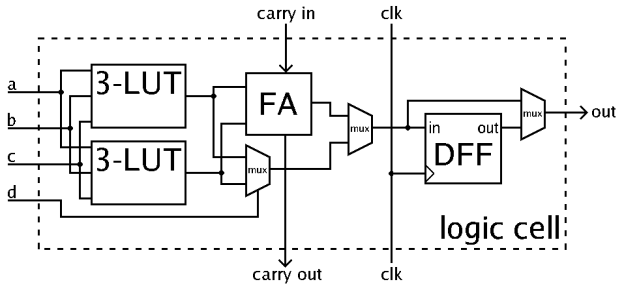
Η εσωτερική δομή ενός μη προγραμματισμένου FPGA δεν κρύβει καμία λογική καθώς αποτελείται από μεγάλο αριθμό επιμέρους blocks, τα οποία με κατάλληλη διαμόρφωση τους και κατάλληλη μεταξύ τους διασύνδεση υλοποιούν εν τέλει τη λογική που επιθυμεί ο χρήστης να προσδώσει στο hardware. Στην εικόνα 2.1 παρουσιάζεται μια απλοιποιημένη μορφή της εικόνας ενός FPGA, ενώ παρακάτω παρουσιάζονται τα διαφορετικά είδη block που εμφανίζονται στο εσωτερικό του.



Σχήμα 2.1: Απλοποιημένη αναπαράσταση εσωτερικής δομής FPGA [17]

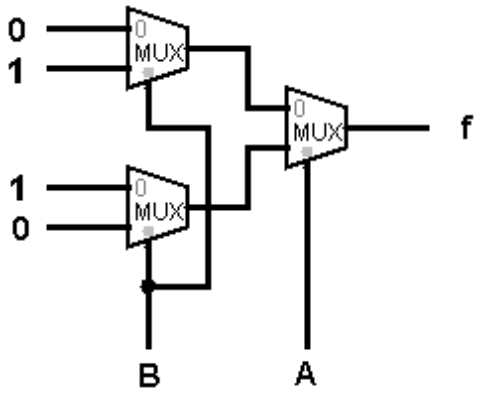
- Configurable Logic Blocks

  Τα configurable logic blocks αποτελούν ουσιαστικά τη θεμελιώδη μονάδα για υλοποίηση λογικής που διαθέτει ένα FPGA και σε σύγχρονα chips εμφανίζεται κατά χιλιάδες. Κάθε configurable logic block αποτελείται από Look Up Tables (LUTs), flip-flops, πολυπλέκτες και αθροιστές όπως εμφανίζεται στην εικόνα 2.2 μία άποψη τους.



Σχήμα 2.2: Άποψη εσωτερικής δομής ενός configurable logic block [18]

Τα LUTs δομούνται από σειριακά τοποθετημένους πολυπλέκτες των οποίων η τελική έξοδος εξαρτάται από τον τρόπο όπου έχει προγραμματιστεί το chip. Στην εικόνα 2.3 παρουσιάζεται ένα παράδειγμα χρήσης LUT δύο εισόδων για την υλοποίηση της λογικής συνάρτησης $f = A \bigoplus B$.



Σχήμα 2.3: Υλοποίηση λογικής συνάρτησης $f = A \bigoplus B$

Είναι αυτονόητο ότι με κατάλληλη ρύθμιση των εισόδων των πολυπλεκτών του πρώτου επιπέδου μπορεί να υλοποιηθεί οποιαδήποτε λογική συνάρτηση δύο μεταβλητών. Με χρήση επιπλέον LUT δύναται η υλοποίηση συναρτήσεων περισσότερων μεταβλητών.

Επιπλέον, η χρήση του Full Adder μπορεί να παρακαμφθεί ξανά με κατάλληλη ρύθμιση του πολυπλέκτη που βρίσκεται ακριβώς μετά από αυτόν όπως και του D flip-flop για την υλοποίηση μίας καθαρά συνδυαστικής λογικής σε περίπτωση που αυτό είναι επιθυμητό. Η ρύθμιση των επιμέρους πολυπλεκτών γίνεται μέσω του bitstream το οποίο παράγεται μετά το compilation του αντίστοιχου κώδικα HDL.

- Hard Logic Blocks

  Τα Hard Logic Blocks αποτελούν blocks τα οποία σε αντίθεση με τα CLBs υλοποιούν προκαθορισμένη λογική. Τέτοια blocks μπορούν να είναι Digital Signal Processors (DSPs), CPUs, multipliers, high speed transceivers. Η ύπαρξη τους οφείλεται στη συχνή ανάγκη για χρήση τους. Προκειμένου ο χρήστης να μην είναι αναγκασμένος να σχεδιάζει εκ νέου κάποιο από τα παραπάνω components, αυτά προσφέρονται έτοιμα για χρήση μέσα στο chip του FPGA.

- Configurable Interconnections

Αποτελούν δίκτυο καλωδιώσεων για την κατάλληλη διασύνδεση των επιμέρους blocks ελεγχόμενη από το bitstream που παράγεται από τον κώδικα HDL του χρήστη, με στόχο την υλοποίηση της επιθυμητής λογικής.

- I/O Blocks

  Τα blocks αυτά προσφέρουν την ικανότητα στο FPGA για μεταφορά δεδομένων από αυτό και προς αυτό ενώ διαθέτουν και αυτά δυνατότητα για configuration.

- Block RAMs

  Αποτελούν blocks μνήμης τα οποία μπορούν να λειτουργήσουν ανεξάρτητα ή να σχηματίσουν μια ενιαία μνήμη. Μεγάλο όφελος της χρήσης τους είναι η ικανότητα ανάγνωσης και εγγραφής ταυτόχρονα από διαφορετικά blocks προσφέροντας παραλληλοποίηση των προσβάσεων στη μνήμη.

2.1.2 Εφαρμογές & Πλεονεκτήματα

Γενικότερα τα FPGA μπορούν να χρησιμοποιηθούν για την επίλυση οποιουδήποτε υπολογίσιμου προβλήματος και αυτό μπορεί να αποδειχθεί από το γεγονός ότι μπορούν να χρησιμοποιηθούν για την υλοποίηση soft-microprocessors. Το πλεονέκτημα τους ωστόσο έγκειται στο γεγονός ότι είναι σημαντικά ταχύτερα για ορισμένες εφαρμογές εξαιτίας της παράλληλης φύσης τους καθώς και του βέλτιστου αριθμού πυλών που χρησιμοποιούν για συγκεκριμένες εφαρμογές. [18]
Η δυνατότητα των FPGA για πραγματικά παράλληλους υπολογισμούς έχει δημιουργήσει την τάση για χρήση τους ως hardware accelerators. Συγκεκριμένα, μπορεί κανείς να τα χρησιμοποιήσει για την επιτάχυνση συγκεκριμένων τμημάτων ενός αλγορίθμου και στη συνέχεια να μοιραστεί τα αποτελέσματα του υπολογισμού με έναν επεξεργαστή για τη μετέπειτα εκτέλεση του. Επιπλέον, τα FPGA κατέχουν σημαντικό ρόλο στην ανάπτυξη ενσωματωμένων συστημάτων καθώς επιτρέπουν την ανάπτυξη του λογισμικού του συστήματος ταυτόχρονα με την ανάπτυξη του hardware, επιτρέπουν εκτιμήσεις της απόδοσης του συστήματος ακόμα και στα πρώιμα στάδια της ανάπτυξης του και καθιστούν δυνατές αλλαγές και δοκιμές πριν την τελική επιλογή της αρχιτεκτονικής του συστήματος [18]. Μερικές εφαρμογές που βρίσκουν ευρεία χρήση είναι:

- Σε συστήματα ενσύρματων (Optical Networks) αλλά και ασύρματων επικοινωνιών (4G & 5G Mobile Base Stations) εξαιτίας της ικανότητας τους για επιτάχυνση αλγορίθμων επεξεργασίας σήματος.

- Ως επιταχυντές νευρωνικών δικτύων για εφαρμογές μηχανικής μάθησης εκμεταλλευόμενοι την παράλληλη φύση τους.

- Ως εργαλείο για την ανάπτυξη ολοκληρωμένων κυκλωμάτων όπου εκμεταλλευόμενοι την ικανότητα τους για επαναπρογραμματισμό χρησιμοποιούνται για τη σχεδίαση πρωτοτύπων ASICs και την μίμηση τμημάτων hardware υπολογιστών.

- Σε κέντρα δεδομένων (data centers), όπου η απόδοση και η μικρή κατανάλωση ενέργειας σε servers, routers, switches και gateways κατέχουν σημαντικό ρόλο.

- Σε διαστημικές εφαρμογές όπου είναι επιθυμητή η χαμηλή κατανάλωση ενέργειας εκμεταλλευόμενοι τη χαμηλή κατανάλωση που παρουσιάζουν τα FPGAs συγκριτικά με άλλους accelerators όπως οι κάρτες γραφικών (GPUs).

- Σε εφαρμογές που αφορούν image processing και high resolution video display.

2.2 Παρουσίαση Προβλήματος

Στο πρώτο μέρος παρουσιάστηκε η διαδικασία αναζήτησης και εκπαίδευσης ενός νευρωνικού δικτύου το οποίο είναι ικανό να αναπαράγει το απόν τμήμα ημιτελών εικόνων από το MNIST dataset. Το τελικό μοντέλο περιέχει 3 κρυφά επίπεδα νευρώνων, το επίπεδο νευρώνων εξόδου και συνολικά 1159304 παραμέτρους, όπως παρουσιάστηκε. Στην θεωρία που αναπτύχθηκε στο πρώτο μέρος, έγινε γνωστό ότι κάθε νευρώνας διαθέτει ένα set παραμέτρων, τα βάρη, που τον χαρακτηρίζουν, καθώς και το κατώφλι ενεργοποίησης του. Η έξοδος του δίνεται από τη σχέση:

$$y_i = f\left(\sum_{i=1}^{m} w_{ji}x_i + b_j\right) = f\left(W_j X_j^T + b_j\right)$$

Όπου W_j το διάνυσμα βαρών του νευρώνα, X_j το διάνυσμα εισόδων του και b_j ο σταθερός όρος, ή αλλιώς το κατώφλι ενεργοποίησης του. Για ένα ολόκληρο επίπεδο νευρώνων επομένως, το διάνυσμα που περιέχει την έξοδο κάθε νευρώνα, και αποτελεί διάνυσμα εισόδου για κάθε νευρώνα του επόμενου επιπέδου, είναι:

$$\Upsilon = f\left(XW + B\right)$$

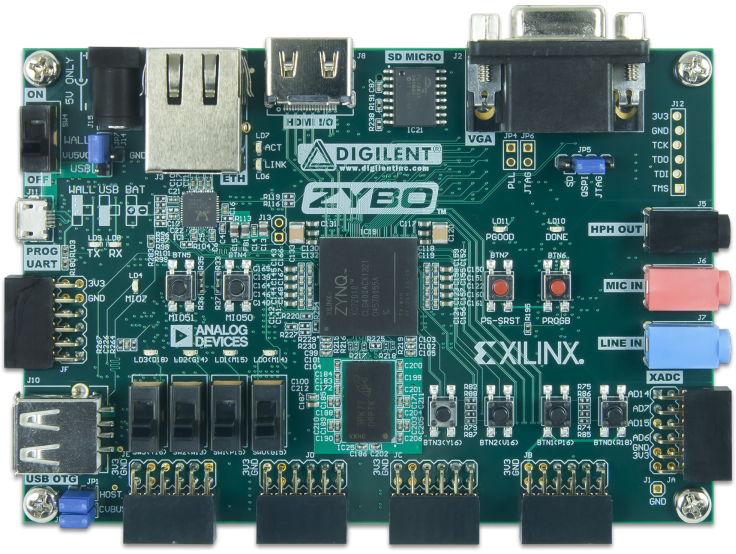
Όπου, X το διάνυσμα εισόδου m στοιχείων του κάθε νευρώνα (ίδιο για κάθε νευρώνα εφόσον κάθε ένας δέχεται σαν είσοδο όλες τις εξόδους του προηγούμενου επιπέδου), W ο πίνακας $m * n$, με n το πλήθος νευρώνων του επιπέδου και m το πλήθος εισόδων κάθε νευρώνα, και B το διάνυσμα n στοιχείων με το κατώφλι του εκάστοτε νευρώνα στην αντίστοιχη θέση.
Όπως είναι κατανοητό, ένα πολυεπιπέδο νευρωνικό δίκτυο περιγράφεται μαθηματικά από διαδοχικούς πολλαπλασιασμούς και αθροίσεις πινάκων και διανυσμάτων με την εφαρμογή μίας συνάρτησης ενεργοποίησης στα αποτελέσματα κάθε επιπέδου. Η όλη διαδικασία υπολογισμού της εξόδου του νευρωνικού δικτύου καλείται forward propagation εξαιτίας της ροής της πληροφορίας από το πρώτο κρυφό επίπεδο προς το επίπεδο εξόδου. Ο πολλαπλασιασμός ενός πίνακα με ένα διάνυσμα σε ένα πρόγραμμα που εκτελείται σειριακά σε CPU, έχει πολυπλοκότητα $O(n^2)$. Η πολυπλοκότητα αυτή είναι αρκετή ώστε σε ένα σύστημα όπου χρησιμοποιείται ένα τέτοιο δίκτυο με μεγάλο πλήθος νευρώνων να παρουσιάζεται μεγάλη καθυστέρηση εξόδου. Αν υποθέσουμε ότι το σύστημα αυτό θα πρέπει να κάνει real time ανακατασκευή εικόνων οι οποίες εισέρχονται στην είσοδο από κάποια κάμερα, θα ήταν επιθυμητή η άμεση απόκριση της εξόδου.
Το δεύτερο αυτό μέρος, αφορά την επιτάχυνση των υπολογισμών που υλοποιεί το νευρωνικό δίκτυο που αναπτύχθηκε, με τη χρήση FPGA. Στην επόμενη παράγραφο γίνεται αναλυτική παρουσίαση των εργαλείων που χρησιμοποιήθηκαν γι' αυτό τον σκοπό, ενώ στην παράγραφο 2.4 γίνεται παρουσίαση της διαδικασίας επιτάχυνσης.

2.3 Παρουσίαση Εργαλείων

Για την επιτάχυνση των υπολογισμών που πραγματοποιεί το νευρωνικό δίκτυο χρησιμοποιήθηκε το ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board, το οποίο αποτελεί εκπαιδευτική πλακέτα για την ανάπτυξη εφαρμογών σε FPGA. Όντας εκπαιδευτική πλακέτα με περιορισμένους πόρους, ήταν αναγκαία η συρρίκνωση του νευρωνικού δικτύου προκειμένου να ανταποκρίνεται στους περιορισμούς που επέβαλε το hardware. Για την περιγραφή του κυκλώματος που πραγματοποιεί την επιτάχυνση των υπολογισμών γίνεται χρήση High Level Synthesis (ή αλλιώς C synthesis), όπου αποτελεί αυτοματοποιημένη διαδικασία σχεδίασης ικανή να «μεταφράσει» έναν αλγόριθμο σε ψηφιακό κύκλωμα που αναπαράγει την ίδια συμπεριφορά. Τέλος, γίνεται ευρεία χρήση του SDx IDE της Xilinx για την ανάπτυξη του αλγορίθμου και την περιγραφή του υλικού που υλοποιεί τους υπολογισμούς του νευρωνικού δικτύου. Στις επόμενες παραγράφους ακολουθεί αναλυτική περιγραφή των εργαλείων που αναφέρθηκαν.

2.3.1 Zybo Zynq-7000 ARM/FPGA SoC Trainer Board

Το ZYBO board αποτελεί πλατφόρμα ανάπτυξης ενσωματωμένου software και ψηφιακών κυκλωμάτων της Digilent και διαθέτει το μικρότερο μέλος της οικογένειας chip Xilinx Zynq-7000, το Z-7010. Το Z-7010 βασίζεται στην αρχιτεκτονική All Programmable System-on-Chip (AP SoC) της Xilinx. Ενσωματώνει διπύρηνο επεξεργαστή ARM Cortex-A9 με τη Xilinx 7-series Field Programmable Gate Array (FPGA) λογική. [19], [20]

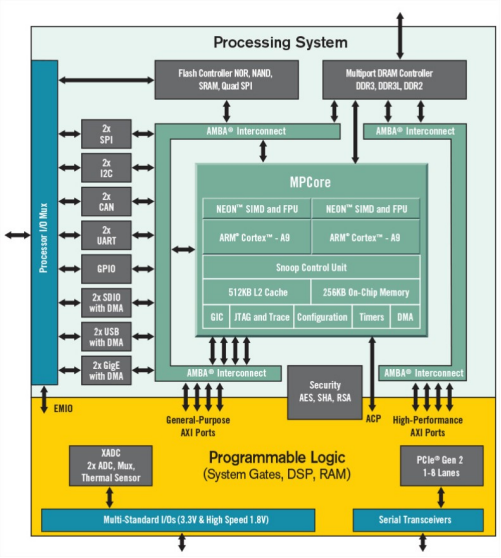


Σχήμα 2.4: Άποψη του ZYBO board της Digilent

Στον παρακάτω πίνακα παρουσιάζεται η διαθεσιμότητα των βασικών πόρων του FPGA του συγκεκριμένου chip. [20]

Recource	#
Programmable Logic Blocks	28000
Look-Up Tables (LUTs)	17600
Flip-Flops	35200
Block RAM (36Kb Blocks)	60 (2.1Mb)
DSP Slices	80

Η παρουσία επεξεργαστή και προγραμματιζόμενης λογικής στο ίδιο chip μας επιτρέπει την ανάπτυξη εφαρμογών όπου γίνεται ανάθεση υπολογιστικά έντονων διεργασιών στην προγραμματιζόμενη λογική, ενώ στη συνέχεια τα αποτελέσματα που παράγονται σε αυτή μεταφέρονται στον επεξεργαστή για περεταίρω επεξεργασία και χρήση. Η επικοινωνία μεταξύ προγραμματιζόμενης λογικής και επεξεργαστή γίνεται μέσω θυρών AXI, οι οποίες υλοποιούν το πρωτόκολλο επικοινωνίας AXI. Στο σχήμα 2.5 παρουσιάζεται το block διάγραμμα των επιμέρους στοιχείων που συνθέτουν το chip.
Τέλος, η πλατφόρμα διαθέτει μεγάλη γκάμα θυρών για συνδεσιμότητα και είσοδο-έξοδο δεδομένων. Κλασσικοί διακόπτες και κουμπιά σε συνδυασμό με LEDs συντελούν στην ανάπτυξη απλών εφαρμογών, ενώ υπάρχει δυνατότητα για είσοδο και έξοδο ήχου από θύρες μικροφώνων και ακουστικών, μεταφορά πληροφορίας εικόνας και video από τις θύρες HDMI & VGA, σύνδεση με δίκτυα χάρη στην θύρα Ethernet καθώς και για σύνδεση με περιφερειακές συσκευές μέσω των 6 Pmod ports.



Σχήμα 2.5: Block διάγραμμα των chips της οικογένειας Zynq-7000 [21]

2.3.2 High Level Synthesis

Η σύνθεση υψηλού επιπέδου (High-level synthesis) αποτελεί αυτοματοποιημένη διαδικασία σχεδίασης κατά την οποία γίνεται μετάφραση της αλγοριθμικής περιγραφής μίας επιθυμητής συμπεριφοράς σε ψηφιακό hardware, το οποίο υλοποιεί αυτή τη συμπεριφορά. Ερευνητικές και εμπορικές εφαρμογές δέχονται συνήθως ως γλώσσες περιγραφής του αλγορίθμου υποσύνολα των ANSI C/C++ , System C ή Matlab. Ο κώδικας περιγραφής αναλύεται και εν τέλει μεταφράζεται σε ένα register-transfer level (RTL) σχέδιο μίας γλώσσας περιγραφής υλικού, που με τη σειρά του μετατρέπεται στο επίπεδο πυλών (gate level) με τη χρήση εργαλείων λογικής σύνθεσης. [22], [23]

Στόχος της σύνθεσης υψηλού επιπέδου είναι να επιτρέψει στους hardware designers να αναπτύσσουν και να κάνουν επαλήθευση της λειτουργίας του hardware αποδοτικά, δίνοντας τους αρχικά την ευχέρεια να περιγράψουν τη σχεδίαση σε ένα υψηλότερο επίπεδο αφαιρετικότητας και επιπλέον, ένα καλύτερο έλεγχο των βελτιστοποιήσεων της αρχιτεκτονικής του σχεδίου. [22]

Γενικά, ένας αλγόριθμος δύναται να ολοκληρωθεί σε περισσότερους κύκλους ρολογιού με τη χρήση λιγότερων πόρων hardware ή σε λιγότερους κύκλους ρολογιού με τη χρήση περισσότερων πόρων. Συνεπώς, μια αλγοριθμική περιγραφή μπορεί να μεταφραστεί σε πολλές διαφορετικές αρχιτεκτονικές hardware οι οποίες υλοποιούν την ίδια συμπεριφορά. Για τον έλεγχο της τελικής αρχιτεκτονικής γίνεται χρήση οδηγιών (directives), οι οποίες δίνονται στο εργαλείο. Με τα directives γίνεται δυνατός ο έλεγχος της καθυστέρησης, του throughput, της επιφάνειας καθώς και του ποσοστού χρησιμοποίησης των συνολικών διαθέσιμων πόρων του τελικού design. [22], [23]

Στην παρούσα εργασία γίνεται χρήση της σύνθεσης υψηλού επιπέδου της Xilinx, γνωστή ως Vivado High-Level Synthesis. Η Xilinx με αυτό το εργαλείο κάνει δυνατή την επιτάχυνση της σχεδίασης IP, με την περιγραφή της συμπεριφοράς τους σε C, C++ και System C. [23]

2.3.3 Xilinx SDSoC IDE 2017.1

Το ολοκληρωμένο περιβάλλον ανάπτυξης SDSoC της Xilinx προσφέρει φιλική προς το χρήστη εμπειρία σχεδίασης ενσωματωμένων εφαρμογών C/C++/OpenCL για ετερογενή συστήματα Zynq SoC. Διαθέτοντας τον πρώτο compiler βελτιστοποίησης πλήρους συστήματος της βιομηχανίας επιτρέπει την αυτοματοποιημένη επιτάχυνση λογισμικού σε προγραμματιζόμενη λογική. [24]

2.4 Διαδικασία Επίτευξης Επιτάχυνσης

Στην παρούσα παράγραφο παρουσιάζονται εν σειρά τα βήματα που οδήγησαν στην υλοποίηση του νευρωνικού δικτύου με ικανότητα ανακατασκευής ημιτελών εικόνων από το MNIST dataset στο FPGA με τελικό στόχο την επιτάχυνση των υπολογισμών που πραγματοποιεί το μοντέλο. Οι δυσκολίες που παρουσιάστηκαν στη διαδικασία σχετίζονταν κυρίως με το μεγάλο πλήθος παραμέτρων που εν γένει ένα νευρωνικό δίκτυο διαθέτει και του περιορισμένου αριθμού πόρων που διαθέτει ένα FPGA. Γι' αυτούς τους λόγους, η διαχείριση των πόρων για τη σχεδίαση του τελικού κυκλώματος είναι απαραίτητο να γίνεται με προσοχή. Παράλληλα, η προσπάθεια για σμίκρυνση του αρχικού μοντέλου με στόχο τον περιορισμό των παραμέτρων του είναι απαραίτητη ειδικά στην περίπτωση της παρούσας εργασίας όπου γίνεται χρήση μίας απλής εκπαιδευτικής πλατφόρμας, η οποία διαθέτει ένα μικρό FPGA με πολύ περιορισμένο πλήθος υλικών πόρων. Όπως αναφέρεται και στην παράγραφο 2.3.1, το ZYBO Zynq-7000 ARM/FPGA SoC Trainer Board, διαθέτει το μικρότερο FPGA της οικογένειας Xilinx Zynq-7000 και γι' αυτό το λόγο, η προσπάθεια επιτάχυνσης ενός νευρωνικού δικτύου αποτελεί σημαντική πρόκληση.
Για την επιτάχυνση, οφείλει να δημιουργηθεί μία συνάρτηση σε γλώσσα C++ στην οποία θα εμπεριέχονται κατάλληλες εντολές προς τον compiler, οι οποίες θα τον καθοδηγούν σε ένα τελικό design κυκλώματος που θα υλοποιεί τους υπολογισμούς του νευρωνικού δικτύου. Μία τέτοια συνάρτηση καλείται hardware function. Οι hardware functions οφείλουν να ακολουθούν μερικές κατευθυντήριες γραμμές οι οποίες παρουσιάζονται παρακάτω: [26]

- Πολλαπλές hardware functions μπορούν να εκτελεστούν ταυτόχρονα υπό τον έλεγχο ενός master thread. Πολλαπλά master threads υποστηρίζονται.
- Μία top-level hardware function οφείλει να είναι global function και όχι κάποια μέθοδος κλάσης ή μία overloaded function.
- Δεν είναι δυνατό η διαχείριση εξαιρέσεων (exception handling) εντός μίας hardware function.
- Η αναφορά σε μία global μεταβλητή εντός μίας hardware function αποτελεί λάθος όταν η μεταβλητή αυτή χρησιμοποιείται σε κάποια άλλη software function.
- Μία hardware function πρέπει να έχει τουλάχιστον ένα όρισμα.

2.4.1 Συρρίκνωση Αρχικού Νευρωνικού Δικτύου

Στο κεφάλαιο 1.7 παρουσιάστηκε το τελικό γεννητικό νευρωνικό δίκτυο το οποίο μετά την από κοινού εκπαίδευση του με το «αντίπαλο» διακριτικό μοντέλο, απέκτησε την ικανότητα να παράγει στην έξοδο του το απόν κάτω μισό τμήμα των ημιτελών εικόνων από το MNIST dataset που τροφοδοτούνταν στην είσοδο του. Το μοντέλο αυτό διαθέτει 3 κρυφά επίπεδα (hidden layers) νευρώνων, το επίπεδο νευρώνων εξόδου και συνολικά 1159304 παραμέτρους. Για την υλοποίηση του νευρωνικού δικτύου στην προγραμματιζόμενη λογική του FPGA κάθε μία από τις παραμέτρους του μοντέλου οφείλει να αποθηκευτεί στις Block RAMs. Όπως παρουσιάστηκε και στην παράγραφο 2.3.3, το FPGA Z-7010 που διαθέτει η πλατφόρμα ZYBO διαθέτει 60 block RAMs με συνολική χωρητικότητα 2.1MegaBit. Υποθέτοντας ότι χρησιμοποιούνται 8 ή 16 ή 32 bit για την αναπαράσταση και αποθήκευση των παραμέτρων στις block RAMs του Z-7010, προφανώς με τις αντίστοιχες απώλειες ακρίβειας για μικρότερο πλήθος bits, λαμβάνεται ο εξής πίνακας:

Number of bits	8	16	32
Total Memory (Mbits)	9.27	18.54	37.08
Utilization	441.49%	882.98%	1765.96%

Είναι προφανές ότι η χρήση του αρχικού set παραμέτρων για την υλοποίηση του νευρωνικού δικτύου στο FPGA είναι απαγορευτική λόγω του τεράστιου μεγέθους του αρχικού μοντέλου. Ταυτόχρονα, πέρα από τα προβλήματα περιορισμένης διαθέσιμης μνήμης, τίθενται και θέματα περιορισμένου αριθμού LUTs και Flip-Flops τα οποία αναγκάζουν τον περιορισμό της τελικής επιτάχυνσης που θα προσφέρει το τελικό κύκλωμα. Γι' αυτούς τους

λόγους, γίνεται αναζήτηση ενός μικρότερου νευρωνικού δικτύου, το οποίο με μικρότερο πλήθος παραμέτρων να είναι ικανό να πετύχει ικανοποιητικά ποιοτική ανακατασκευή των εικόνων.
Το νέο γεννητικό μοντέλο που εκπαιδεύτηκε αποτελείται από δύο κρυφά επίπεδα νευρώνων και το επίπεδο νευρώνων εξόδου ενώ ο «ανταγωνιστής» του με το διακριτικό ρόλο παραμένει ίδιος. Επιπλέον, για την περαιτέρω μείωση του αριθμού των παραμέτρων αλλά και για την επιτάχυνση της διαδικασίας του υπολογισμού, παραλείπεται από κάθε νευρώνα η χρήση του κατωφλίου. Έτσι, κάθε νευρώνας διαθέτει μόνο ένα set βαρών, ένα για κάθε σύναψη του, και το κατώφλι b_k θεωρείται πάντα μηδέν χωρίς να υπάρχει ανάγκη για τον υπολογισμό μίας επιπλέον άθροισης στο τελικό αποτέλεσμα. Η δημιουργία του γεννητικού μοντέλου γίνεται με το εξής τμήμα κώδικα:

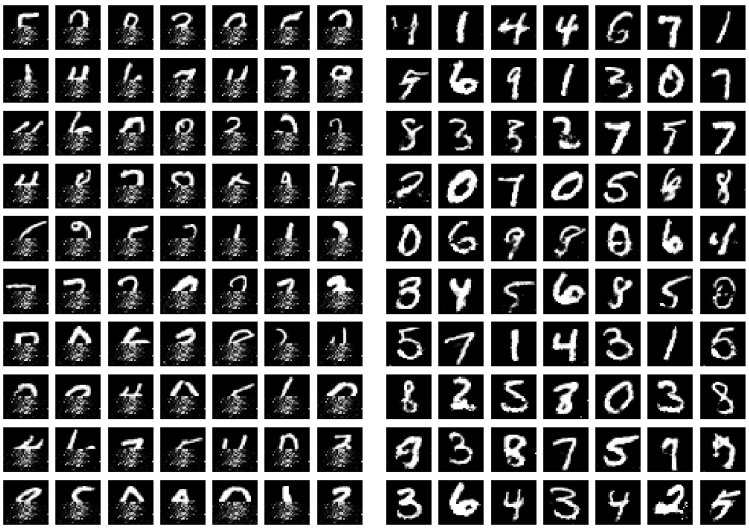
```
1  #    Generative Network
2
3  minmax = tensorflow.keras.constraints.MinMaxNorm(min_value=-2, max_value=2, rate=1.0, axis
       =0)
4
5  generator = Sequential()
6  generator.add(Dense(30, input_dim = 392, use_bias=False, kernel_initializer = initializers.
       RandomNormal(stddev=0.4), kernel_constraint = minmax))
7  generator.add(ReLU())
8  generator.add(Dense(50, use_bias = False, kernel_constraint = minmax))
9  generator.add(ReLU())
10 generator.add(Dense(392, use_bias = False, activation='tanh', kernel_constraint = minmax))
11
12 generator.compile(loss='binary_crossentropy', optimizer= optimizer)
13
```

Listing 2.1: Reduced Neural Network

Ενδιαφέρον στον παραπάνω κώδικα εγείρει η ύπαρξη του constraint στη γραμμή 3, η χρησιμότητα του οποίου είναι μεγάλη και σχολιάζεται στην παράγραφο 2.4.2. Στον παρακάτω πίνακα παρουσιάζονται οι λεπτομέρειες του δικτύου συνοπτικά.

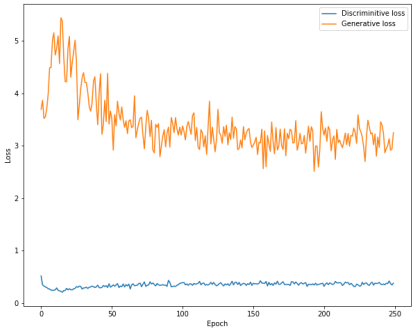
Generator Model				
Layer	**Neurons**	**Inputs**	**Activation Function**	**Trainable Parameters**
Hidden Dense Layer 1	30	392	ReLU	392*30 = 11790
Hidden Dense Layer 2	50	30	ReLU	30*50 = 1500
Output Layer	392	50	Tanh	50*392 = 19600
				Total : 32860

Η διαδικασία εκπαίδευσης παραμένει ίδια με αυτή των αντίστοιχων παραγράφων στο πρώτο μέρος. Όπως είναι εμφανές και από τον πίνακα, το πλήθος των παραμέτρων του νέου μοντέλου είναι κατά πολύ μικρότερο από αυτό του αρχικού δικτύου. Οι παράμετροι πλέον, όχι μόνο είναι αρκετές ώστε να είναι εφικτή η αποθήκευση τους στις Block RAMs του FPGA, αλλά μπορούν επιλεκτικά να τοποθετηθούν και σε διαφορετικά Blocks επιτρέποντας τεράστια παραλληλία συγκεκριμένων υπολογισμών. Περισσότερες λεπτομέρειες σχετικά με την εκμετάλλευση των Block RAMs παρουσιάζονται στην παράγραφο 2.4.4. Στο σχήμα 2.6 παρουσιάζεται η βελτίωση του μοντέλου κατά την πάροδο της εκπαίδευσης του με τη χρήση μίας συλλογής 70 φωτογραφιών για δύο διαφορετικές εποχές.



Σχήμα 2.6: Παραγόμενες από τον Generator εικόνες ύστερα από την πρώτη και την διακοσιοστή εποχή εκπαίδευσης

Είναι προφανές ότι μία τόσο μεγάλη μείωση των παραμέτρων του μοντέλου έχεις ως επακόλουθο την μείωση της ποιότητας των ανακατασκευασμένων εικόνων. Ωστόσο το trade-off μεταξύ ποιότητας εικόνων και συρρίκνωσης του μοντέλου είναι απαραίτητο εξαιτίας των περιορισμών που επιβάλλονται από το chip Z-7010. Στο σχήμα 2.7, παρουσιάζονται οι απώλειες των μοντέλων κατά τη διάρκεια της εκπαίδευσης τους και γίνεται ξεκάθαρη η αδυναμία του γεννητικού μοντέλου να ανταγωνιστεί το κατά πολύ μεγαλύτερο διακριτικό δίκτυο.

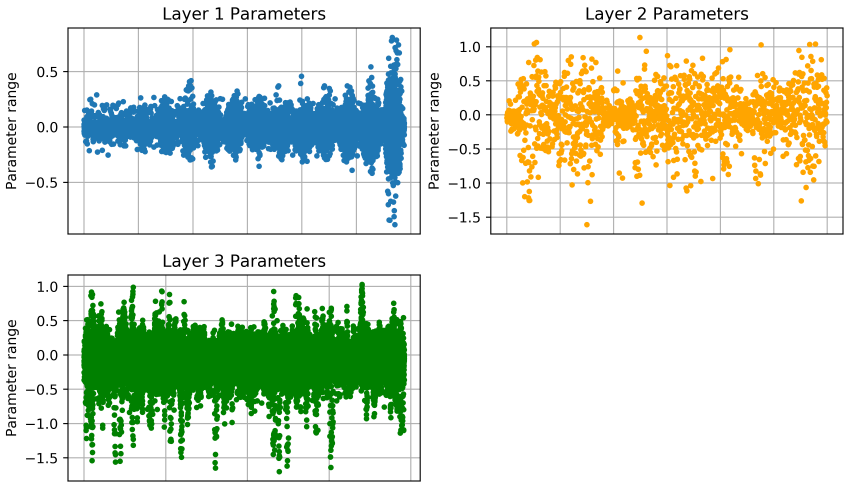


Σχήμα 2.7: Απώλειες μοντέλων για τις διάφορες εποχές εκπαίδευσης.

Το γεννητικό μοντέλο εν τέλει αποθηκεύεται και χρησιμοποιείται στην επόμενη παράγραφο για μελέτη των παραμέτρων του και τη μετέπειτα διαχείρισή τους για να επιτευχθεί η επιτάχυνση.

2.4.2 Ανάλυση Παραμέτρων Μοντέλου

Το παρόν βήμα κατέχει κρίσιμη σημασία για την μετέπειτα λήψη σχεδιαστικών αποφάσεων που αφορούν την αναπαράσταση των παραμέτρων των νευρώνων στην προγραμματιζόμενη λογική. Συγκεκριμένα, γίνεται μελέτη του εύρους των τιμών που λαμβάνουν οι παράμετροι όλων των νευρώνων καθώς και τα εύρη των τιμών των ενδιάμεσων αποτελεσμάτων u_j των επιμέρους επιπέδων νευρώνων, πριν εφαρμοστεί σε αυτά κάποια συνάρτηση ενεργοποίησης.
Αρχικά, στα διαγράμματα του σχήματος 2.8 παρουσιάζονται οι τιμές που λαμβάνουν τα βάρη των συνάψεων των νευρώνων του εκάστοτε επιπέδου.



Σχήμα 2.8: Τιμές παραμέτρων νευρώνων για τα διάφορα επίπεδα.

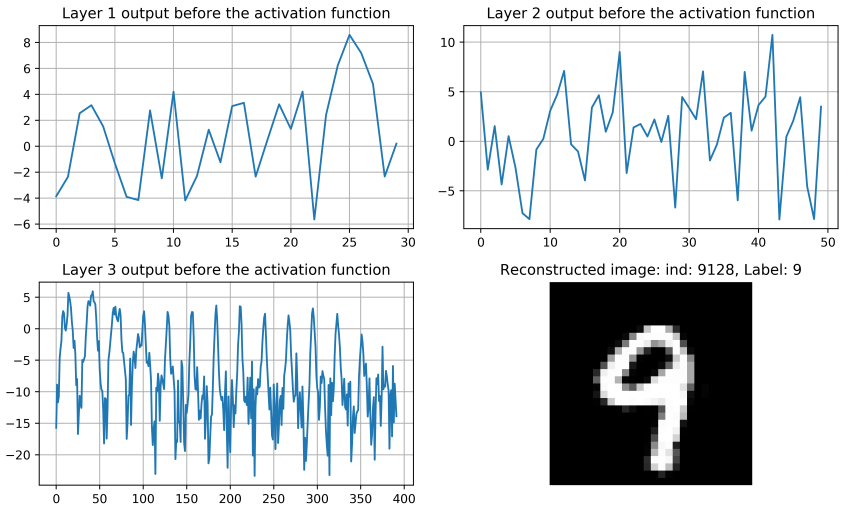
Όπως γίνεται αντιληπτό, όλες οι παράμετροι βρίσκονται εντός του εύρους των ακέραιων αριθμών -2 και 2. Η εκπαίδευση του μοντέλου έχει οδηγήσει σε αυτό το εύρος τιμών των παραμέτρων όχι τυχαία. Όπως παρουσιάστηκε και στην παραπάνω παράγραφο, έγινε χρήση του MinMaxNorm constraint που παρέχεται από τη βιβλιοθήκη του TensorFlow. Η επιβολή αυτού του constraint στα dense layers του μοντέλου, με παραμέτρους min_value = -2 και max_value = 2, είναι αυτή που οδηγεί τις παραμέτρους του μοντέλου σε αυτά τα εύρη τιμών. Η χρησιμότητα του constraint είναι μεγάλη καθώς πλέον μπορεί να γίνει κατάλληλη επιλογή πλήθους bits για την αναπαράσταση των παραμέτρων εντός του FPGA ώστε να εξοικονομούνται υλικοί πόροι και ταυτόχρονα να αποφευχθούν φαινόμενα υπερχείλισης. Περισσότερες λεπτομέρειες σχετικά με τον τρόπο αναπαράστασης παρουσιάζονται στην αμέσως επόμενη παράγραφο.

Πέρα από την ανάλυση των παραμέτρων του μοντέλου, σημαντική σημασία κατέχει και η ανάλυση των ενδιάμεσων αποτελεσμάτων που παράγονται από τα νευρωνικά επίπεδα, καθώς είναι απαραίτητος ο υπολογισμός τους με χρήση όσο το δυνατόν λιγότερου υλικού με την αποφυγή φαινομένων υπερχειλίσεων όπου θα οδηγούσαν στην αλλοίωση των αποτελεσμάτων. Στο σχήμα 2.9 παρουσιάζονται οι τιμές που λαμβάνουν οι έξοδοι των νευρώνων πριν την εφαρμογή της συνάρτησης ενεργοποίησης, για ένα τυχαίο παράδειγμα του dataset, καθώς και η τελική ανακατασκευασμένη εικόνα, ενώ στον αμέσως επόμενο πίνακα παρουσιάζονται μέγιστες και ελάχιστες τιμές που εμφανίζονται στις εξόδους των νευρωνικών επιπέδων πάνω σε όλο το training set του dataset.

	Maximum Value	**Minimum Value**
Layer 1	17.496	-18.743
Layer 2	25.197	-27.721
Layer 3	19.786	-69.221

Πίνακας 2.1: Μέγιστες και ελάχιστες τιμές στις εξόδους των νευρωνικών επιπέδων πάνω σε όλο το training set.

Γίνεται η υπόθεση, ότι οι μέγιστες και ελάχιστες τιμές που δύνανται να εμφανιστούν στα νευρωνικά επίπεδα είναι ίδιες ακόμα και για τα παραδείγματα του test set, εφόσον όλα τα δεδομένα ακολουθούν την ίδια κατανομή, και επομένως οι επιλογές που γίνονται στην παράγραφο 2.4.3 βασίζονται στα παραπάνω αποτελέσματα.



Σχήμα 2.9: Έξοδοι νευρωνικών επιπέδων και τελική ανακατασευασμένη εικόνα για τυχαίο δείγμα του training set.

2.4.3 Επιλογή Κατάλληλης Αναπαράστασης Παραμέτρων στην Προγραμματιζόμενη Λογική

Στην περίπτωση συγγραφής κώδικα κλασσικά σε γλώσσα C για την υλοποίηση των υπολογισμών του νευρωνικού δικτύου, ο οποίος κώδικας θα εκτελείτο σε CPU, τα βάρη καθώς και όλα τα αποτελέσματα των νευρώνων θα δηλώνονταν ως μεταβλητές τύπου float. Οι μεταβλητές τύπου float (κινητής υποδιαστολής) είναι απαραίτητες για την αναπαράσταση δεκαδικών αριθμών. Η χρήση τους προσφέρει μεγάλη ακρίβεια στους υπολογισμούς ωστόσο οι πράξεις μεταξύ αριθμών κινητής υποδιαστολής είναι και υπολογιστικά χρονοβόρες αλλά και απαιτούν κατάλληλο και σχετικά περίπλοκο hardware.
Στο FPGA, το τεράστιο πλήθος βαρών του νευρωνικού δικτύου αλλά και το περίπλοκο hardware που απαιτείται για την υλοποίηση πράξεων μεταξύ αριθμών κινητής υποδιαστολής σε συνδυασμό με τους περιορισμένους υλικούς πόρους που είναι διαθέσιμοι, καθιστούν την χρήση αριθμών κινητής υποδιαστολής για την αναπαράσταση των παραμέτρων και των αποτελεσμάτων απαγορευτική. Ωστόσο, η Xilinx προσφέρει τη δυνατότητα χρήσης αριθμών αυθαίρετης ακρίβειας, είτε ακεραίων είτε δεκαδικών μη κινητής υποδιαστολής, μέσω των βιβλιοθηκών ap_int & ap_fixed στις οποίες εμπεριέχονται οι template classes ap_[u]int και ap_[u]fixed. [25]

Η ap_[u]int template class

Η συγκεκριμένη κλάση καθιστά δυνατή την υλοποίηση ακέραιων μεταβλητών αυθαίρετης ακρίβειας στο hardware. Προσφέρει όλους τους arithmetic, bitwise, logical και relational operators που υπάρχουν και στις τυπικές μεταβλητές της γλώσσας C. Για την χρήση της κλάσης ap_[u]int αρκεί η δήλωση μίας μεταβλητής ως εξής:

ap_[u]int<int W> var

Στην περίπτωση προσημασμένων ακεραίων (ap_int), η μεταβλητή var μπορεί να λάβει τιμές από -2^{W-1} έως $2^{W-1}-1$ ενώ για την περίπτωση μη προσημασμένων (ap_uint), από 0 έως $2^{W}-1$.

Η ap_[u]fixed template class

Η συγκεκριμένη κλάση καθιστά δυνατή την υλοποίηση μεταβλητών σταθερής υποδιαστολής ,για την αναπαράσταση δεκαδικών αριθμών, στο hardware. Όπως και παραπάνω, προσφέρει όλους τους arithmetic, bitwise, logical και relational operators που υπάρχουν και στις τυπικές μεταβλητές της γλώσσας C. Για την χρήση της κλάσης ap_[u]fixed αρκεί η δήλωση μίας μεταβλητής ως εξής:

ap_[u]fixed<int W, int I, ap_q_mode Q, ap_o_mode O, ap_sat_bits N> var

Η τιμή του W καθορίζει το πλήθος bits που χρησιμοποιούνται για την αναπαράσταση όλης της μεταβλητής, ενώ η τιμή του I καθορίζει το πλήθος των bits που χρησιμοποιούνται για την αναπαράσταση του ακέραιου μέρους της. Στην περίπτωση προσημασμένων δεκαδικών αριθμών σταθερής υποδιαστολής (ap_fixed), η μεταβλητή var μπορεί να λάβει τιμές από -2^{I-1} έως $(2^{W-1}-1)2^{-(W-I)}$, ενώ στην περίπτωση των μη προσημασμένων δεκαδικών αριθμών σταθερής υποδιαστολής (ap_ufixed) η μεταβλητή var μπορεί να λάβει τιμές από 0 έως $(2^{W}-1)2^{-(W-I)}$. Ο περιορισμένος αριθμός bit για την αναπαράσταση του δεκαδικού μέρους του αριθμού επιβάλει στη μεταβλητή να παίρνει διακριτές κβαντισμένες τιμές. Το βήμα του κβαντισμού, δηλαδή το μέγεθος του κβάντου είναι σε κάθε περίπτωση $2^{-(W-I)}$. Επομένως, διαδοχικές τιμές που μπορεί να λάβει η μεταβλητή var διαφέρουν κατά $2^{-(W-I)}$.
Η παράμετρος Q καθορίζει τον τρόπο με τον οποίο γίνεται κβαντισμός κατά την ανάθεση μίας τιμής στη μεταβλητή. Η παράμετρος αυτή μπορεί να λάβει συγκεκριμένες προκαθορισμένες τιμές ενώ η default τιμή της είναι AP_TRN. Μερικές από τις δυνατές τιμές με τους τρόπους κβαντισμού που επιβάλλουν είναι οι εξής: [25]

- AP_RND : Στρογγυλοποίηση προς την πλησιέστερη έγκυρη τιμή.

```
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.5
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0

```

Listing 2.2: AP_RND quantization mode

- AP_RND_ZERO : Στρογγυλοποίηση προς την πλησιέστερη έγκυρη τιμή προς το μηδέν.

```
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0

```

Listing 2.3: AP_RND_ZERO quantization mode

- AP_RND_MIN_INF : Στρογγυλοποίηση προς την πλησιέστερη έγκυρη τιμή προς το πλην άπειρο.

```
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5

```

Listing 2.4: AP_RND_MIN_INF quantization mode

- AP_RND_INF : Στρογγυλοποίηση προς το άπειρο.

```
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.5
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5

```

Listing 2.5: AP_RND_INF quantization mode

Η παράμετρος O καθορίζει τη συμπεριφορά στην περίπτωση υπερχείλισης. Λαμβάνει και αυτή προκαθορισμένες τιμές με default τιμή την AP_WRAP. Όλες οι δυνατές τιμές και η συμπεριφορά που επιβάλλουν είναι οι εξής: [25]

- AP_SAT : Η τιμή παραμένει στο θετικό κορεσμό για θετικές τιμές ή στον αρνητικό για αρνητικές τιμές.

```
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 7.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: -8.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: 0.0

```

Listing 2.6: AP_SAT overflow mode

- AP_SAT_ZERO : Η τιμή τίθεται στο μηδέν.

```
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields: 0.0

```

Listing 2.7: AP_SAT_ZERO overflow mode

- AP_SAT_SYM : Η τιμή παραμένει στο θετικό κορεσμό για θετικές τιμές και στην αρνητική τιμή του θετικού κορεσμού για αρνητικές τιμές.

```
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields: 7.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields: -7.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields: 0.0

```

Listing 2.8: AP_SAT_SYM overflow mode

- AP_WRAP : Η τελική τιμή προκύπτει από κυκλική τύλιξη της αρχικής τιμής.

```
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0; // Yields: -1.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: -3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0; // Yields: 3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: 13.0

```

Listing 2.9: AP_WRAP overflow mode

- AP_WRAP_SYM : Η τελική τιμή προκύπτει από προσημασμένη κυκλική τύλιξη της αρχικής τιμής.

```
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = 19.0; // Yields: -4.0
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = -19.0; // Yields: 2.0

```

Listing 2.10: AP_WRAP_SYM overflow mode

Η παράμετρος N σχετίζεται με τη λειτουργία των AP_WRAP & AP_WRAP_SYM overflow modes και δε θα γίνει περαιτέρω χρήση αυτής αλλά και των αντίστοιχων overflow modes στην παρούσα εργασία. [25]

Επιλογή κατάλληλου τύπου δεδομένων για την αναπαράσταση παραμέτρων & αποτελεσμάτων

Έχοντας προηγηθεί μελέτη των παραμέτρων και των εξόδων των επιπέδων του νευρωνικού δικτύου οι οποίες προφανώς θα αποθηκεύονται σε κάποια μεταβλητή, είναι δυνατή η επιλογή του κατάλληλου τύπου για την αναπαράσταση των παραπάνω στην προγραμματιζόμενη λογική.
Εφόσον λόγω των constraints που επιβλήθηκαν κατά την εκπαίδευση του μοντέλου στα dense layers, τα βάρη λαμβάνουν τιμές από -2 έως και 2, όλες οι παράμετροι του μοντέλου θα αποθηκευτούν ως ap_fixed αριθμοί με 2 bits για το ακέραιο μέρους τους. Επιπλέον, είναι επιθυμητή η κβάντιση των τιμών των παραμέτρων με τρόπο ώστε η τελική τιμή τους μετά την ανάθεση να είναι όσο το δυνατόν πλησιέστερα στην πραγματική τιμή, επομένως η απλή λειτουργία AP_RND αρκεί ως quantization mode. Επίσης, εφόσον οι τιμές των παραμέτρων μένουν αμετάβλητες καθ' όλη τη διάρκεια των υπολογισμών δεν υπάρχει το ενδεχόμενο υπερχείλισης και έτσι το overflow mode αφήνεται στη default λειτουργία του. Τέλος, για το πλήθος των bits με τα οποία θα γίνεται αναπαράσταση του δεκαδικού μέρους έγιναν αρκετές δοκιμές στις οποίες υπήρχε trade-off μεταξύ ακρίβειας και σπατάλης υλικών πόρων. Η τελική επιλογή αφορά 10 bits για το δεκαδικό μέρος των αριθμών, ωστόσο στην παράγραφο 2.5.2 γίνεται παρουσίαση του σφάλματος της εξόδου για διάφορες τιμές του πλήθους bits για την αναπαράσταση του δεκαδικού μέρους.
Σχετικά με τα αποτελέσματα που εγείρονται στις εξόδους των νευρωνικών επιπέδων, όπως παρουσιάστηκε, η ελάχιστη και μέγιστη τιμή κυμαίνεται από -69.221 έως 25.197. Καθώς αυτοί οι αριθμοί αποτελούν τελικά αποτελέσματα επιμέρους αθροίσεων είναι βέβαιο ότι ενδιάμεσα αποτελέσματα πριν ολοκληρωθούν όλες οι αθροίσεις

θα εγείρουν μεγαλύτερες ή μικρότερες τιμές. Υπενθυμίζεται ότι η ποσότητα που περνά από τη συνάρτηση ενεργοποίησης ενός νευρώνα είναι:

$$u_k = \sum_{i=1}^{m} w_{ki} x_i + b_k$$

Γι' αυτό το λόγο, αλλά και για λόγους ασφάλειας για την περίπτωση που υπάρξει κάποιο παράδειγμα εισόδου το οποίο να μεγιστοποιήσει ή ελαχιστοποιήσει περεταίρω την τιμή κάποιας εξόδου νευρώνα, επιλέγονται για την αναπαράσταση των αποτελεσμάτων ap_fixed αριθμοί με 9 bits για την αναπαράσταση του ακέραιου μέρους τους. Ξανά, επιλέγεται η λειτουργία AP_RND ως quantization mode. Σχετικά με τη λειτουργία υπερχείλισης, επιθυμητή θα ήταν η λειτουργία AP_SAT καθώς θα περιόριζε το τελικό σφάλμα του νευρώνα στην περίπτωση που σε κάποια ενδιάμεση άθροιση συνέβαινε υπερχείλιση του αποτελέσματος. Ωστόσο, για την υλοποίηση της λειτουργίας AP_SAT απαιτούνται πολλοί υλικοί πόροι οι οποίοι δεν είναι διαθέσιμοι. Γι' αυτό το λόγο, το quantization mode αφήνεται στη default λειτουργία του και το πλήθος bits του ακέραιου μέρους επιλέγεται με τρόπο τέτοιο ώστε να προσφέρεται μία ασφάλεια για τυχόν υπερχείλιση. Το πλήθος bits που αφορά το δεκαδικό μέρος επιλέγεται τελικά να είναι ίσο με 10 προσφέροντας την ίδια ακρίβεια που προσφέρεται και στην περίπτωση των παραμέτρων του μοντέλου.
Όλα τα παραπάνω υλοποιούνται με την δημιουργία δύο νέων τύπων δεδομένων (quantized_type και l_quantized_type) με τη χρήση της κλάσης ap_fixed όπως ακριβώς παρουσιάζεται στο παρακάτω τμήμα κώδικα:

```
typedef ap_fixed<10,2,AP_RND> quantized_type;
typedef ap_fixed<17,9,AP_RND> l_quantized_type;

```

Listing 2.11: Custom datatypes

Datatype	Integer Part	Fraction Part	Signed	Minimum	Maximum	Quantum Size
quantized_type	2 bits	10 bits	Yes	-2	1.9990234375	0.0009765625
l_quantized_type	9 bits	10 bits	Yes	-256	255.9990234375	0.0009765625

Πίνακας 2.2: Λεπτομέρειες νεοορισθέντων τύπων δεδομένων

2.4.4 Δημιουργία Κατάλληλης Αρχιτεκτονικής Μνήμης στην Προγραμματιζόμενη Λογική

Όπως ήδη αναφέρθηκε, η συμπεριφορά κάθε νευρωνικού επιπέδου περιγράφεται ως πολλαπλασιασμός ενός διανύσματος εισόδου X με έναν δισδιάστατο πίνακα W διαστάσεων $m * n$, όπου n το πλήθος των νευρώνων του επιπέδου και m το πλήθος των εισόδων κάθε νευρώνα όπου ακολουθείται από πρόσθεση του προκύπτοντας διανύσματος με το διάνυσμα των κατωφλιών των επιμέρους νευρώνων. Κάθε στήλη του πίνακα W περιέχει τα βάρη του ενός νευρώνα. Στην περίπτωση της παρούσας εργασίας, εφόσον για ελαχιστοποίηση τους πλήθους των παραμέτρων οι νευρώνες έχουν πάντοτε κατώφλι ίσο με μηδέν όπως αναλύθηκε στην παράγραφο 2.4.1 , η τελική πρόσθεση με το διάνυσμα των κατωφλίων δεν πραγματοποιείται. Έτσι, για το παρόν νευρωνικό δίκτυο, οι υπολογισμοί που λαμβάνουν χώρα είναι με τη σειρά οι εξής:

$$Y_1 = ReLU(XW_1) \rightarrow Y_2 = ReLU(Y_1 W_2) \rightarrow Y_3 = Tanh(Y_2 W_3)$$

Τα βάρη τα οποία έχουν υπολογιστεί κατά τη διαδικασία της εκπαίδευσης, για να αποθηκευτούν στο FPGA αρκεί η δήλωση δισδιάστατων πινάκων στον κώδικα της High Level Synthesis και η αρχικοποίηση τους με τις τιμές των ίδιων των βαρών. Καθώς το μέγεθος των πινάκων είναι μεγάλο, η αρχικοποίηση αυτή γίνεται με τη χρήση του Python script του παραρτήματος Γ'1. Οι τελικοί πίνακες που εμπεριέχουν τα βάρη έχουν την εξής μορφή:

```
quantized_type W1[392][30] = {
{...},
 ...
{...}};
```

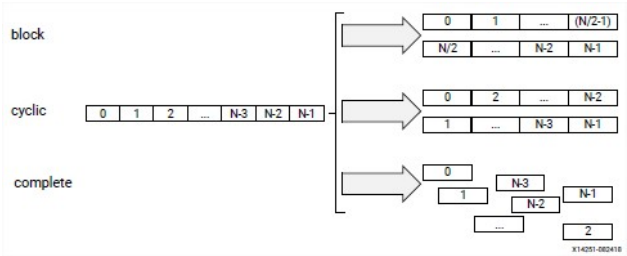
```
5
6  quantized_type W2[30][50] = {
7  {...},
8   ...
9  {...}};
10
11 quantized_type W3[50][392] = {
12 {...},
13  ...
14 {...}};
15
16
```

Listing 2.12: Neural network weights definitions

Ωστόσο, για την επίτευξη της επιτάχυνσης της διαδικασίας, δεν αρκεί απλά η τοποθέτηση των βαρών σε δισδιάστατους πίνακες για τη μετέπειτα ανάγνωση τους και τον πολλαπλασιασμό τους με τα αντίστοιχα διανύσματα. Προκειμένου οι πολλαπλασιασμοί με τα διανύσματα να γίνονται αποδοτικά είναι αναγκαίος ο παράλληλος υπολογισμός των επιμέρους γινομένων. Γι' αυτό το λόγο, είναι απαραίτητο το hardware που θα υλοποιήσει τους υπολογισμούς να δύναται να έχει ταυτόχρονη πρόσβαση σε επιμέρους στοιχεία των πινάκων.
Όπως αναφέρθηκε και στην παράγραφο 2.1.1, υπεύθυνο block για την αποθήκευση πληροφοριών στο εσωτερικό του FPGA, είναι οι block RAMs. Κάθε block RAM του chip Z-7010 που χρησιμοποιείται διαθέτει δύο ports για την εγγραφή και ανάγνωση πληροφορίας καθιστώντας δυνατή την ταυτόχρονη επίτευξη δύο προσβάσεων στο block. [26]. Για να είναι δυνατή η ταυτόχρονη πρόσβαση σε περισσότερες από δύο παραμέτρους, είναι αναγκαία η τοποθέτηση τους σε διαφορετική block RAM. Συγκεκριμένα, αν είναι επιθυμητή η ταυτόχρονη πρόσβαση σε k διαφορετικές παραμέτρους, είναι απαραίτητο αυτές να τοποθετηθούν σε $k/2$ διαφορετικές block RAM, εφόσον κάθε block επιτρέπει ταυτόχρονη πρόσβαση σε δύο.
Για την τοποθέτηση στοιχείων πινάκων σε block RAMs της επιλογής του σχεδιαστή, παρέχεται η οδηγία pragma HLS ARRAY_PARTITION η οποία μπορεί να χρησιμοποιηθεί ως οδηγία προς τον compiler να διαμερίσει κατάλληλα έναν πίνακα σε μικρότερους. Οι διαμερίσεις που επιτρέπονται είναι οι εξής:[26]

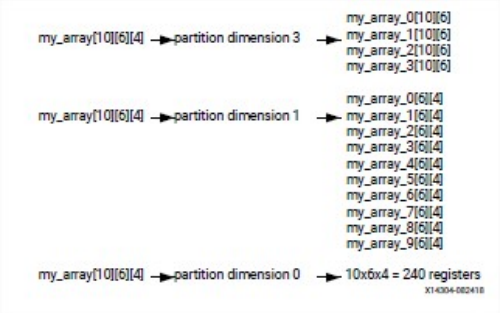
- Block : Ο αρχικός πίνακας διαιρείται σε blocks ίσου μεγέθους αποτελούμενα από διαδοχικά στοιχεία του αρχικού πίνακα.
- Cyclic : Ο αρχικός πίνακας διαιρείται σε blocks ίσου μεγέθους συμπλέκοντας τα στοιχεία του αρχικού πίνακα.
- Complete : Ο αρχικός πίνακας διαμερίζεται στα επιμέρους στοιχεία του τα οποία τοποθετούνται σε ξεχωριστούς καταχωρητές.

Για τα block και cyclic partitioning , η παράμετρος factor καθορίζει το πλήθος των μικρότερων πινάκων που δημιουργούνται. Στην περίπτωση που το πλήθος των στοιχείων των επιμέρους πινάκων δεν προκύπτει ακέραιο, οι τελευταίοι πίνακες της διαμέρισης θα διαθέτουν λιγότερα στοιχεία. Στο σχήμα 2.10 παρουσιάζεται ο τρόπος λειτουργίας των επιμέρους διαμερίσεων.



Σχήμα 2.10: Είδη διαμερίσεων με factor=2. [26]

Για την περίπτωση πινάκων με περισσότερες από μία διαστάση, η διάσταση στην οποία θα συμβεί η διαμέριση επιλέγεται μέσω της παραμέτρου dim [26]. Παράδειγμα διαμέρισης πολυδιάστατου πίνακα παρουσιάζεται στο σχήμα 2.11.



Σχήμα 2.11: Διαμέριση πολυδιάστατου πίνακα. [26]

Βάση των παραπάνω, για τους πίνακες που εμπεριέχουν τις παραμέτρους των τριών νευρωνικών επιπέδων του δικτύου της παρούσας εφαρμογής, γίνονται οι εξής σχεδιαστικές επιλογές όσον αφορά την αρχιτεκτονική της μνήμης, ενώ η υλοποίηση των πολλαπλασιασμών των πινάκων παρουσιάζεται στην παράγραφο 2.4.6.

- Hidden Layer 1

  Κάθε νευρώνας του πρώτου νευρωνικού επιπέδου διαθέτει 392 εισόδους ενώ συνολικά υπάρχουν 30 διαφορετικοί νευρώνες. Μία απόπειρα παράλληλου υπολογισμού και των 392 διαφορετικών γινομένων ενός νευρώνα θα ήταν αδύνατη καθώς στη διάθεση του σχεδιαστή το συγκεκριμένο chip προσφέρει 60 διαφορετικές block RAMs. Γι' αυτό το λόγο, στο πρώτο νευρωνικό επίπεδο η παραλληλοποίηση θα υπάρξει στο επίπεδο των νευρώνων και όχι στο επίπεδο των εισόδων ενός νευρώνα. Αυτό σημαίνει πως η διαμέριση θα γίνει με τέτοιο τρόπο ώστε να είναι δυνατός ο ταυτόχρονος υπολογισμός της m-οστής εισόδου με το m-οστό συναπτικό βάρος όλων των νευρώνων του επιπέδου. Πέρα από τον ήδη υπάρχων πίνακα βαρών, οφείλει να δημιουργηθεί και ένας πίνακας για τα αποτελέσματα των νευρώνων, κατάλληλα διαμερισμένος για την επίτευξη παραλληλίας στο επόμενο επίπεδο.

- Hidden Layer 2

  Κάθε νευρώνας του δεύτερου νευρωνικού επιπέδου διαθέτει 30 εισόδους ενώ συνολικά υπάρχουν 50 διαφορετικοί νευρώνες. Αυτή τη φορά είναι δυνατός ο παράλληλος υπολογισμός όλων των γινομένων εισόδων – συναπτικών βαρών ενός νευρώνα και έτσι η παραλληλοποίηση έγκειται στο επίπεδο των εισόδων ενός νευρώνα. Έτσι, τα συναπτικά βάρη ενός νευρώνα θα αποθηκευτούν σε 15 διαφορετικές block RAMs ενώ θα δημιουργηθεί ξανά πίνακας για τα αποτελέσματα των νευρώνων με την κατάλληλη διαμέριση για επίτευξη παραλληλίας στο επόμενο επίπεδο.

- Output Layer 3

  Κάθε νευρώνας του τρίτου και τελευταίου νευρωνικού επιπέδου διαθέτει 50 εισόδους ενώ συνολικά υπάρχουν 392 διαφορετικοί νευρώνες. Ξανά, η παραλληοποίηση θα γίνει στο επίπεδο των εισόδων κάθε νευρώνα με ταυτόχρονο υπολογισμό όλων των γινομένων εισόδων - συναπτικών βαρών. Έτσι, τα συναπτικά βάρη ενός νευρώνα θα αποθηκευτούν σε 25 διαφορετικές block RAMs με τα τελικά αποτελέσματα να επιστρέφουν στον επεξεργαστή όπως θα παρουσιαστεί στην παράγραφο 2.4.8

Όλα τα παραπάνω, περιγράφονται στον κώδικα της High-level Synthesis ως εξής:

```
1 l_quantized_type layer_1_out[30];
2 l_quantized_type layer_2_out[50];
3
4 #pragma HLS array_partition variable=layer_1_out block factor=15 dim=1
5 #pragma HLS array_partition variable=layer_2_out block factor=25 dim=1
6 #pragma HLS array_partition variable=W1 block factor=15 dim=2
7 #pragma HLS array_partition variable=W2 block factor=15 dim=1
8 #pragma HLS array_partition variable=W3 block factor=25 dim=1
9
```

Listing 2.13: Memory architecture

Σημειώνεται ότι για τα αποτελέσματα των κρυφών επιπέδων χρησιμοποιήθηκε ο τύπος l_quantized_type για τους λόγους που εξηγήθηκαν στην παράγραφο 2.4.3.

2.4.5 Μεταφορά Δεδομένων Μεταξύ Επεξεργαστή & FPGA

Μέσω του κώδικα High-level Synthesis, γίνεται περιγραφή σε υψηλό επίπεδο του υλικού το οποίο θα υλοποιεί τους υπολογισμούς του νευρωνικού δικτύου. Η περιγραφή αυτή έχει τη μορφή μίας κλασσικής συνάρτησης στη γλώσσα C++ με επιπρόσθετες εντολές για τον compiler όπου εν τέλει υλοποιείται στην προγραμματιζόμενη λογική του FPGA (hardware function). Η συνάρτηση αυτή περιέχει δύο ορίσματα, τα οποία αφορούν τα δεδομένα εισόδου και εξόδου. Τα δεδομένα εισόδου της hardware function είναι τα δεδομένα που εισέρχονται στο πρώτο επίπεδο νευρώνων του νευρωνικού δικτύου και αποτελούνται από τις τιμές των επιμέρους pixels της ημιτελούς εικόνας προς ανακατασκευή. Αντίστοιχα, τα δεδομένα εξόδου της hardware function είναι τα δεδομένα που παράγονται στο επίπεδο εξόδου του νευρωνικού δικτύου, και αποτελούνται από τις τιμές των επιμέρους pixels του απόντος κάτω τμήματος της αρχικής εικόνας που αναπαράγει το μοντέλο.
Στην παρούσα εφαρμογή, το dataset των ημιτελών εικόνων βρίσκεται αποθηκευμένο υπό τη μορφή αρχείου csv και σε αυτό έχει πρόσβαση ο επεξεργαστής μέσω των κλασσικών εντολών της C++ για άνοιγμα, ανάγνωση και κλείσιμο αρχείων. Για το parsing του csv file έχει δημιουργηθεί κατάλληλη συνάρτηση η οποία παρουσιάζεται όπως και ο συνολικός κώδικας στο παράρτημα Β΄4. Μετά το parsing του αρχείου, το σύνολο των δεδομένων βρίσκεται αποθηκευμένο σε πίνακα και διαθέσιμο για περεταίρω επεξεργασία από τη CPU. Ωστόσο, είναι επιθυμητή η μεταφορά αυτών των δεδομένων στην προγραμματιζόμενη λογική για την πραγματοποίηση των υπολογισμών σε αυτή με τελικό στόχο την επιτάχυνση τους. Επομένως, είναι επιθυμητή η επικοινωνία επεξεργαστή και FPGA για τη μεταφορά των δεδομένων.
Η επικοινωνία αυτή υλοποιείται μέσω του πρωτόκολλου AXI. Ωστόσο, χάρη στην υψηλού επιπέδου αφαιρετικότητα της περιγραφής High-level Synthesis, ο χρήστης δεν χρειάζεται να εμπλακεί με λεπτομέρειες χαμηλού επιπέδου και η μεταφορά των δεδομένων μπορεί να επιτευχθεί εύκολα, ξανά με τη βοήθεια συγκεκριμένων εντολών προς τον compiler. Επιπλέον, για την όσο το δυνατόν γρηγορότερη μεταφορά των δεδομένων, τα δεδομένα του dataset είναι επιθυμητό να αποθηκευτούν σε φυσικά συνεχείς περιοχές της μνήμης. Αυτό γίνεται εφικτό με τη δέσμευση μνήμης για τους πίνακές που εμπεριέχουν τα δεδομένα του dataset αλλά και των αποτελεσμάτων μέσω της εντολής sds_alloc. [26]
Για τη μεταφορά των δεδομένων, δηλαδή 392 float αριθμών προς το FPGA και 392 float αποτελεσμάτων από το FPGA, οφείλει η hardware function που υλοποιείται να διαθέτει δύο ορίσματα, ένα για τα δεδομένα εξόδου, και ένα για τα δεδομένα εισόδου. Η προκαθορισμένη λειτουργία μιας hardware function για τη μεταφορά δεδομένων, είναι να γίνεται αντιγραφή τους, με αποτέλεσμα, ένα όρισμα που αφορά κάποιον πίνακα να μπορεί να χρησιμοποιηθεί είτε ως είσοδος είτε ως έξοδος δεδομένων. Η συμπεριφορά αυτή μπορεί να επιβληθεί με τη χρήση της εντολής pragma SDS data copy. Στην περίπτωση που είναι επιθυμητή η χρήση ενός ορίσματος που μπορεί να χρησιμοποιηθεί και ως είσοδος αλλά και ως έξοδος δεδομένων από το FPGA, πρέπει να χρησιμοποιηθεί η εντολή pragma SDS data zero_copy η οποία πληροφορεί τον compiler ότι ο πίνακας που αφορά το όρισμα της hardware function πρέπει να τοποθετηθεί σε κάποια κοινόχρηστη μνήμη και να μη γίνει αντιγραφή του. [26]
Για την υλοποίηση της μεταφοράς, ένας μεταφορέας δεδομένων δημιουργείται από τον compiler και είναι υπεύθυνος για τη μετακίνηση των δεδομένων από και προς το hardware. Γενικότερα, ένας μεταφορέας δεδομένων μπορεί να είναι μία FIFO ή μία direct memory access (DMA) διεπαφή μεταξύ του επεξεργαστή και της προγραμματιζόμενης λογικής. Η επιλογή του data mover μπορεί να γίνει από τον compiler αυτόματα βάση του όγκου των δεδομένων προς μεταφορά καθώς και το μοτίβο των προσβάσεων που αναμένονται από το hardware που καταναλώνει ή παράγει τα δεδομένα, ενώ η υλοποίηση της γίνεται από τον compiler στην προγραμματιζόμενη λογική. [26]
Για να προσδιοριστεί στον compiler ο τρόπος με τον οποίο γίνεται πρόσβαση στα δεδομένα εισόδου και εξόδου, χρησιμοποιείται η εντολή pragma DATA ACCESS_PATTERN. Συγκεκριμένα, για κάθε όρισμα πίνακα δίνονται οι επιλογές για SEQUENTIAL και RANDOM πρόσβαση. Στην περίπτωση της SEQUENTIAL πρόσβασης στα δεδομένα, δημιουργείται μία διεπαφή ροής ενώ για την περίπτωση της RANDOM πρόσβασης, δημιουργείται μία διεπαφή RAM. Θεμελιώδης διαφορά αποτελεί το γεγονός ότι με την SEQUENTIAL πρόσβαση, είναι δυνατή η πρόσβαση στα στοιχεία του πίνακα που μεταφέρεται μία φορά ενώ με τη RANDOM πρόσβαση είναι δυνατή η πρόσβαση οποιουδήποτε στοιχείου του πίνακα με οποιαδήποτε σειρά. Ωστόσο στην περίπτωση της τυχαίας πρόσβασης, ακόμα και αν δεν γίνει πρόσβαση σε όλο τον όγκο των δεδομένων, η μεταφορά του συνολικού όγκου θα πραγματοποιηθεί. [26]

Για την παρούσα εφαρμογή, γίνεται αντιγραφή των δεδομένων εισόδου/εξόδου ενώ προσδιορίζεται στον compiler ότι η πρόσβαση σε αυτά θα γίνει σειριακά. Επιπλέον, η επιλογή data mover δεν αφήνεται στην επιλογή του compiler και προσδιορίζεται μέσω της εντολής pragma SDS data data_mover. Έγινε πειραματισμός με τους διαφορετικούς μεταφορείς δεδομένων που προσφέρονται (AXIFIFO, AXIDMA_SG, AXIDMA_SIMPLE), με τελική επιλογή τον AXIDMA_SIMPLE καθώς προσέφερε μεγαλύτερη ταχύτητα μεταφοράς. Τα παραπάνω, καθώς και η δήλωση της hardware function και των ορισμάτων της παρουσιάζονται στο παρακάτω τμήμα κώδικα:

```
#pragma SDS data copy(x[0:392], y[0:392])
#pragma SDS data access_pattern(x:SEQUENTIAL, y:SEQUENTIAL)
#pragma SDS data data_mover(x:AXIDMA_SIMPLE, y:AXIDMA_SIMPLE)

void forward_propagation(float *x, float *y);

```

Listing 2.14: Data mover design choises

2.4.6 Υλοποίηση Συναρτήσεων Ενεργοποίησης

Όπως ήδη παρουσιάστηκε στη θεωρία των νευρωνικών δικτύων, κάθε νευρώνας πέρα από στάθμιση των εισόδων του με πολλαπλασιασμό τους με τα συναπτικά του βάρη, διοχετεύει το τελικό αποτέλεσμα της στάθμισης μέσα από μία συνάρτηση ενεργοποίησης. Για την επιτάχυνση των υπολογισμών που πραγματοποιεί το νευρωνικό δίκτυο είναι απαραίτητος ο σχεδιασμός hardware το οποίο θα υλοποιεί τη λειτουργία των συναρτήσεων ενεργοποίησης του μοντέλου. Το παρόν μοντέλο που έχει αναπτυχθεί χρησιμοποιεί ως συναρτήσεις ενεργοποίησης τις ReLU και Tanh. Παρακάτω, παρουσιάζεται η υλοποίηση των δύο αυτών συναρτήσεων στην προγραμματιζόμενη λογική.

Υλοποίηση συνάρτησης ReLU

Η συνάρτηση ReLU όντας απλή στη λειτουργία της μπορεί εύκολα να γίνει περιγραφή της σε High-level Synthesis. Απλούστατα, ορίζεται μία συνάρτηση η οποία πραγματοποιεί έναν έλεγχο. Αν η είσοδός της είναι αρνητική επιστρέφει την τιμή 0, διαφορετικά επιστρέφει την τιμή που δόθηκε στην είσοδό της. Εφόσον, όπως παρουσιάστηκε, τα αποτελέσματα των νευρώνων αποθηκεύονται σε πίνακες τύπου l_quantized_type, η είσοδος της συνάρτησης ReLU είναι τύπου l_quantized_type όπως και η τιμή η οποία επιστρέφει. Παρακάτω παρουσιάζεται ο κώδικας που περιγράφει την λειτουργία της.

```
l_quantized_type ReLu(l_quantized_type res)
{
  if (res < 0)
    return 0;

  return res;
}

```

Listing 2.15: ReLU implementation

Υλοποίηση συνάρτησης Tanh

Εφόσον τα δεδομένα εισόδου της συνάρτησης είναι τύπου l_quantized_type μπορούν να λάβουν κβαντισμένες τιμές ενώ το εύρος τους είναι συγκεκριμένο Θεωρείται ότι τιμές εισόδου μικρότερες του -2 και μεγαλύτερες του 1.99609375 δίνουν στην έξοδο τιμές πολύ κοντά στο -1 και στο +1 αντίστοιχα. Γι' αυτό το λόγο, όταν εντοπίζονται στην είσοδο τιμές εκτός αυτού του εύρους τιμών, η έξοδος της συνάρτησης τίθεται αυτόματα στο -1 ή στο +1. Όπως έχει ήδη αναλυθεί, το μέγεθος του κβάντου, δηλαδή η ελάχιστη απόσταση μεταξύ δύο διαδοχικών τιμών είναι 0.0009765625. Για κάθε μία από αυτές τις διακριτές τιμές x στο εύρος από -2 έως 1.99609375, υπολογίζεται και η τιμή της συνάρτησης $tanh(x)$. Τα αποτελέσματα αποθηκεύονται σε πίνακα 4096 θέσεων καθώς στο συγκεκριμένο εύρος κβαντισμένων τιμών υπάρχουν 4096 διαφορετικοί αριθμοί. Πλέον, όταν στην είσοδο της συνάρτησης εμφανίζεται αριθμός εντός αυτού του εύρους, η ακολουθία των bits του χρησιμοποιείται για τη δεικτοδότηση του πίνακα και η κατάλληλη αποθηκευμένη τιμή επιστρέφεται στην έξοδο. Παρακάτω παρουσιάζεται ο κώδικας που περιγράφει τη συγκεκριμένη λειτουργία. Σημειώνεται ότι η μέθοδος range() της κλάσης ap_fixed στην γραμμή 9 χρησιμοποιείται για την αντιγραφή της αλληλουχίας των bits μίας συγκεκριμένης μεταβλητής.

```
l_quantized_type tanh(l_quantized_type res)
{
  if (res >= 2)
    return 1;
  else if (res < -2)
    return -1;
  else
  {
    ap_int <12> i = res.range();       // Copy bit pattern to i
    return tanh_vals[2048 + i.to_int()];
  }
}

```

Listing 2.16: Tanh implementation

Για το τελικό design υλοποιούνται 30 διαφορετικές συναρτήσεις ReLU στο hardware εφόσον τα αποτελέσματα του πρώτου επιπέδου που αποτελείται από τριάντα διαφορετικούς νευρώνες παράγονται ταυτόχρονα και για τη μέγιστη δυνατή επιτάχυνση τα αποτελέσματα αυτά πρέπει να διοχετευτούν ταυτόχρονα στην είσοδο των συναρτήσεων ενεργοποίησης. Για την περίπτωση της συνάρτησης Tanh ωστόσο, μία ακριβώς υλοποίηση αρκεί εφόσον τα αποτελέσματα των νευρώνων του τελευταίου επιπέδου παράγονται σειριακά όπως θα παρουσιαστεί και στην παράγραφο 2.4.7. Για τον περιορισμό του πλήθους των συναρτήσεων που θα υλοποιήσει στο hardware ο compiler γίνεται χρήση της εντολής pragma HLS allocation, και συγκεκριμένα ζητείται η υλοποίηση ακριβώς τριάντα συναρτήσεων ReLU και μίας συνάρτησης Tanh [27]. Ο περιορισμός αυτός επιβάλλεται ως εξής:

```
#pragma HLS allocation instances=tanh limit=1 function
#pragma HLS allocation instances=ReLu limit=30 function

```

Listing 2.17: Restricting the number of functions

2.4.7 Δημιουργία Hardware για τον Αποδοτικό Πολλαπλασιασμό Πινάκων

Πλέον έχουν ορισθεί οι κατάλληλοι τύποι δεδομένων, έχει δημιουργηθεί κατάλληλη αρχιτεκτονική μνήμης για επίτευξη επιτάχυνσης και έχει γίνει περιγραφή κατάλληλου hardware για την υλοποίηση των συναρτήσεων ενεργοποίησης. Χρησιμοποιώντας όλα τα παραπάνω είναι δυνατή η περιγραφή του hardware που θα υλοποιεί όλους τους υπολογισμούς του νευρωνικού δικτύου της παρούσας εφαρμογής.
Στην παρούσα παράγραφο, παρουσιάζονται οι δύο διαφορετικοί τρόποι που χρησιμοποιήθηκαν για τον πολλαπλασιασμό των πινάκων που περιγράφουν τη λειτουργία των νευρωνικών επιπέδων ενώ γίνεται επεξήγηση των εντολών pragma HLS pipeline και pragma HLS unroll των οποίων ο ρόλος για την επίτευξη της επιτάχυνσης των υπολογισμών κρίνεται καθοριστικός.

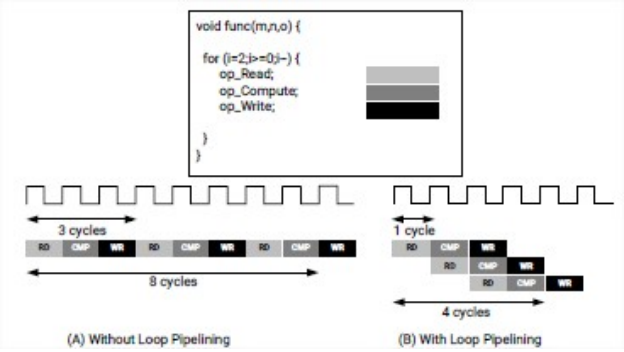
Η οδηγία Pragma HLS Unroll

Η συγκεκριμένη οδηγία χρησιμοποιείται για το «ξετύλιγμα» (unroll) κάποιου βρόχου δημιουργώντας αυτόνομες λειτουργίες αντί για μία συλλογή λειτουργιών. Το αποτέλεσμα είναι η δημιουργία πολλαπλών αντιγράφων του περιεχομένου του βρόχου στο hardware που επιτρέπουν την ταυτόχρονη εκτέλεση των επιμέρους επαναλήψεων του βρόχου. Η προκαθορισμένη λειτουργία όταν πρέπει να υλοποιηθεί κάποιος βρόχος στο hardware είναι η δημιουργία συγκεκριμένης λογικής για την εκτέλεση μίας επανάληψης του βρόχου, η οποία λογική χρησιμοποιείται επαναληπτικά και για τις επόμενες επαναλήψεις. Με τη συγκεκριμένη εντολή παρέχεται η δυνατότητα είτε για πλήρες είτε για μερικό unroll. Για μερικό unroll, η παράμετρος factor καθορίζει το πλήθος των αυτόνομων διεργασιών που επιθυμεί να δημιουργήσει ο χρήστης. [27]

Η οδηγία Pragma HLS Pipeline

Με τη συγκεκριμένη οδηγία είναι εφικτή η μείωση του διαστήματος έναρξης (initiation interval) μίας συνάρτησης ή κάποιου loop επιτρέποντας την ταυτόχρονη εκτέλεση επιμέρους διεργασιών. Μία pipelined συνάρτηση ή loop μπορεί να επεξεργαστεί μία νέα είσοδο κάθε N κύκλους ρολογιού όπου N αποτελεί το διάστημα έναρξης. Το προκαθορισμένο initiation interval που προσπαθεί ο compiler να επιτύχει είναι 1, δηλαδή μία νέα είσοδος επεξεργάζεται σε κάθε κύκλο ρολογιού. Ωστόσο, το initiation interval δύναται να επιλεχθεί και από το χρήστη

μέσω της παραμέτρου II. Σε περίπτωση που το initiation interval II δεν μπορεί να επιτευχθεί, το τελικό design επιτυγχάνει το μικρότερο δυνατό. [27]



Σχήμα 2.12: Επιτάχυνση συνάρτησης με αρχικό initiation interval ίσο με 3. [27]

Για την επιτάχυνση των υπολογισμών που πραγματοποιεί το δίκτυο, όπως παρουσιάστηκε και στην παράγραφο 2.4.4, έχει δημιουργηθεί κατάλληλη αρχιτεκτονική μνήμης προκειμένου να γίνεται παραλληλοποίηση του υπολογισμού επιμέρους γινομένων εισόδων – συναπτικών βαρών. Η παραλληλοποίηση αυτή, στο πρώτο νευρωνικό επίπεδο έγκειται στο επίπεδο των ίδιων των νευρώνων, ενώ στην περίπτωση του δεύτερου και του τρίτου επιπέδου στο επίπεδο των εισόδων των νευρώνων για τους λόγους που αναφέρθηκαν. Παρακάτω, γίνεται ανάλυση των δύο διαφορετικών μεθόδων ενώ παρουσιάζεται και η High-level Synthesis περιγραφή τους για τις περιπτώσεις του πρώτου και του δεύτερου επιπέδου.

Παραλληλοποίηση υπολογισμών στο επίπεδο των νευρώνων

Όπως έχει ήδη αναφερθεί, οι έξοδοι ενός νευρωνικού επιπέδου m νευρώνων, δίνονται από τον πολλαπλασιασμό του διανύσματος των n εισόδων, x, με έναν πίνακα W ο οποίος περιέχει σε κάθε στήλη του τα συναπτικά βάρη ενός νευρώνα. Έτσι, το στοιχείο $w_{i,j}$ του πίνακα αναφέρεται στο συναπτικό βάρος της i-οστής εισόδου του j-οστού νευρώνα. Με την παραλληλοποίηση στο επίπεδο των νευρώνων, είναι εφικτή η ταυτόχρονη στάθμιση της i-οστής εισόδου, όλων των νευρώνων του επιπέδου. Το αποτελέσμα κάθε στάθμισης είτε αποθηκεύεται αυτούσιο στην αντίστοιχη θέση του διανύσματος εξόδου του νευρωνικού επιπέδου, αν η στάθμιση αφορά την πρώτη είσοδο, είτε προστίθεται στο προηγούμενο αποτελέσμα για την περίπτωση των υπόλοιπων εισόδων. Στον πίνακα 2.3, παρουσιάζεται η σειρά με την οποία πραγματοποιούνται οι υπολογισμοί. Πράξεις που ανήκουν στην ίδια γραμμή πραγματοποιούνται ταυτόχρονα.

Step	**Input**	**Neuron 1**	**Neuron 2**	.....	**Neuron m**
1	1	$x_0 w_{0,0}$	$x_0 w_{0,1}$		$x_0 w_{0,m-1}$
2	2	$x_1 w_{1,0}$	$x_1 w_{1,1}$		$x_1 w_{1,m-1}$
.....					
n	n	$x_{n-1} w_{n-1,0}$	$x_{n-1} w_{n-1,1}$		$x_{n-1} w_{n-1,m-1}$

Πίνακας 2.3: Σειρά πραγματοποίησης υπολογισμών στην περίπτωση της παραλληλοποίησης στο επίπεδο των νευρώνων.

Για την υλοποίηση των παραπάνω και την επιτάχυνση των υπολογισμών του πρώτου επιπέδου, δημιουργείται η εξής περιγραφή High-level Synthesis:

```
1 // Layer 1
2 layer_1:
3 for(int i=0; i<392; i++)
4 {
5   #pragma HLS PIPELINE II=1
6
7   for(int j=0; j<30; j++)
8   {
```

```
    #pragma HLS unroll factor=30
    l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j];
    quantized_type term = xbuf[i] * W1[i][j];
    layer_1_out[j] = last + term;
  }
}

```

Listing 2.18: Parallelization at neurons' level

Με εξαίρεση την ύπαρξη των εντολών pragma hls pipeline και pragma hls unroll, ο παραπάνω κώδικας θα αποτελούσε μία απλή υλοποίηση πολλαπλασιασμού διανύσματος με πίνακα σε γλώσσα C/C++ όπου οι είσοδοι βρίσκονται στον πίνακα x_buf, οι παράμετροι των νευρώνων του επιπέδου στον δισδιάστατο πίνακα W_1 και το διάνυσμα των εξόδων του επιπέδου στον πίνακα layer_1_out. Η προσθήκη της εντολής unroll στον εσωτερικό βρόχο επιτυγχάνει την δημιουργία hardware για τον ταυτόχρονο υπολογισμό των επιμέρους γινομένων που βρίσκονται στην ίδια γραμμή στον πίνακα 2.3 ενώ η εντολή Pipeline ευθύνεται για την ελαχιστοποίηση του initiation interval του εξωτερικού βρόχου. Στην εντολή Unroll, επιλέγεται παράγοντας ίσος με 30 εφόσον στο πρώτο νευρωνικό επίπεδο υπάρχουν 30 διαφορετικοί νευρώνες. Το loop unrolling είναι εφικτό εφόσον είναι δυνατή η ταυτόχρονη πρόσβαση στις αντίστοιχες παραμέτρους των νευρώνων εξαιτίας της κατάλληλης αρχιτεκτονικής μνήμης που επιλεχθηκε στην παράγραφο 2.4.3. Τέλος, η συνάρτηση ενεργοποίησης εφαρμόζεται στο παρακάτω τμήμα περιγραφής High-level Synthesis. Επισημαίνεται ότι επειδή δημιουργήθηκαν 30 διαφορετικές συναρτήσεις Relu στο hardware, ο παρακάτω βρόχος γίνεται unroll για τον ταυτόχρονο υπολογισμό των τελικών αποτελεσμάτων.

```
for(int i=0; i<30; i++)
  {
    #pragma HLS unroll factor=30
    layer_1_out[i] = ReLu(layer_1_out[i]);
  }

```

Listing 2.19: Applying activation function

Παραλληλοποίηση υπολογισμών στο επίεδο των εισόδων των νευρώνων

Με την παραλληλοποίηση στο επίπεδο των εισόδων των νευρώνων είναι εφικτή η ταυτόχρονη στάθμιση όλων των εισόδων του j-οστού νευρώνα. Τα αποτέλεσματα της στάθμισης όλων των εισόδων τροφοδοτούνται στην είσοδο της συνάρτησης ενεργοποίησης και η τελική έξοδος αποθηκεύεται στην αντίστοιχη θέση του διανύσματος εξόδου του νευρωνικού επιπέδου. Στον πίνακα 2.4 παρουσιάζεται η σειρά με την οποία πραγματοποιούνται οι υπολογισμοί. Σε αντίθεση με την παραλληλοποίηση στο επίπεδο των νευρώνων, ταυτόχρονα, δηλαδή σε ίδιες γραμμές του πίνακα, πραγματοποιούνται οι σταθμίσεις των εισόδων ενός συγκεκριμένου νευρώνα.

Step	**Neuron**	**Input 1**	**Input 2**	.....	**Input n**
1	1	$x_0 w_{0,0}$	$x_1 w_{1,0}$		$x_{n-1} w_{n-1,0}$
2	2	$x_0 w_{0,1}$	$x_1 w_{1,1}$		$x_{n-1} w_{n-1,1}$
.....					
m	m	$x_0 w_{0,m-1}$	$x_1 w_{1,m-1}$		$x_{n-1} w_{n-1,m-1}$

Πίνακας 2.4: Σειρά πραγματοποίησης υπολογισμών στην περίπτωση της παραλληλοποίησης στο επίπεδο των εισόδων των νευρώνων

Για την υλοποίηση των παραπάνω και την επιτάχυνση των υπολογισμών του δεύτερου επιπέδου, δημιουργείται η εξής περιγραφή High-level Synthesis:

```
// Layer 2
layer_2:
    for(int i=0; i<50; i++)
    {
      #pragma HLS PIPELINE II=1

      l_quantized_type result = 0;
      for(int j=0; j<30; j++)
      {
        #pragma HLS unroll factor=30
        l_quantized_type term = layer_1_out[j] * W2[j][i];
        result += term;
      }
      layer_2_out[i] = ReLu(result);
    }

```

Listing 2.20: Parallelization at neurons' inputs level

Στην παραπάνω περίπτωση, οι είσοδοι του νευρωνικού επιπέδου βρίσκονται στον πίνακα layer_1_out καθώς αποτελούν αποτελέσματα του προηγούμενου νευρωνικού επιπέδου, οι παράμετροι των νευρώνων στο δισδιάστατο πίνακα W_2 και το διάνυσμα των εξόδων του επιπέδου στον πίνακα layer_2_out. Ξανά, η προσθήκη της εντολής Unroll στον εσωτερικό βρόχο επιτυγχάνει τη δημιουργία hardware για τον ταυτόχρονο υπολογισμό των επιμέρους γινομένων που βρίσκονται σε κοινές γραμμές στον πίνακα 2.4 ενώ η εντολή Pipeline ευθύνεται για την ελαχιστοποίηση του initiation interval του εξωτερικού βρόχου. Ο παράγοντας του unroll είναι ίσος με 30 καθώς κάθε νευρώνας του επιπέδου διαθέτει τριάντα διαφορετικές εισόδους. Η πρόσβαση στις παραμέτρους του εκάστοτε νευρώνα καθώς και στις εισόδους του δύναται να γίνει ταυτόχρονα εφόσον επιλέχθηκε κατάλληλη αρχιτεκτονική μνήμης για να είναι δυνατή αυτή η παραλληλοποίηση. Η συνάρτηση ενεργοποίησης εφαρμόζεται κάθε φορά μετά τη στάθμιση των εισόδων στο βασικό βρόχο επανάληψης.

2.5 Εξαγωγή & Παρουσίαση Αποτελεσμάτων

Στο κεφάλαιο 2.4 πραγματοποιήθηκε παρουσίαση και επεξήγηση των επιλογών που λήφθηκαν για την επίτευξη της επιτάχυνσης των υπολογισμών που πραγματοποιεί το νευρωνικό δίκτυο. Στο παρόν κεφάλαιο παρουσιάζονται τα αποτελέσματα της παραπάνω διαδικασίας. Στην παράγραφο 2.5.1 γίνεται ανασκόπηση της διαδικασίας που πραγματοποιείται για τον προγραμματισμό του FPGA ενώ στη συνέχεια παρουσιάζονται λεπτομέρειες σχετικά με το τελικό design, τα σφάλματα κβάντισης για τους διάφορους τύπους δεδομένων που χρησιμοποιήθηκαν πριν την τελική επιλογή καθώς και το τελικό speed-up που προκύπτει.

2.5.1 Διαδικασία Δημιουργίας Bitstream

Για τον προγραμματισμό του FPGA μέσω του SDx IDE 2017.1 αρχικά δημιουργείται ένα καινούριο project στις ρυθμίσεις του οποίου επιλέγεται το target platform (στην παρούσα περίπτωση το ZYBO board) καθώς και λεπτομέρειες για το Software Platform (System configuration : Linux, Runtime : C/C++). Μετά τη δημιουργία του Project, προστίθενται σε αυτό τα κατάλληλα αρχεία που περιέχουν τον κώδικα που θα τρέξει στη CPU της πλατφόρμας, την High-level Synthesis περιγραφή του accelerator, καθώς και αρχεία που περιέχουν τις δηλώσεις τον πινάκων με τις παραμέτρους του μοντέλου και των συναρτήσεων ενεργοποίησης. Συγκεκριμένα, τα αρχεία που περιέχονται είναι τα εξής ενώ το πλήρες περιεχόμενο τους παρουσιάζεται στο παράρτημα.

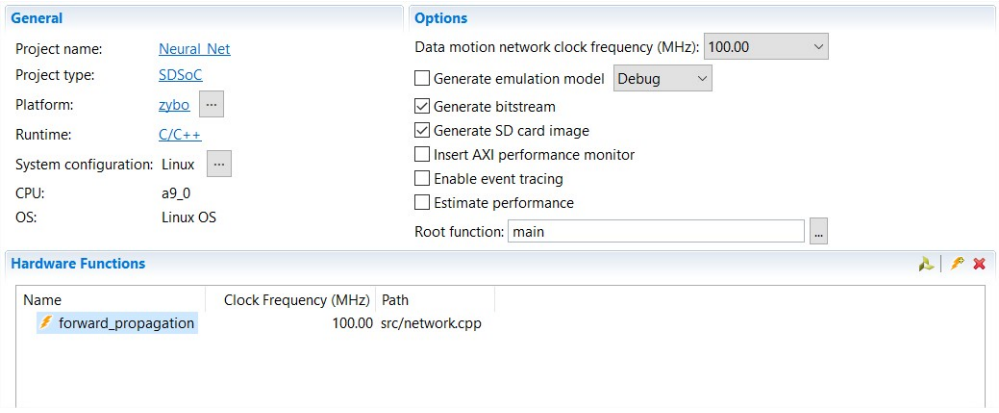
- main.cpp : Το αρχείο που περιέχει τον κώδικα C++ που εκτελείται από τη CPU της πλατφόρμας.
- software_weight_definitions.h : Το αρχείο που περιέχει τις δηλώσεις των πινάκων με τις παραμέτρους των νευρωνικών επιπέδων που χρησιμοποιούνται για τον υπολογισμό της εξόδου του μοντέλου σε επίπεδο software.
- network.h : Το header file που περιέχει τη δήλωση της hardware function, απαραίτητα definitions, λεπτομέρειες σχετικά με τον τρόπο μεταφοράς των δεδομένων από και προς το hardware καθώς και τους νεοορισθέντες τύπους δεδομένων.

- network.cpp : Το αρχείο με τη High-level Synthesis περιγραφή του κυκλώματος που υλοποιεί την επιτάχυνση των υπολογισμών.

- weight_definitions.h : Το αρχείο που εμπεριέχει τις δηλώσεις των πινάκων των νευρωνικών επιπέδων και γίνεται include στη High-level Synthesis περιγραφή για αποθήκευση των παραμέτρων στις block RAMs.

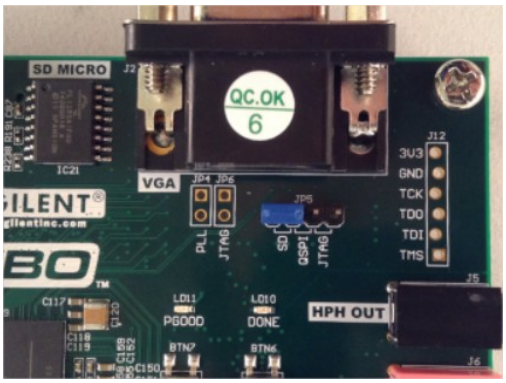
- tanh.h : Το αρχείο με τον πίνακα που κάνει χαρτογράφηση 4096 τιμών της συνάρτησης tanh και γίνεται include στη High-level Synthesis περιγραφή για αποθήκευση των παραμέτρων στις Block RAMs.

Σημειώνεται ότι τα αρχεία weight_definitions.h, software_weight_definitions.h και tanh.h παράγονται αυτόματα από τα αντίστοιχα Python scripts που παρουσιάζονται στο παράρτημα ενώ τα ίδια τα αρχεία δεν εμπεριέχονται στην παρούσα εργασία εξαιτίας του μεγάλου μεγέθους τους.
Για την παραγωγή του bitstream, πραγματοποιούνται οι ρυθμίσεις της εικόνας 2.13 για το Project προκειμένου να παραχθούν τα τελικά αρχεία τα οποία αφού μεταφερθούν σε κάρτα SD και αφού η κάρτα SD τοποθετηθεί στην πλατφόρμα, επιτυγχάνεται ο προγραμματισμός του FPGA.



Σχήμα 2.13: Ρυθμίσεις για την παραγωγή του τελικού bitstream.

Ιδιαίτερα χρήσιμη ήταν και η επιλογή «Estimate Performance» καθώς ήταν δυνατή η εκτίμηση της απόδοσης του κυκλώματος και της χρησιμοποίησης των υλικών πόρων του FPGA, χωρίς να είναι αναγκαία η παραγωγή του bitstream, γεγονός που επιτάχυνε τη διαδικασία ανάπτυξης και περιγραφής του κυκλώματος επιτρέποντας συνεχείς αλλαγές και πειραματισμούς στην High-level Synthesis περιγραφή.
Για τον προγραμματισμό της πλατφόρμας είναι δυνατή η χρήση microSD, QSPI Flash και JTAG. Προκειμένου να γίνει επιλογή της κάρτας SD ως μέσου προγραμματισμού της είναι απαραίτητη η σωστή τοποθέτηση του mode jumper (JP5) όπως παρουσιάζεται στην εικόνα 2.14.



Σχήμα 2.14: Ρύθμιση του boot mode σε microSD.

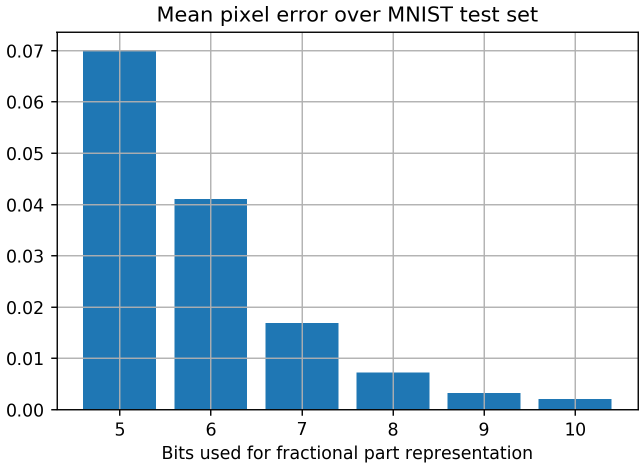
Τέλος, η πλατφόρμα συνδέεται με τον προσωπικό υπολογιστή με τη χρήση καλωδίου USB και μέσω του SDx terminal είναι δυνατός ο έλεγχός της μέσα από το περιβάλλον SDx IDE 2017.1

2.5.2 Σφάλματα Κβάντισης

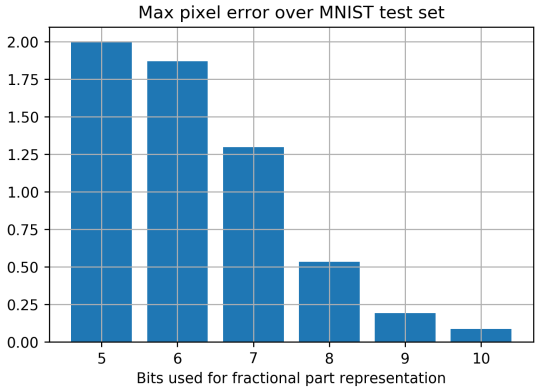
Στην παρούσα παράγραφο, παρουσιάζονται τα σφάλματα που προκύπτουν στην τελική έξοδο του FPGA εξαιτίας της κβάντισης που έχει συμβεί σε παραμέτρους, εισόδους και εξόδους. Η σύγκριση πάντα γίνεται με τις τιμές που προκύπτουν από την εκτέλεση των υπολογισμών σε software όπου γίνεται χρήση αριθμών κινητής υποδιαστολής οι οποίοι διαθέτουν πολύ μεγαλύτεροι ακρίβεια. Τα αποτελέσματα λαμβάνονται από την ανακατασκευή και των 10000 εικόνων που βρίσκονται στο test set, αφού τα δεδομένα εισόδου εγγραφούν σε αρχείο csv και γίνει parsing αυτών από τη CPU της πλατφόρμας. Στη συνέχεια, κάθε ημιτελής εικόνα μεταφέρεται στο FPGA για την επιταχυμένη ανακατασκευή της ενώ το σύνολο των αποτελεσμάτων εγγράφεται με τη σειρά του σε αρχείο csv. Τα διάφορα σφάλματα προκύπτουν από χρήση διαφορετικού πλήθους bits για την αναπαράσταση του δεδικαδικού μέρους των αριθμών σταθερής υποδιαστολής που χρησιμοποιούνται στο FPGA.
Τα αποτελέσματα του πίνακα 2.5 και των σχημάτων 2.15 και 2.16 λαμβάνονται με χρήση 5 έως 10 ψηφίων για την αναπαράσταση του δεκαδικού μέρους των αριθμών σταθερής υποδιαστολής που χρησιμοποιούνται στην προγραμματιζόμενη λογική. Για κάθε διαφορετική υλοποίηση, είναι αναγκαία και η τροποποίηση της tanh συνάρτησης ενεργοποίησης και συγκεκριμένα του πίνακα που διαθέτει τις αντιστοιχίσεις των τιμών εισόδου με τις τιμές εξόδου, με σωστό mapping του διαφορετικού κάθε φορά πλήθους των δυνατών εισόδων της. Για παράδειγμα, όταν χρησιμοποιούνται 7 bits για την αναπαράσταση του δεδικαδικού μέρους των αριθμών, στο διάστημα [-2,2) υπάρχουν 512 διαφορετικές κβαντισμένες τιμές ενώ με χρήση 8 bits, οι δυνατές κβαντισμένες τιμές είναι 1024.

Fraction Part (bits)	Quantum Size	Mean Error	Max Error
5	0.03125	0.07007	1.99792
6	0.015625	0.041027	1.870625
7	0.0078125	0.0168661	1.297707
8	0.00390625	0.00722744	0.532246
9	0.001953125	0.003212701	0.192475
10	0.0009765625	0.0020825756	0.087019

Πίνακας 2.5: Μέσο και μέγιστο σφάλμα κβάντισης υπολογισμένο σε όλο το test set.



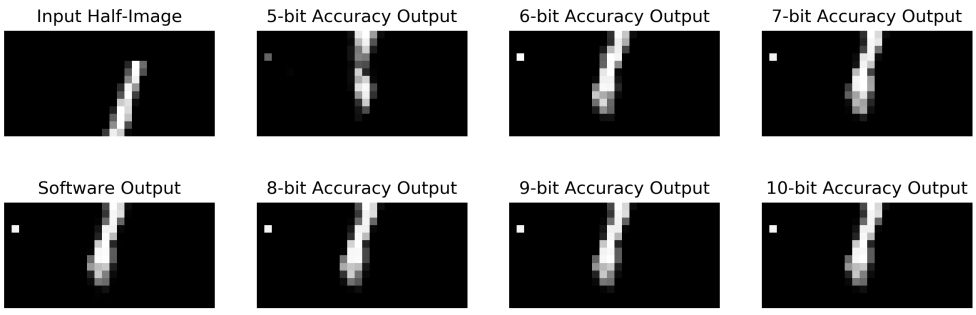
Σχήμα 2.15: Μέσες τιμές σφάλματος pixel για χρήση διαφορετικού πλήθους bits για την αναπαράσταση του δεκαδικού μέρους αριθμών.



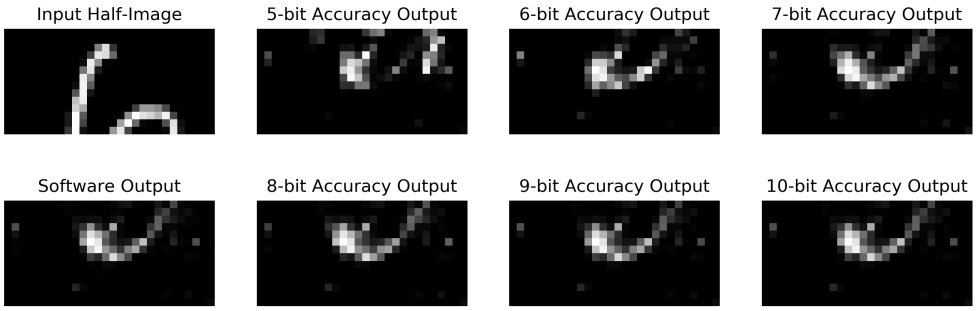
Σχήμα 2.16: Μέγιστες τιμές σφάλματος pixel για χρήση διαφορετικού πλήθους bits για την αναπαράσταση του δεκαδικού μέρους αριθμών.

Παρατηρείται η σταδιακή μείωση του μέσου και του μέγιστου σφάλματος που εμφανίζονται για χρήση όλο και περισσότερων bits για την αναπαράσταση του δεκαδικού μέρους των αριθμών εντός του FPGA. Παρόλο που ένα μέσο σφάλμα σαν αυτό της περίπτωσης των 5 bits μπορεί να θεωρηθεί μηδαμινό, εφόσον μία διαφορά 0.7 σε ένα pixel που μπορεί να λάβει τιμές από -1 (μαύρο) έως 1 (λευκό) είναι ανεπαίσθητη, παρατηρείται ότι μεγαλύτερα σφάλματα προκύπτουν σε σημεία ενδιαφέροντος των ανακατασκευασμένων εικόνων, δηλαδή σημεία που δεν είναι μαύρα. Εφόσον τα μαύρα τμήματα είναι κατά πολύ περισσότερα, ο μέσος όρος σφάλματος μειώνεται παρόλο που εμφανίζονται αρκετά μεγαλύτερα σφάλματα σε μη μαύρες περιοχές. Γι' αυτό το λόγο, η τελική επιλογή γίνεται με γνώμονα τη μετρική Max Error όπου είναι το μέγιστο σφάλμα που εμφανίζεται σε όλο το test set. Όταν το μέγιστο σφάλμα είναι επαρκώς μικρό, μπορεί να γίνει η τελική επιλογή πλήθους bits για την αναπαράσταση του δεκαδικού μέρους των αριθμών. Η τελική επιλογή αφορά, όπως έχει ήδη αναφερθεί, 10 bits, ενώ στα σχήματα 2.17, 2.18 και 2.19 παρουσιάζονται οι οπτικές διαφορές μεταξύ ανακατασκευασμένων τμημάτων εικόνων στα οποία χρησιμοποιείται διαφορετικό πλήθος bits.

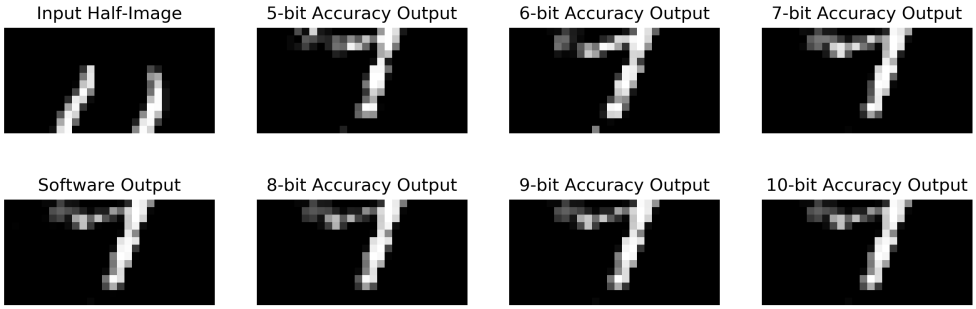
Η σύγκριση κάθε φορά, όπως αναφέρθηκε, γίνεται με τα αποτελέσματα που προκύπτουν στο software καθώς λόγω της χρήσης floating point αριθμών η ακρίβεια των αποτελεσμάτων είναι πολύ μεγαλύτερη γι΄ αυτό και έχει υλοποιηθεί αντίστοιχη συνάρτηση σε C++ η οποία καλούμενη στη main συνάρτηση πραγματοποιεί τους αντίστοιχους υπολογισμούς. Διαφοροποίηση με τα αποτελέσματα που προκύπτουν στο software οφείλεται στην περιορισμένη ακρίβεια που είναι απόρροια της χρήσης fixed point δεκαδικών αριθμών.



Σχήμα 2.17: Αποτελέσματα ανακατασκευής ψηφίου 1 με χρήση διαφορετικού πλήθους bit για την αναπαράσταση του δεκαδικού μέρους αριθμών στο FPGA.



Σχήμα 2.18: Αποτελέσματα ανακατασκευής ψηφίου 6 με χρήση διαφορετικού πλήθους bit για την αναπαράσταση του δεκαδικού μέρους αριθμών στο FPGA.



Σχήμα 2.19: Αποτελέσματα ανακατασκευής ψηφίου 4 με χρήση διαφορετικού πλήθους bit για την αναπαράσταση του δεκαδικού μέρους αριθμών στο FPGA.

2.5.3 Χρησιμοποίηση Υλικών Πόρων & Λεπτομέρειες Τελικού Design

Στην παρούσα παράγραφο γίνεται ανάλυση της κατανάλωσης των υλικών πόρων που διαθέτει το FPGA ενώ ύστερα γίνεται παρουσίαση περεταίρω λεπτομερειών που αφορούν το τελικό design, όπως το χρονικό κόστος της μεταφοράς δεδομένων από και προς το FPGA, ο βαθμός του pipelining που έχει επιτευχθεί σε διάφορα τμήματα της συνολικής διαδικασίας κ.α.

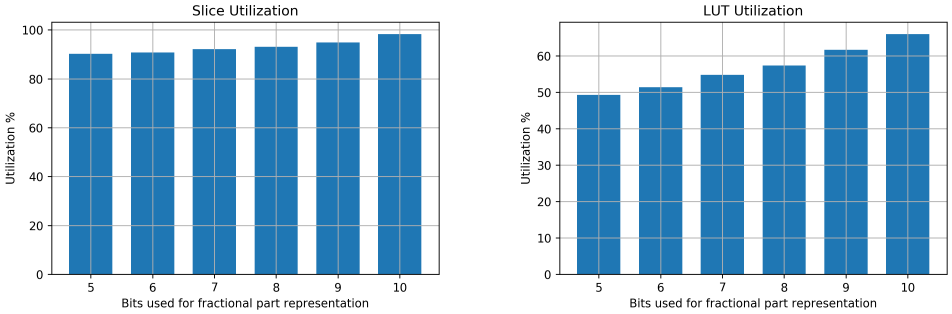
Πριν την ανάλυση της κατανάλωσης των φυσικών πόρων του FPGA, είναι ωφέλιμο να γίνει μία σύντομη παρουσίαση της εσωτερικής αρχιτεκτονικής που διαθέτουν τα FPGA της σειράς 7 της Xilinx (Xilinx 7 Series FPGAs) και συγκεκριμένα των Configurable Logic Blocks τα οποία έχουν ήδη αναφερθεί στην παράγραφο 2.1.1. Όπως αναφέρεται εκεί, το FPGA διαθέτει την ικανότητα αναδιαμόρφωσης της εσωτερικής του δομής με τελικό στόχο την υλοποίηση συγκεκριμένων κυκλωμάτων που έχουν σκοπό την υλοποίηση λογικών συναρτήσεων. Θεμελιώδης μονάδα του FPGA με την ικανότητα αναδιαμόρφωσης της για την πραγματοποίηση των παραπάνω στόχων είναι τα Configurable Logic Blocks.

Κάθε CLB των FPGA της σειράς 7 αποτελείται από δύο Slices, ενώ κάθε Slice αποτελείται από 4 LUTs έξι εισόδων, 8 Flip-Flops και πολυπλέκτες. Συνολικά 4 flip-flops, ένα για κάθε LUT δύνανται να χρησιμοποιηθούν ως latches, ωστόσο σε αυτή την περίπτωση, τα τέσσερα εναπομείναντα οφείλουν να μείνουν αχρησιμοποίητα. Περίπου τα δύο τρίτα των Slices είναι SLICEL λογικά Slices ενώ τα υπόλοιπα είναι SLICEM τα οποία μπορούν να χρησιμοποιήσουν τα LUTs τους για την υλοποίηση κατανεμημένης 64-bit RAM ή shift registers [28]. Το chip Z-7010 που διαθέτει η πλατφόρμα ZYBO διαθέτει 4400 slices δίνοντας τελικά 17600 LUTs και 35200

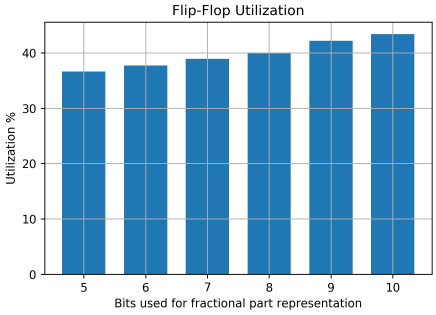
Flip-Flops όπως παρουσιάστηκε και στην παράγραφο 2.3.1. Στον πίνακα 2.6 παρουσιάζεται η χρησιμοποίηση των πόρων του FPGA για τις διάφορες επιλογές που έγιναν για το πλήθος των bits που χρησιμοποιείται για την αναπαράσταση του δεκαδικού μέρους των αριθμών εντός της προγραμματιζόμενης λογικής. Στα σχήματα 2.20 και 2.21 εμφανίζονται τα διαγράμματα με την ποσοστιαία χρησιμοποίηση κάθε πόρου ξανά για τα διάφορα πλήθη bits.

Resource	**5-bits Accuracy**	**6-bits Accuracy**	**7-bits Accuracy**	**8-bits Accuracy**	**9-bits Accuracy**	**10-bits Accuracy**	**Total**
Slices	3966	3992	4051	4092	4173	4323	4400
LUTs	8669	9042	9625	10084	10842	11602	17600
Flip-Flops	12885	13267	13688	14084	14841	15270	35200
BRAMs	49.5	51	51	51	51.5	52	60
DSP	75	76	80	80	80	80	80

Πίνακας 2.6: Χρησιμοποίηση υλικών πόρων για τις διάφορες ακρίβειες του δεκαδικού μέρους των αριθμών.



Σχήμα 2.20: Ποσοστά χρησιμοποίησης slices και LUTs.



Σχήμα 2.21: Ποσοστά χρησιμοποίησης Flip-Flops.

Όπως έχει ήδη αναφερθεί, το τελικό design διαθέτει για την αναπαράσταση του δεκαδικού μέρους των αριθμών 10 bits. Λεπτομέρειες για τη χρησιμοποίηση των υλικών πόρων που αφορούν αποκλειστικά στο τελικό design βρίσκονται στον πίνακα 2.7.

Resource	Used	Total	Utilization (%)
Slices	4323	4400	98.25
LUTs	11602	17600	65.92
Flip-Flops	15270	35200	43.38
BRAMs	52	60	86.67
DSP	80	80	100

Πίνακας 2.7: Χρησιμοποίηση πόρων τελικού design.

Γίνεται αντιληπτό ότι χρήση περισσότερων bits για την αναπαράσταση των δεκαδικών αριθμών ναι μεν προσφέρει μεγαλύτερη ακρίβεια ωστόσο έχει άμεσο αντίκτυπο στους καταναλισκόμενους υλικούς πόρους. Εφόσον στην παρούσα εφαρμογή το FPGA χρησιμοποιείται αποκλειστικά για την επιτάχυνση του μοντέλου και όχι για κάποια ακόμα εφαρμογή που θα απαιτούσε με τη σειρά της μερίδιο των συνολικών διαθέσιμων πόρων, δύναται η αξιοποίηση όλων για τη βελτιστοποίηση του τελικού αποτελέσματος. Το bottleneck στο διαρκώς αυξανόμενο πλήθος bits για την αύξηση της ακρίβειας έρχεται από τη χρησιμοποίηση των slices, όπου με 10 bits για την αναπαράσταση του δεκαδικού μέρους φτάνει στο 98.25%.
Μετά το επιτυχές compilation του τελικού design, παρέχονται πληροφορίες από το περιβάλλον SDx IDE σχετικά με estimations που αφορούν τις καθυστερήσεις των επιμέρους τμημάτων του design. Στον πίνακα 2.8 παρουσιάζονται λεπτομέρειες σχετικά με τη μεταφορά δεδομένων προς και από το FPGA ενώ στον πίνακα 2.9 παρουσιάζονται λεπτομέρειες σχετικά με την καθυστέρηση και τα initiation intervals των επιμέρους επιπέδων καθώς και του συνολικού design.

IP Port	Transfer Size (bytes)	Paged or Contiguous	Datamover Setup Time (CPU cycles)	Transfer Time (CPU Cycles)
x	392*4	Contiguous	1118	3796
y	392*4	Contiguous	1118	3796

Πίνακας 2.8: Καθυστερήσεις μεταφοράς δεδομένων από και προς το FPGA.

Module	Module Section	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip Count
forward_propagation	-	-	1254	1255	-	-
	read_input	yes	395	1	5	392
	layer_1	yes	394	1	4	392
	layer_2	yes	54	1	6	50
	layer_3	yes	403	1	13	392
ReLU	-	-	0	1	1	-
Tanh	-	-	1	1	2	-

Πίνακας 2.9: Λεπτομέρειες καθυστέρησεων και βαθμού pipelining τελικού design και των επιμέρους σταδίων του.

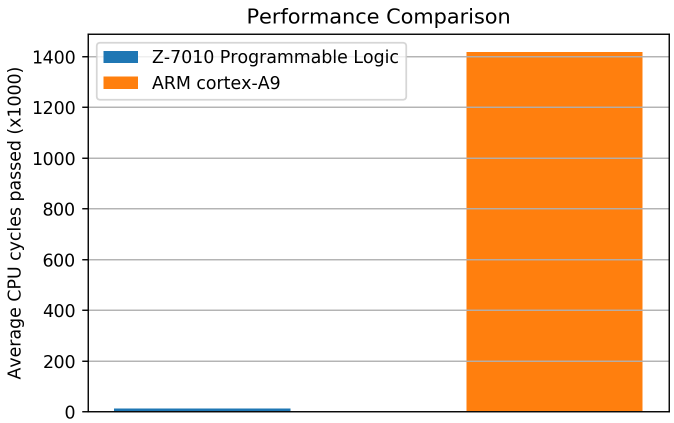
Όπως φαίνεται και στους πίνακες, στο τελικό design δεν γίνεται pipelining μεταξύ των επιπέδων το οποίο έχει ως αποτέλεσμα το initiation interval του ολικού design να είναι ίδιο με το latency του. Για να επιτευχθεί task level pipelining μεταξύ των νευρωνικών επιπέδων είναι απαραίτητη η χρήση της οδηγίας pragma hls dataflow η οποία μπορεί να επιτρέψει την επικάλυψη της λειτουργίας των νευρωνικών επιπέδων μειώνοντας latency και initiation interval. Ωστόσο, οι περιορισμένοι πόροι της εκπαιδευτικής πλατφόρμας που χρησιμοποιείται δεν επιτρέπουν τέτοιου είδους βελτιστοποίηση.

2.5.4 Αποτελέσματα Επιτάχυνσης Υπολογισμών

Στην παρούσα παράγραφο παρουσιάζονται τα αποτελέσματα που αφορούν τη σύγκριση των χρόνων υπολογισμού των εξόδων του νευρωνικού δικτύου αρχικά μεταξύ του επεξεργαστή ARM cortex-A9 που διαθέτει το ίδιο το chip Z-7010 και του dedicated κυκλώματος που υλοποιήθηκε στο FPGA με στόχο την επιτάχυνση της διαδικασίας. Ύστερα, παρουσιάζονται και συγκρίσεις των αποδόσεων μεταξύ και άλλων επεξεργαστών και του FPGA. Τέλος, γίνεται παρουσίαση και σύγκριση εικόνων ανακατασκευασμένων σε CPUs και εικόνων ανακατασκευασμένων στην προγραμματιζόμενη λογική.
Αρχικά, για τον υπολογισμό του χρόνου εκτέλεσης των υπολογισμών στον ARM cortex-A9 και στο FPGA, δημιουργείται μία κλάση perf_counter η οποία με χρήση της συνάρτησης sds_clock_counter της βιβλιοθήκης sds_lib.h της Xilinx, δίνει τη δυνατότητα μετρήσεων του χρόνου εκτέλεσης τμημάτων κώδικα σε κύκλους ρολογιού. Συγκεκριμένα, η συνάρτηση sds_clock_counter επιστρέφει την τιμή ενός μετρητή ο οποίος μετρά τους κύκλους ρολογιού του ARM επεξεργαστή. Στη συνέχεια ωστόσο, για τη σύγκριση των χρόνων εκτέλεσης μεταξύ διαφορετικών επεξεργαστών και του FPGA γίνεται χρήση της βιβλιοθήκης sys/time.h και συγκεκριμένα της συνάρτησης gettimeofday() για τον υπολογισμό των επιθυμητών χρονικών διαστημάτων. Στους πίνακες 2.10 και 2.11 και στα αντίστοιχα σχήματα, παρουσιάζονται οι μέσοι χρόνοι ανακατασκευής μίας εικόνας, χρόνοι που προέκυψαν από την ανακατασκευή όλων των εικόνων του test set του MNIST dataset. Σημειώνεται ότι ο υπολογισμός των εξόδων του νευρωνικού δικτύου στο επίπεδο software γίνεται μέσω της συνάρτησης sw_forward_propagation() στην οποία γίνονται οι ίδιοι πολλαπλασιασμοί πινάκων που πραγματοποιούνται και στο FPGA με τη διαφορά ότι τα στοιχεία των πινάκων διατρέχονται κατά γραμμή για την εκμετάλλευση των πλεονεκτημάτων που προσδίδουν οι caches των CPUs.

Hardware used for computation	**Average Cycles Passed**	**Speed-Up**
ARM cortex-A9	1417945	119.315
Z-7010 Programmable Logic	11884	

Πίνακας 2.10: Σύγκριση απόδοσης ARM cortex-A9 & και FPGA για την ανακατασκευή μίας εικόνας.

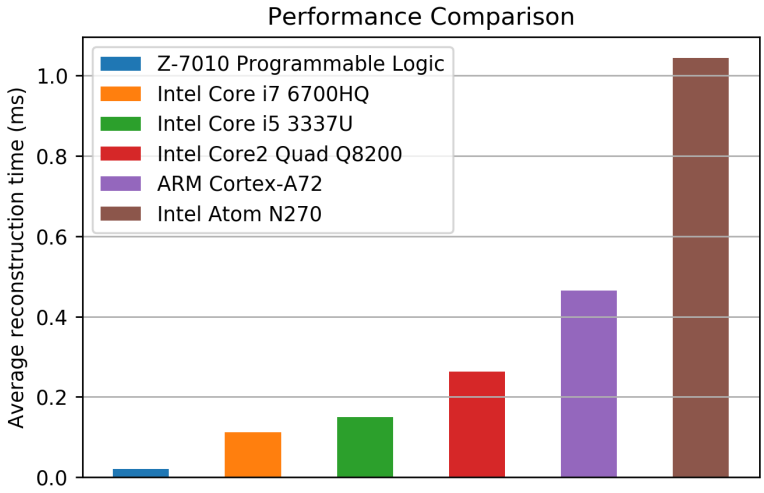


Σχήμα 2.22: Σύγκριση μέσης ταχύτητας ανακατασκευής προγραμαμτιζόμμενης λογικής και ARM cortex-A9 CPU.

Hardware used for computation	**Max clock frequency (GHz)**	**Average image reconstruction time (ms)**	**Programmable Logic Speed-Up**
Intel Atom N270	1.6	1.04409	53.128
Intel Core2 Quad Q8200	2.33	0.263327	13.399
Intel Core i5 3337U	2.7	0.149574	7.611
Intel Core i7 6700HQ	3.5	0.111135	5.655
ARM Cortex-A9	0.65	2.182310	111.047
ARM Cortex-A72	1.5	0.463971	23.609
Programmable Logic	0.1	0.019652	1

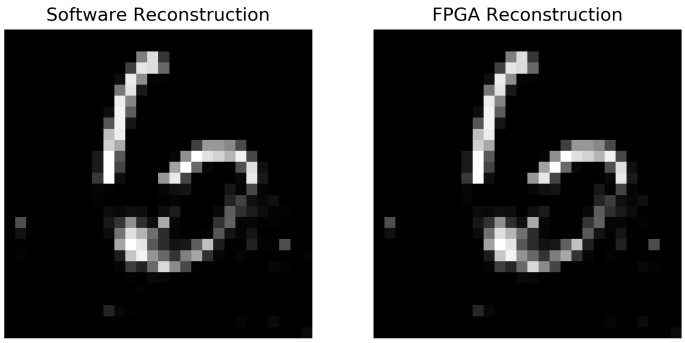
Πίνακας 2.11: Σύγκριση μέσου χρόνου ανακατασκευής μίας εικόνας σε διάφορες CPUs και στην προγραμματιζόμενη λογική.

Η επιτάχυνση στη συγκεκριμένη περίπτωση έχει μετρηθεί με τη βιβλιοθήκη sys/time.h και όχι με τη χρήση κάποιου counter των κύκλων ρολογιού. Γι' αυτό το λόγο, στην περίπτωση του επεξεργαστή ARM cortex-A9 τα αποτελέσματα της επιτάχυνσης είναι ελαφρώς διαφορετικά.

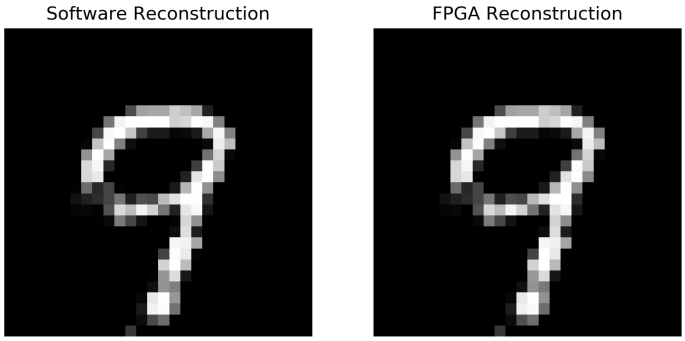


Σχήμα 2.23: Χρόνοι ανακατασκευής.

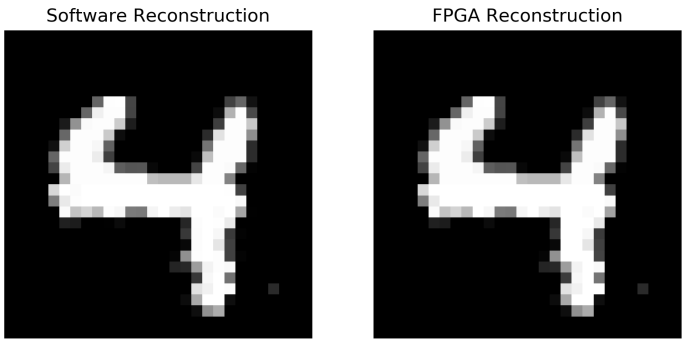
Παρατηρείται ότι ακόμα και σε άδικη σύγκριση μεταξύ της προγραμματιζόμενης λογικής του chip Z-7010 και επεξεργαστών με πολύ υψηλότερη συχνότητα ρολογιού όπως ο Intel Core i7 6700HQ, το FPGA καταφέρνει να περατώσει τους υπολογισμούς ταχύτερα. Σε μία embedded εφαρμογή, όπου η χρήση κάποιου επεξεργαστή σαν τον i7 δε θα ήταν δυνατή και οι εναλλακτικές συνήθως αποτελούν επεξεργαστές ARM, η επιτάχυνση που παρουσιάζεται είναι σημαντική.
Τέλος, στα επόμενα σχήματα παρουσιάζονται ανακατασκευασμένες εικόνες, αριστερά αυτές όπου η ανακατασκευή τους έχει πραγματοποιηθεί με την εκτέλεση κώδικα σε CPU και δεξιά αυτές όπου η ανακατασκευή τους έχει επιταχυνθεί και έχει πραγματοποιηθεί στην προγραμματιζόμενη λογική του FPGA. Σημειώνεται ότι στα παρακάτω σχήματα σημασία δεν έχει το κατά πόσο είναι πετυχημένη η ανακατασκευή, καθώς αυτό σχετίζεται με την ικανότητα του νευρωνικού δικτύου και έχει αναλυθεί στα αντίστοιχα κεφάλαια, αλλά το κατά πόσο τα αποτελέσματα που παράχθηκαν στην προγραμματιζόμενη λογική ταυτίζονται με αυτά που παράχθηκαν σε CPU.



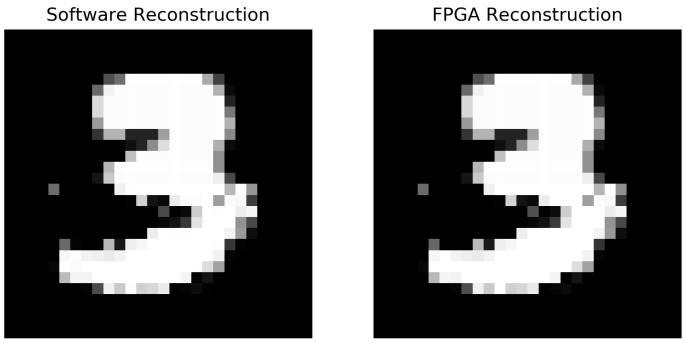
Σχήμα 2.24: Σύγκριση ανακατασκευής ψηφίου 6 με μέσο σφάλμα 0.003 και μέγιστο σφάλμα 0.036.



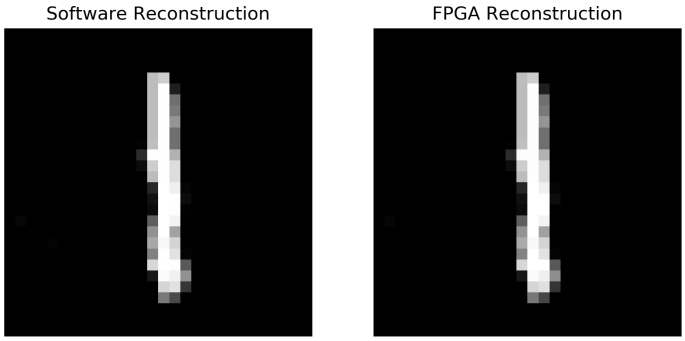
Σχήμα 2.25: Σύγκριση ανακατασκευής ψηφίου 9 με μέσο σφάλμα 0.002 και μέγιστο σφάλμα 0.045.



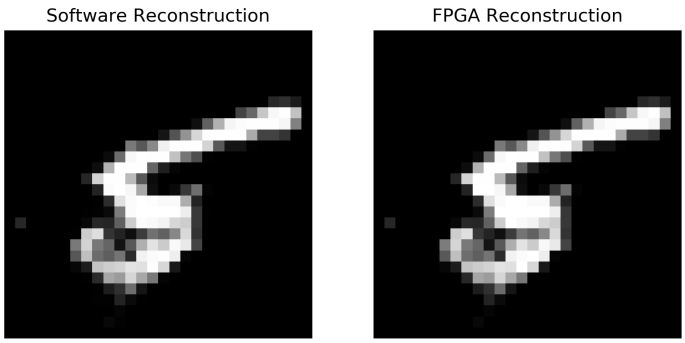
Σχήμα 2.26: Σύγκριση ανακατασκευής ψηφίου 4 με μέσο σφάλμα 0.001 και μέγιστο σφάλμα 0.034.



Σχήμα 2.27: Σύγκριση ανακατασκευής ψηφίου 3 με μέσο σφάλμα 0.002 και μέγιστο σφάλμα 0.055.



Σχήμα 2.28: Σύγκριση ανακατασκευής ψηφίου 1 με μέσο σφάλμα 0.001 και μέγιστο σφάλμα 0.029.



Σχήμα 2.29: Σύγκριση ανακατασκευής ψηφίου 5 με μέσο σφάλμα 0.002 και μέγιστο σφάλμα 0.034.

Επίλογος

Στο πρώτο μέρος της παρούσας εργασίας, ύστερα από την παρουσίαση του θεωρητικού υποβάθρου για τον τρόπο λειτουργίας των νευρωνικών δικτύων, έγιναν εμφανείς οι δυνατότητες των Generative Adversarial Networks, όπου με την ικανότητα τους να μετασχηματίζουν γεννετικά προβλήματα σε διακριτικά, επιτρέπουν την ανάπτυξη μοντέλων για την ολοκλήρωση διαδικασιών σαν αυτές της σύνθεσης και της ανακατασκευής εικόνων. Το τελικό μοντέλο που αναπτύχθηκε σε αυτό το μέρος έδινε άριστα αποτελέσματα ανακατασκευασμένων εικόνων. Ωστόσο, διέθετε τεράστιο αριθμό παραμέτρων εξαιτίας των πολλών νευρωνικών επιπέδων που διέθετε και του μεγάλου πλήθους νευρώνων σε κάθε ένα από αυτά.

Στο δεύτερο μέρος, αρχικά έγινε παρουσίαση θεμελιωδών στοιχείων που αφορούν τη δομή και τη λειτουργία των FPGA ενώ στη συνέχεια παρουσιάστηκε διεξοδικά η διαδικασία επιτάχυνσης των υπολογισμών που πραγματοποιεί το νευρωνικό δίκτυο με τη χρήση FPGA. Εφόσον έγινε χρήση της εκπαιδευτικής πλατφόρμας ZYBO της Digilent, η οποία φέρει το μικρότερο chip της σειράς 7 της Xilinx, Z-7010 με πολύ περιορισμένους διαθέσιμους υλικούς πόρους, η ανάπτυξη ενός μοντέλου με μικρότερο πλήθος παραμέτρων κρίθηκε αναγκαία. Μετά την σμίκρυνση του αρχικού νευρωνικού δικτύου, ακολούθησε ανάλυση των παραμέτρων και των εξόδων των νευρωνικών επιπέδων, κατάλληλη επιλογή πλήθους bits για την fixed point αριθμητική αναπαράσταση των αριθμών εντός του FPGA καθώς και παραλληλοποίηση των υπολογισμών στα διάφορα νευρωνικά επίπεδα με διαφορετικούς τρόπους.

Η αρχική σύγκριση του speed up που προκύπτει γίνεται μεταξύ της προγραμματιζόμενης λογικής του FPGA και του επεξεργαστή ARM cortex-A9 που διαθέτει το ίδιο το chip. Τα αποτελέσματα που λαμβάνονται κρίνονται ενθαρρυντικά καθώς επιτυγχάνονται ταχύτητες ανακατασκευής στο FPGA περίπου 119 φορές μεγαλύτερες από αυτές στη CPU με μέση απόκλιση τιμών στις εξόδους του FPGA 0.002 και μέγιστη 0.087 πάνω σε όλο το dataset που ανακατασκευάστηκε. Επιπλέον, η σύγκριση συνεχίστηκε και για άλλες περιπτώσεις επεξεργαστών όπως αυτή του Intel Core i7 6700HQ όπου το FPGA διατήρησε το προβάδισμα του. Για τις διάφορες σχεδιαστικές επιλογές που έγιναν παρουσιάστηκαν παράλληλα οι επιπτώσεις τους στα τελικά σφάλματα των εξόδων καθώς και στη χρησιμοποίηση των υλικών πόρων.

Ως μελλοντική εργασία, μπορεί αρχικά να γίνει βελτιστοποίηση του τελικού accelerator που αναπτύχθηκε για αύξηση του throughput του με στόχο την ανακατασκευή ολόκληρων batches εικόνων καθώς στην παρούσα εργασία η υλοποίηση του accelerator έγινε με γνώμονα τη βελτιστοποίηση του latency για την ανακατασκευή μίας εικόνας. Κάτι τέτοιο απαιτεί ωστόσο και χρήση FPGA με περισσότερους διαθέσιμους υλικούς πόρους. Στη συνέχεια, εφόσον όλη η εργασία έγινε γύρω από τα πλαίσια του MNIST dataset, τελικό στόχο μπορεί να αποτελέσει η ανακατασκευή εικόνων ανθρώπινων προσώπων όπου σε συνδυασμό με τη βελτιστοποίηση του accelerator από άποψη throughput, δύναται να επιτρέψει την real time ανακατασκευή ημιτελών προσώπων που εισέρχονται στο FPGA από αισθητήρες κάμερας.

Παράρτημα Α΄

Ανάπτυξη & Εκπαίδευση Νευρωνικών Δικτύων

Α΄.1 GANs για την εκ νέου σύνθεση εικόνων

```
import tensorflow as tf

import os
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

from tensorflow.keras import Sequential, Model
from tensorflow.keras.layers import Dense, Dropout, LeakyReLU, Activation, InputLayer, ReLU
from tensorflow.keras.activations import sigmoid, tanh
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import RandomNormal

path  = 'drive/My Drive/'
randomDim = 100

# Load MNIST data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5)/127.5
X_test = (X_test.astype(np.float32) - 127.5)/127.5
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)


adam = Adam(lr=0.0002, beta_1=0.5)

generator = Sequential()
generator.add(Dense(256, input_dim=randomDim, kernel_initializer=RandomNormal(stddev=0.4)))
generator.add(ReLU())
generator.add(Dense(512))
generator.add(ReLU())
generator.add(Dense(1024))
generator.add(ReLU())
generator.add(Dense(784))
generator.add(Activation(tanh))
generator.compile(loss='binary_crossentropy', optimizer=adam)

discriminator = Sequential()
discriminator.add(Dense(1024, input_dim=784, kernel_initializer=RandomNormal(stddev=0.02)))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(512))
```

```
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(256))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(1))
discriminator.add(Activation(sigmoid))
discriminator.compile(loss='binary_crossentropy', optimizer=adam)

# Combined network
discriminator.trainable = False
ganInput = tf.keras.Input(shape=(randomDim,))
x = generator(ganInput)
ganOutput = discriminator(x)
gan = Model(inputs=ganInput, outputs=ganOutput)
gan.compile(loss='binary_crossentropy', optimizer=adam)

dLosses = []
gLosses = []

# Plot the loss from each batch
def plotLoss(epoch):
    plt.figure(figsize=(10, 8))
    plt.plot(dLosses, label='Discriminitive loss')
    plt.plot(gLosses, label='Generative loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.grid('minor')
    plt.legend()
    plt.savefig(path + 'results/gan_loss_epoch_%d.png' % epoch)

# Create a wall of generated MNIST images
def plotGeneratedImages(epoch, examples=70, dim=(10, 7), figsize=(7, 10)):
    noise = np.random.normal(0, 1, size=[examples, randomDim])
    generatedImages = generator.predict(noise)
    generatedImages = generatedImages.reshape(examples, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generatedImages.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generatedImages[i], interpolation='nearest', cmap='gray')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig(path + 'results/gan_generated_image_epoch_%d.png' % epoch)

# Save the generator and discriminator networks (and weights) for later use
def saveModels(epoch):
    generator.save(path + 'results/gan_generator_epoch_%d.h5' % epoch)
    discriminator.save(path + 'results/gan_discriminator_epoch_%d.h5' % epoch)

def train(epochs=1, batchSize=128):
    batchCount = X_train.shape[0] / batchSize
    print('Epochs:', epochs)
    print('Batch size:', batchSize)
    print('Batches per epoch:', batchCount)

    for e in range(1, epochs+1):
        print('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in tqdm(range(int(batchCount))):
            # Get a random set of input noise and images
            noise = np.random.normal(0, 1, size=[batchSize, randomDim])
            imageBatch = X_train[np.random.randint(0, X_train.shape[0], size=batchSize)]

            # Generate fake MNIST images
            generatedImages = generator.predict(noise)
            # print np.shape(imageBatch), np.shape(generatedImages)
            X = np.concatenate([imageBatch, generatedImages])

```

```
            # Labels for generated and real data
            yDis = np.zeros(2*batchSize)
            # One-sided label smoothing
            yDis[:batchSize] = 0.9

            # Train discriminator
            discriminator.trainable = True
            dloss = discriminator.train_on_batch(X, yDis)

            # Train generator
            noise = np.random.normal(0, 1, size=[batchSize, randomDim])
            yGen = np.ones(batchSize)
            discriminator.trainable = False
            gloss = gan.train_on_batch(noise, yGen)

        # Store loss of most recent batch from this epoch
        dLosses.append(dloss)
        gLosses.append(gloss)

        if e < 10 or e % 50 == 0:
            plotGeneratedImages(e)

    # Plot losses from every epoch
    plotLoss(e)
    saveModels(e)

if __name__ == '__main__':

    os.mkdir(path + 'results')
    train(250)

```

Listing A.1: GANs for new image creation

A′.2 GANs για την ανακατασκευή εικόνων

```
import tensorflow as tf

import os
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

from tensorflow.keras import Sequential, Model
from tensorflow.keras.layers import Dense, Dropout, LeakyReLU, Activation, Input, ReLU,
    concatenate
from tensorflow.keras.activations import sigmoid, tanh
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import RandomNormal


path  = 'drive/My Drive/'

# Load MNIST data


(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5)/127.5                    # min = -1, max =
    1
X_train = X_train.reshape(60000, 784)
X_test = (X_test.astype(np.float32) - 127.5)/127.5                      # min = -1, max = 1
X_test = X_test.reshape(10000, 784)

# Create set of half images

X_train_cut = X_train[:,0:int(784/2)]
```

```
X_test_cut = X_test[:,0:int(784/2)]


# Fuction to combine network output with the cut image

def image_combiner(cut, generated):

  if(len(cut.shape) == 2):
    full_image = np.concatenate((cut,generated), axis = 1)
  else:
    full_image = np.concatenate((cut,generated))
  return full_image

def ganCreator():

    randomDim = int(784/2)
    optimizer = Adam(lr=0.0002, beta_1=0.5)

    #   Generative Network

    generator = Sequential()
    generator.add(Dense(256, input_dim = 392, kernel_initializer = RandomNormal(stddev=0.2))
    )
    generator.add(ReLU())
    generator.add(Dense(512))
    generator.add(ReLU())
    generator.add(Dense(1024))
    generator.add(ReLU())
    generator.add(Dense(392, activation = 'tanh'))
    generator.compile(loss='binary_crossentropy', optimizer= optimizer)

    # Discriminator Network

    discriminator = Sequential()
    discriminator.add(Dense(1024, input_dim = 784, kernel_initializer = RandomNormal(stddev
    =0.8)))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))
    discriminator.add(Dense(512))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))
    discriminator.add(Dense(256))
    discriminator.add(LeakyReLU(0.2))
    discriminator.add(Dropout(0.3))
    discriminator.add(Dense(1, activation = 'sigmoid'))
    discriminator.compile(loss='binary_crossentropy', optimizer= optimizer)

    # Combined Network - GAN

    discriminator.trainable = False
    ganInput = Input(shape=(392,))
    x = generator(ganInput)
    image = concatenate([ganInput,x])
    ganOutput = discriminator(image)
    gan = Model(inputs = ganInput, outputs = ganOutput)
    gan.compile(loss='binary_crossentropy', optimizer= optimizer)

    return gan,generator,discriminator

# Plot the loss from each batch
def plotLoss(epoch, gLosses, dLosses, path):
    plt.figure(figsize=(10, 8))
    plt.plot(dLosses, label='Discriminitive loss')
    plt.plot(gLosses, label='Generative loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.savefig(path + '/gan_loss_epoch_%d.png' % epoch)
```

```
96
97 # Create a wall of generated MNIST images
98 def plotGeneratedImages(epoch, path, examples=70, dim=(10, 7), figsize=(7, 10)):
99     example_in = X_test_cut[np.random.randint(0, X_test_cut.shape[0], size=examples)]
100     generatedImages = generator.predict(example_in)
101     fullImages = image_combiner(example_in, generatedImages)
102     fullImages = fullImages.reshape(examples, 28, 28)
103
104     plt.figure(figsize=figsize)
105     for i in range(fullImages.shape[0]):
106         plt.subplot(dim[0], dim[1], i+1)
107         plt.imshow(fullImages[i], interpolation='nearest', cmap='gray')
108         plt.axis('off')
109     plt.tight_layout()
110     image_name = path + '/gan_generated_image_epoch_' +str(epoch)+'.png'
111     plt.savefig(image_name)
112
113 # Save the generator and discriminator networks (and weights) for later use
114 def saveModels(generator, discriminator, epoch, path):
115     generator.save(path + '/gan_generator_epoch_%d.h5' % epoch)
116     discriminator.save(path + '/gan_discriminator_epoch_%d.h5' % epoch)
117
118 def train(gan, generator, discriminator, path, epochs=1, batchSize=128):
119     batchCount = X_train.shape[0] / batchSize
120     print('Epochs:', epochs)
121     print('Batch size:', batchSize)
122     print('Batches per epoch:', batchCount)
123
124     min_loss = 500
125     max_loss = -1
126
127     gLosses = []
128     dLosses = []
129
130     for e in range(1, epochs+1):
131         print('-'*15, 'Epoch %d' % e, '-'*15)
132         for _ in tqdm(range(int(batchCount))):
133
134
135             imageCutBatch = X_train_cut[np.random.randint(0, X_train_cut.shape[0], size=
    batchSize)]
136             imageBatch = X_train[np.random.randint(0, X_train_cut.shape[0], size=batchSize)]
137
138             # Generate fake MNIST images
139             generatedHalfImages = generator.predict(imageCutBatch)
140             generatedImages = image_combiner(imageCutBatch, generatedHalfImages)
141             X = np.concatenate([imageBatch, generatedImages])
142
143             # Labels for generated and real data
144             yDis = np.zeros(2*batchSize)
145             # One-sided label smoothing
146             yDis[:batchSize] = 0.9
147
148             # Train discriminator
149             discriminator.trainable = True
150             dloss = discriminator.train_on_batch(X, yDis)
151
152             # Train generator
153             imageCutBatch = X_train_cut[np.random.randint(0, X_train_cut.shape[0], size=
    batchSize)]
154             yGen = np.ones(batchSize)
155             discriminator.trainable = False
156             gloss = gan.train_on_batch(imageCutBatch, yGen)
157
158         # Store loss of most recent batch from this epoch
159         dLosses.append(dloss)
160         gLosses.append(gloss)
161
```

```
162         if e < 10 or e % 50 == 0:
163             plotGeneratedImages(e, path)
164 
165 
166     # Plot losses from every epoch
167     plotLoss(e, gLosses, dLosses, path)
168     saveModels(generator, discriminator, e, path = path)
169 
170 if __name__ == '__main__':
171 
172     os.mkdir(path + 'results')
173     gan, generator, discriminator = ganCreator()
174     train(gan, generator, discriminator, path+'results', epochs = 250, batchSize = 128)
175 
```

Listing A.2: GANs for image reconstruction

Παράρτημα Β΄

Διαδικασία Επιτάχυνσης Υπολογισμών Μοντέλου

Β΄.1 Συρίκνωση αρχικού μοντέλου

```
import tensorflow as tf

import os
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

from tensorflow.keras import Sequential, Model
from tensorflow.keras.layers import Dense, Dropout, LeakyReLU, Activation, Input, ReLU,
    concatenate
from tensorflow.keras.activations import sigmoid, tanh
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import RandomNormal
from tensorflow.keras.constraints import MinMaxNorm

path  = 'drive/My Drive/'

# Load MNIST data


(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5)/127.5                    # min = -1, max =
    1
X_train = X_train.reshape(60000, 784)
X_test = (X_test.astype(np.float32) - 127.5)/127.5                      # min = -1, max = 1
X_test = X_test.reshape(10000, 784)

# Create set of half images

X_train_cut = X_train[:,0:int(784/2)]
X_test_cut = X_test[:,0:int(784/2)]


# Fuction to combine network output with the cut image

def image_combiner(cut, generated):

  if(len(cut.shape) == 2):
    full_image = np.concatenate((cut,generated), axis = 1)
  else:
    full_image = np.concatenate((cut,generated))
  return full_image
```

```
42
43 def ganCreator():
44
45     randomDim = int(784/2)
46     optimizer = Adam(lr=0.0002, beta_1=0.5)
47
48     #   Generative Network
49
50     minmax = MinMaxNorm(min_value=-2, max_value=2, rate=1.0, axis=0)
51
52     generator = Sequential()
53     generator.add(Dense(30, input_dim = 392, use_bias=False, kernel_initializer =
       RandomNormal(stddev=0.4), kernel_constraint = minmax))
54     generator.add(ReLU())
55     generator.add(Dense(50, use_bias = False, kernel_constraint = minmax))
56     generator.add(ReLU())
57     generator.add(Dense(392, use_bias = False, activation='tanh', kernel_constraint = minmax
       ))
58     generator.compile(loss='binary_crossentropy', optimizer= optimizer)
59
60     # Discriminator Network
61
62     discriminator = Sequential()
63     discriminator.add(Dense(1024, input_dim = 784, kernel_initializer = RandomNormal(stddev
       =0.8)))
64     discriminator.add(LeakyReLU(0.2))
65     discriminator.add(Dropout(0.3))
66     discriminator.add(Dense(512))
67     discriminator.add(LeakyReLU(0.2))
68     discriminator.add(Dropout(0.3))
69     discriminator.add(Dense(256))
70     discriminator.add(LeakyReLU(0.2))
71     discriminator.add(Dropout(0.3))
72     discriminator.add(Dense(1, activation = 'sigmoid'))
73     discriminator.compile(loss='binary_crossentropy', optimizer= optimizer)
74
75     # Combined Network - GAN
76
77     discriminator.trainable = False
78     ganInput = Input(shape=(392,))
79     x = generator(ganInput)
80     image = concatenate([ganInput,x])
81     ganOutput = discriminator(image)
82     gan = Model(inputs = ganInput, outputs = ganOutput)
83     gan.compile(loss='binary_crossentropy', optimizer= optimizer)
84
85     return gan,generator,discriminator
86
87 # Plot the loss from each batch
88 def plotLoss(epoch, gLosses, dLosses, path):
89     plt.figure(figsize=(10, 8))
90     plt.plot(dLosses, label='Discriminitive loss')
91     plt.plot(gLosses, label='Generative loss')
92     plt.xlabel('Epoch')
93     plt.ylabel('Loss')
94     plt.legend()
95     plt.savefig(path + '/gan_loss_epoch_%d.png' % epoch)
96
97 # Create a wall of generated MNIST images
98 def plotGeneratedImages(epoch, path, examples=70, dim=(10, 7), figsize=(7, 10)):
99     example_in = X_test_cut[np.random.randint(0, X_test_cut.shape[0], size=examples)]
100    generatedImages = generator.predict(example_in)
101    fullImages = image_combiner(example_in, generatedImages)
102    fullImages = fullImages.reshape(examples, 28, 28)
103
104    plt.figure(figsize=figsize)
105    for i in range(fullImages.shape[0]):
106        plt.subplot(dim[0], dim[1], i+1)
```

```
        plt.imshow(fullImages[i], interpolation='nearest', cmap='gray')
        plt.axis('off')
    plt.tight_layout()
    image_name = path + '/gan_generated_image_epoch_' +str(epoch)+'.png'
    plt.savefig(image_name)

# Save the generator and discriminator networks (and weights) for later use
def saveModels(generator, discriminator, epoch, path):
    generator.save(path + '/gan_generator_epoch_%d.h5' % epoch)
    discriminator.save(path + '/gan_discriminator_epoch_%d.h5' % epoch)

def train(gan, generator, discriminator, path, epochs=1, batchSize=128):
    batchCount = X_train.shape[0] / batchSize
    print('Epochs:', epochs)
    print('Batch size:', batchSize)
    print('Batches per epoch:', batchCount)

    min_loss = 500
    max_loss = -1

    gLosses = []
    dLosses = []

    for e in tqdm(range(1, epochs+1)):
        print('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in range(int(batchCount)):


            imageCutBatch = X_train_cut[np.random.randint(0, X_train_cut.shape[0], size=
    batchSize)]
            imageBatch = X_train[np.random.randint(0, X_train_cut.shape[0], size=batchSize)]

            # Generate fake MNIST images
            generatedHalfImages = generator.predict(imageCutBatch)
            generatedImages = image_combiner(imageCutBatch, generatedHalfImages)
            X = np.concatenate([imageBatch, generatedImages])

            # Labels for generated and real data
            yDis = np.zeros(2*batchSize)
            # One-sided label smoothing
            yDis[:batchSize] = 0.9

            # Train discriminator
            discriminator.trainable = True
            dloss = discriminator.train_on_batch(X, yDis)

            # Train generator
            imageCutBatch = X_train_cut[np.random.randint(0, X_train_cut.shape[0], size=
    batchSize)]
            yGen = np.ones(batchSize)
            discriminator.trainable = False
            gloss = gan.train_on_batch(imageCutBatch, yGen)

        # Store loss of most recent batch from this epoch
        dLosses.append(dloss)
        gLosses.append(gloss)

        if e < 10 or e % 50 == 0:
            plotGeneratedImages(e, path)


    # Plot losses from every epoch
    plotLoss(e, gLosses, dLosses, path)
    saveModels(generator, discriminator, e, path = path)

if __name__ == '__main__':

    os.mkdir(path + 'results')
```

```
    gan, generator, discriminator = ganCreator()
    train(gan, generator, discriminator, path+'results', epochs = 250, batchSize = 128)

```

Listing B.1: Reduced GAN for image reconstruction

B′.2 Μελέτη παραμέτρων συρικνωμένου μοντέλου

```
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import load_model
from tensorflow.keras.datasets import mnist


(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5)/127.5        # min = -1, max = 1
X_train = X_train.reshape(60000, 784)
X_test = (X_test.astype(np.float32) - 127.5)/127.5          # min = -1, max = 1
X_test = X_test.reshape(10000, 784)

# Create set of half images

X_train_cut = X_train[:,0:int(784/2)]
X_test_cut = X_test[:,0:int(784/2)]


model = load_model("gan_generator_epoch_85.h5")
model.summary()

parameters = model.get_weights()

layer1 = np.asarray(parameters[0])
layer2 = np.asarray(parameters[1])
layer3 = np.asarray(parameters[2])

plt.plot(layer1.flatten())
plt.grid('minor')

plt.plot(layer2.flatten())
plt.grid('minor')

plt.plot(layer3.flatten())
plt.grid('minor')

plt.title('Dense layer parameters')
plt.ylabel('Parameter range')
plt.legend(['Layer1 values', 'Layer2 values', 'Layer3 values'])

input_data = X_test_cut[:].reshape(X_test_cut.shape[0],1,392)
out1 = np.dot(input_data, layer1)
out1_max = out1.max()
out1_min = out1.min()
out1 = np.clip(out1, 0, None)

out2 = np.dot(out1, layer2)
out2_max = out2.max()
out2_min = out2.min()
out2 = np.clip(out2, 0, None)

out3 = np.dot(out2, layer3)
out3_max = out3.max()
out3_min = out3.min()
out3 = np.tanh(out3)

print(f"Maximum of out1 : {out1_max}")
```

```
print(f"Minimum of out1 : {out1_min}")
print(f"Maximum of out2 : {out2_max}")
print(f"Minimum of out2 : {out2_min}")
print(f"Maximum of out3 : {out3_max}")
print(f"Minimum of out3 : {out3_min}")

```

Listing B.2: Generator parameter analyzing

B′.3 High-level Synthesis περιγραφή FPGA accelerator

```
#ifndef _LAYER_H_
#define _LAYER_H_

#include <stdlib.h>
#include <ap_fixed.h>

#define N1 392
#define M1 30

#define N2 30
#define M2 50

#define N3 50
#define M3 392

typedef ap_fixed<12,2,AP_RND> quantized_type;
typedef ap_fixed<19,9,AP_RND> l_quantized_type;

#pragma SDS data copy(x[0:392], y[0:392])
#pragma SDS data access_pattern(x:SEQUENTIAL, y:SEQUENTIAL)
#pragma SDS data data_mover(x:AXIDMA_SIMPLE, y:AXIDMA_SIMPLE)
void forward_propagation(float *x, float *y);

#endif

```

Listing B.3: Hardware function header file network.h

```
#include "network.h"
#include "weight_definitions.h"
#include "tanh.h"

l_quantized_type ReLU(l_quantized_type res)
{
  if (res < 0)
    return 0;

  return res;
}

l_quantized_type tanh(l_quantized_type res)
{
  if (res >= 2)
    return 1;
  else if (res < -2)
    return -1;
  else
  {
    ap_int <12> i = res.range();      // Copy bit pattern to i
    return tanh_vals[2048 + i.to_int()];
  }
}

void forward_propagation(float *x, float *y)
{
  quantized_type xbuf[N1];
```

```
29  l_quantized_type layer_1_out[M1];
30  l_quantized_type layer_2_out[M2];
31
32  #pragma HLS array_partition variable=layer_1_out block factor=15 dim=1
33  #pragma HLS array_partition variable=layer_2_out block factor=25 dim=1
34  #pragma HLS array_partition variable=W1 block factor=15 dim=2
35  #pragma HLS array_partition variable=W2 block factor=15 dim=1
36  #pragma HLS array_partition variable=W3 block factor=25 dim=1
37
38  #pragma HLS allocation instances=tanh limit=1 function
39  #pragma HLS allocation instances=ReLU limit=30 function
40
41  read_input:
42  for (int i=0; i<N1; i++)
43  {
44    #pragma HLS PIPELINE II=1
45    xbuf[i] = x[i];
46  }
47
48  // Layer 1
49  layer_1:
50  for(int i=0; i<N1; i++)
51  {
52    #pragma HLS PIPELINE II=1
53
54    for(int j=0; j<M1; j++)
55    {
56      #pragma HLS unroll factor=30
57      l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j];
58      quantized_type term = xbuf[i] * W1[i][j];
59      layer_1_out[j] = last + term;
60    }
61  }
62  layer_1_act:
63  for(int i=0; i<M1; i++)
64  {
65    #pragma HLS unroll factor = 30
66    layer_1_out[i] = ReLU(layer_1_out[i]);
67  }
68
69  // Layer 2
70  layer_2:
71  for(int i=0; i<M2; i++)
72  {
73    #pragma HLS PIPELINE II=1
74
75    l_quantized_type result = 0;
76    for(int j=0; j<N2; j++)
77    {
78      #pragma HLS unroll factor=30
79      l_quantized_type term = layer_1_out[j] * W2[j][i];
80      result += term;
81    }
82    layer_2_out[i] = ReLU(result);
83  }
84
85  // Layer 3
86  layer_3:
87  for(int i=0; i<M3; i++)
88  {
89    #pragma HLS PIPELINE II=1
90
91    l_quantized_type result = 0;
92    for(int j=0; j<N3; j++)
93    {
94      #pragma HLS unroll factor=50
95      l_quantized_type term = layer_2_out[j] * W3[j][i];
96      result += term;
```

```
97      }
98      y[i] = tanh(result).to_float();
99    }
100 }
101
```

Listing B.4: Hardware function file network.cpp

B′.4 Μετρήσεις απόδοσης accelerator

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include "sds_lib.h"
5  #include "network.h"
6  #include "software_weight_definitions.h"
7  #include <math.h>
8  #include <sys/time.h>
9
10 #define INPUT_IMAGES 10000
11
12 class perf_counter
13 {
14 public:
15      uint64_t tot, cnt, calls, time_cnt;
16      struct timeval T0,T1,res;
17      struct timezone otinanai;
18      perf_counter() : tot(0), cnt(0), calls(0), time_cnt(0) {};
19      inline void reset()
20      {
21        tot = cnt = calls = 0;
22      }
23      inline void start()
24      {
25        cnt = sds_clock_counter();
26        calls++;
27      };
28      inline void stop()
29      {
30        tot += (sds_clock_counter() - cnt);
31      };
32      inline uint64_t avg_cpu_cycles()
33      {
34        return ((tot+(calls>>1)) / calls);
35      };
36 };
37
38 static inline float ReLu(float value)
39 {
40   if (value < 0)
41     return 0;
42   return value;
43 }
44
45 void sw_forward_propagation(float *input, float *output)
46 {
47   float layer_1_out[M1];
48   float layer_2_out[M2];
49
50   // Layer 1
51   for (int i=0; i<M1; i++)
52   {
53     float result = 0;
54     for (int j=0; j<N1; j++)
55     {
56       float term = input[j] * W1_sw[i][j];
57       result += term;
```

```
    }
    layer_1_out[i] = ReLu(result);
  }

  // Layer 2
  for (int i=0; i<M2; i++)
  {
    float result = 0;
    for (int j=0; j<N2; j++)
    {
      float term = layer_1_out[j] * W2_sw[i][j];
      result += term;
    }
    layer_2_out[i] = ReLu(result);
  }

  // Layer 3
  for (int i=0; i<M3; i++)
  {
    float result = 0;
    for (int j=0; j<N3; j++)
    {
      float term = layer_2_out[j] * W3_sw[i][j];
      result += term;
    }
    output[i] = tanh(result);
  }
}

void flush_array(char *ar, int size)
{
    for(int i=0; i<size; i++)
    {
        ar[i] = '\0';
    }
}

void copy_ar(float *source, float *dest)
{
  for (int i=0; i<392; i++)
    dest[i] = source[i];
}

void parse_dataset(float *input, int batch_number)
{
   /* Parse CSV file containing in each row pixel values for half
   cut MNIST images from the whole test set */

  FILE *fp;
  fp = fopen("data.txt", "r");

  int c;
  char number[20] = {'\0'};
  int i = 0;
  int I = 0;

  do
  {
    c = fgetc(fp);
    if (c != ';' && c != '\n')
    {
      number[i] = c;
      i++;
    }
    else
    {
      float value = atof(number);
      input[I] = value;
```

```
126         I++;
127         i=0;
128         flush_array(number,20);
129       }
130
131     }while ( c != EOF && I<batch_number*392);
132
133     fclose(fp);
134 }
135
136
137
138 using namespace std;
139 int main()
140 {
141   float *x, *y_hw, *y_sw;
142
143   x = (float *)malloc(N1 * INPUT_IMAGES * sizeof(float));
144   y_hw = (float *)malloc(M3 * INPUT_IMAGES * sizeof(float));
145   y_sw = (float *)malloc(M3 * INPUT_IMAGES * sizeof(float));
146
147   perf_counter hw_ctr, sw_ctr;
148
149   cout << "Starting dataset parsing..." << endl;
150   parse_dataset(x, INPUT_IMAGES);
151   cout << "Parsing finished..." << endl;
152
153   float *input, *output;
154
155   float *fpga_in, *fpga_out;
156   fpga_in = (float *)sds_alloc(N1 * sizeof(float));
157   fpga_out = (float *)sds_alloc(M3 * sizeof(float));
158
159   cout << "Starting hardware calculations..." << endl;
160
161   for (int i=0; i<INPUT_IMAGES; i++)
162   {
163
164     input = x + i*N1;
165     output = y_hw + i*N1;
166     copy_ar(input, fpga_in);
167
168     hw_ctr.start();
169     forward_propagation(fpga_in, fpga_out);
170     hw_ctr.stop();
171
172     copy_ar(fpga_out, output);
173   }
174
175   cout << "Hardware calculations finished..." << endl;
176
177   cout << "Starting software calculations..." << endl;
178
179   for (int i=0; i<INPUT_IMAGES; i++)
180   {
181     input = x + i*N1;
182     output = y_sw + i*N1;
183     sw_ctr.start();
184     sw_forward_propagation(input, output);
185     sw_ctr.stop();
186   }
187   cout << "Software calculations finished..." << endl;
188
189
190   uint64_t hw_cycles = hw_ctr.avg_cpu_cycles();
191   uint64_t sw_cycles = sw_ctr.avg_cpu_cycles();
192
193   double speedup = ((double) sw_cycles) / ((double) hw_cycles);
```

```

  cout << "Hardware cycles : " << t_hw << endl;
  cout << "Software cycles : " << t_sw << endl;
  cout << "Speed-Up        : " << speedup  << endl;

  // Save results to output.txt file

  FILE *fp;
  fp = fopen("output.txt", "w");

  fprintf(fp, "Software Results;Hardware Results\n");

  cout << "Saving results..." << endl;
  for(int i=0 ; i<INPUT_IMAGES * M3; i++)
  {
    fprintf(fp, "%f;%f\n",y_sw[i], y_hw[i]);
  }
  fclose(fp);


  free(x);
  free(y_hw);
  free(y_sw);
  sds_free(fpga_in);
  sds_free(fpga_out);
}

```

Listing B.5: Passed cycles comparison between FPGA and cortex-A9

B′.5 Μετρήσεις απόδοσης των CPUs

```
#include <iostream>
#include <stdlib.h>
#include <stdint.h>
#include "software_weight_definitions.h"
#include <math.h>
#include <sys/time.h>

#define INPUT_IMAGES 10000

#define N1 392
#define M1 30

#define N2 30
#define M2 50

#define N3 50
#define M3 392

class perf_counter
{
public:
     uint64_t calls, time_cnt;
     struct timeval T0,T1,res;
     struct timezone otinanai;
     perf_counter() :  calls(0), time_cnt(0) {};
     inline void reset()
     {
       time_cnt = calls = 0;
     }
     inline void time_start()
     {
       calls++;
       gettimeofday(&T0, &otinanai);
     }
     inline void time_stop()
```

```
    {
      gettimeofday(&T1, &otinanai);
      timersub(&T1, &T0, &res);
      time_cnt +=  res.tv_sec * 1000000 + res.tv_usec;
    }
    inline float avg_time()
    {
      // Return msec
      return (time_cnt/((float)calls))/1000;
    }
};

static inline float ReLu(float value)
{
  if (value < 0)
    return 0;
  return value;
}

void sw_forward_propagation(float *input, float *output)
{
  float layer_1_out[M1];
  float layer_2_out[M2];

  // Layer 1
  for (int i=0; i<M1; i++)
  {
    float result = 0;
    for (int j=0; j<N1; j++)
    {
      float term = input[j] * W1_sw[i][j];
      result += term;
    }
    layer_1_out[i] = ReLu(result);
  }

  // Layer 2
  for (int i=0; i<M2; i++)
  {
    float result = 0;
    for (int j=0; j<N2; j++)
    {
      float term = layer_1_out[j] * W2_sw[i][j];
      result += term;
    }
    layer_2_out[i] = ReLu(result);
  }

  // Layer 3
  for (int i=0; i<M3; i++)
  {
    float result = 0;
    for (int j=0; j<N3; j++)
    {
      float term = layer_2_out[j] * W3_sw[i][j];
      result += term;
    }
    output[i] = tanh(result);
  }
}

void flush_array(char *ar, int size)
{
    for(int i=0; i<size; i++)
    {
        ar[i] = '\0';
    }
}
```

```
104
105 void copy_ar(float *source, float *dest)
106 {
107   for (int i=0; i<392; i++)
108     dest[i] = source[i];
109 }
110
111 void parse_dataset(float *input, int batch_number)
112 {
113   // Parse CSV file containing in each row pixel values for half
114   // cut MNIST images from the whole test set
115
116   FILE *fp;
117   fp = fopen("data.txt", "r");
118
119
120   int c;
121   char number[20] = {'\0'};
122   int i = 0;
123   int I = 0;
124
125   do
126   {
127
128     c = fgetc(fp);
129     if (c != ';' && c != '\n')
130     {
131       number[i] = c;
132       i++;
133     }
134     else
135     {
136       float value = atof(number);
137       input[I] = value;
138       I++;
139       i=0;
140       flush_array(number,20);
141     }
142
143   }while ( c != EOF && I<batch_number*392);
144
145   fclose(fp);
146 }
147
148 using namespace std;
149 int main()
150 {
151   float *x, *y_sw;
152
153   x = (float *)malloc(N1 * INPUT_IMAGES * sizeof(float));
154   y_sw = (float *)malloc(M3 * INPUT_IMAGES * sizeof(float));
155
156   perf_counter sw_ctr;
157
158   cout << "Starting dataset parsing..." << endl;
159   parse_dataset(x, INPUT_IMAGES);
160   cout << "Parsing finished..." << endl;
161
162   float *input, *output;
163
164
165
166   cout << "Starting software calculations..." << endl;
167
168   for (int i=0; i<INPUT_IMAGES; i++)
169   {
170     input = x + i*N1;
171     output = y_sw + i*N1;
```

```
172     sw_ctr.time_start();
173     sw_forward_propagation(input, output);
174     sw_ctr.time_stop();
175   }
176   cout << "Software calculations finished..." << endl;
177
178
179   float t_sw = sw_ctr.avg_time();
180
181   cout << "Software time : " << t_sw << endl;
182
183   FILE *fp;
184
185   fp = fopen("output.txt", "w");
186
187   fprintf(fp, "Software Results;Hardware Results\n");
188
189   cout << "Saving results..." << endl;
190   for(int i=0 ; i<INPUT_IMAGES * M3; i++)
191   {
192     fprintf(fp, "%f;\n",y_sw[i]);
193   }
194   fclose(fp);
195
196
197   free(x);
198   free(y_sw);
199 }
200
```

Listing B.6: Calculations for CPU measurements

Παράρτημα Γ′

Εργαλεία

Γ′.1 Αποθήκευση παραμέτρων μοντέλου

```python
import numpy as np
from tensorflow.keras.models import load_model

TRANSPOSE = False

def print_line(f,line):

    f.write("{ ")
    for i in range(0,len(line)-1):
        f.write(f"{line[i]}, ")
    f.write(f"{line[-1]}")
    f.write(" }")


if __name__ == "__main__":

    model = load_model("gan_generator_epoch_85.h5")
    model.summary()

    parameters = model.get_weights()
    layers = []

    for i in range(0,len(parameters)):
        if TRANSPOSE:
            layers.append(np.asarray(parameters[i]).transpose())
        else:
            layers.append(np.asarray(parameters[i]))

    f_sw = open("softwate_weight_definitions.h", "w")
    f_hw = open("weight_definitions.h", "w")

    for k, layer in enumerate(layers):
        size = layer.shape

        f_hw.write(f"quantized_type W{k+1}[{size[0]}][{size[1]}] = ")
        f_sw.write(f"float W{k+1}_sw[{size[0]}][{size[1]}] = ")
        f_hw.write("{\n")
        f_sw.write("{\n")

        for i in range(0,size[0]-1):
            print_line(f_hw, layer[i,:])
            f_hw.write(",\n")
            print_line(f_sw, layer[i,:])
            f_sw.write(",\n")

        print_line(f_hw, layer[-1,:])
```

```
        f_hw.write("};\n\n\n")
        print_line(f_sw, layer[-1,:])
        f_sw.write("};\n\n\n")

    f_sw.close()
    f_hw.close()

```

Listing C.1: Script for model parameter saving into C++ arrays

Γ′.2 Mapping τιμών συνάρτησης tanh

```
import numpy as np

if __name__ == "__main__":

    word_size = int(input("Give word size : "))
    int_size = int(input("Give integer part size : "))

    frac_size = word_size - int_size

    min_val = -2**(word_size-1-frac_size)
    max_val = (2**(word_size-1)-1)*(2**(-frac_size))
    q = 2**(-frac_size)

    input_vals = np.arange(min_val, max_val + q, q)
    output_vals = np.tanh(input_vals)

    f = open(f'tanh_{word_size-int_size}.h', 'w')

    f.write(f"// Mapping tanh values from {min_val} to {max_val} with a step of {q}\n\n")

    length = len(input_vals)
    f.write(f"quantized_type tanh_vals[{length}] = ")
    f.write("{ ")

    for i in range(0,length-1):
        f.write(f"{output_vals[i]}, ")

    f.write(f"{output_vals[length-1]}")
    f.write(" };")

```

Listing C.2: Script used for tanh values mapping into C++ array

Βιβλιογραφία

[1] Geron Aurelien, *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems.* Beijing: OReilly, 2019.

[2] *"Artificial neural network,"* [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network

[3] S. S. Haykin, *Neural networks and learning machines.* Delhi: Pearson, 2016.

[4] A. Karpathy, *"Convolutional neural networks for visual recognition"*, [Online]. Available: http://cs231n.github.io/neural-networks-1/

[5] Aston Zhang and Zachary C. Lipton and Mu Li and Alexander J. Smola, *Dive into Deep Learning.* 2020. [Online]. Available: https://d2l.ai

[6] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, *Generative Adversarial Networks,* arXiv:1406.2661

[7] The MNIST Dataset Of Handwritten Digits, Yann LeCun, Corinna Cortes, Christopher J.C. Burges. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[8] Kuhlman, Dave. "A Python Book: Beginning Python, Advanced Python, and Python Exercises". [Online]. Available: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html#part-1-beginning-python

[9] *"Python (programming language)"* [Online]. Available: https://en.wikipedia.org/wiki/Python_(programming_language)

[10] "Why TensorFlow". [Online]. Available: https://www.tensorflow.org/about

[11] *"TensorFlow"* [Online]. Available: https://en.wikipedia.org/wiki/TensorFlow

[12] "About Keras". [Online]. Available: https://keras.io/about/

[13] *"Keras"* [Online]. Available: https://en.wikipedia.org/wiki/Keras

[14] "Jupyter Project Documentation". [Online]. Available: https://jupyter.readthedocs.io/en/latest/

[15] *"Project Jupyter"* [Online]. Available: https://en.wikipedia.org/wiki/Project_Jupyter

[16] "Colaboratory FAQ". [Online]. Available: https://research.google.com/colaboratory/faq.html

[17] University of Miskolc, *"Field Programmable Gate Arrays (FPGA),"* [Online]. Available: http://mazsola.iit.uni-miskolc.hu/cae/docs/pld1.en.html.

[18] *"Field-programmable gate array,"* [Online]. Available: https://en.wikipedia.org/wiki/Field-programmable_gate_array

[19] Zybo Reference. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/zybo/start?redirect=1

[20] Zynq-7000 SoC Data Sheet. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[21] Zynq-7000 SoC. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[22] High-level synthesis. [Online]. Available: https://en.wikipedia.org/wiki/High-level_synthesis

[23] Vivado High-Level Synthesis. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[24] SDSoC Development Environment. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html

[25] Vivado Design Suite User Guide: High-Level Synthesis (UG902). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf

[26] SDSoC Programmers Guide (UG1278). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1278-sdsoc-programmers-guide.pdf

[27] Vivado HLS Optimization Methodology Guide (UG1270). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf

[28] 7 Series FPGAs Configurable Logic Block (UG474). [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf