ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Διαχείριση Πόρων σε Ετερογενείς Αρχιτεκτονικές με Εφαρμογή στο TensorFlow

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Α. Σοφιανίδης

Επιβλέπουσα : Θεοδώρα Βαρβαρίγου
Καθηγήτρια Ε.Μ.Π.

Αθήνα, Νοέμβριος 2019

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Διαχείριση Πόρων σε Ετερογενείς Αρχιτεκτονικές με Εφαρμογή στο TensorFlow

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Α. Σοφιανίδης

Επιβλέπουσα : Θεοδώρα Βαρβαρίγου
Καθηγήτρια Ε.Μ.Π.

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

.....	.....	.....
Θεοδώρα Βαρβαρίγου	Εμμανουήλ Βαρβαρίγος	Συμεών Παπαβασιλείου
Καθηγήτρια Ε.Μ.Π	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2019

.....

Γεώργιος Α. Σοφιανίδης

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

Περίληψη

Στη σύγχρονη εποχή της κατακόρυφης ανάπτυξης του υλικού των υπολογιστών (Computer Hardware) και της Υπολογιστικής Νέφους (Cloud Computing), παρατηρείται μία μεγάλη έκρηξη στην προσφορά υπολογιστικών πόρων από τους παρόχους (Cloud Service Providers) προς το ευρύ κοινό. Οι χρήστες δελεάζονται και επωφελούνται από αυτούς τους πόρους, καθώς μπορούν να εκτελέσουν τις εργασίες τους στο νέφος χωρίς να χρειάζεται να ξοδέψουν αρκετά χρήματα για την αγορά ισχυρών υπολογιστικών πόρων. Από την άλλη, οι πάροχοι επωφελούνται από την διάθεση των πόρων τους, καθώς χρεώνουν τους χρήστες ανάλογα με το ποσοστό χρήσης.

Ταυτόχρονα, η άνθιση της μηχανικής μάθησης και συγκεκριμένα των νευρωνικών δικτύων έχει οδηγήσει τους χρήστες να στρέφονται στην Υπολογιστική Νέφους για την εκπαίδευση των μοντέλων τους καθώς αυτά αυξάνονται σε πολυπλοκότητα και κατά συνέπεια απαιτούν ισχυρότερα μηχανήματα. Υπάρχει μία επιτακτική ανάγκη για τη διαχείριση των υπολογιστικών πόρων έτσι ώστε οι χρήστες να εκμεταλλεύονται πλήρως τα μηχανήματα που επινοικιάζουν. Από την άλλη, και οι πάροχοι επιδιώκουν την αποτελεσματικότερη κατανομή των πόρων τους έτσι ώστε να μειώνουν το κόστος λειτουργίας και να μεγιστοποιούν την αναλογία χρήσης και λειτουργίας.

Στόχος της παρούσας διπλωματικής εργασίας είναι η παρουσίαση τρόπων με τους οποίους μπορούμε να επιλέξουμε τους κατάλληλους πόρους από μία δεξαμενή διαθέσιμων μηχανημάτων για την εκτέλεση μίας εφαρμογής, με έμφαση στις εφαρμογές νευρωνικών δικτύων. Για το σκοπό αυτό χρησιμοποιούμε την πλατφόρμα TensorFlow, όπου παρουσιάζουμε παράλληλα τους τρόπους με τους οποίους η εργασία μας μπορεί να κατανεμηθεί σε ένα σύνολο από πόρους με ετερογενείς αρχιτεκτονικές (GPUs, CPUs κ.α.).

Λέξεις Κλειδιά: Κατανομή Πόρων, Διαχείριση Μηχανημάτων, Υπολογιστική Νέφους, Βελτιστοποίηση, Μηχανική Μάθηση, Νευρωνικά Δίκτυα, Προγραμματισμός Εργασιών, TensorFlow

Abstract

Nowadays, the rise of computer hardware and cloud computing has driven Cloud Service Providers to offer a wide variety of computer resources to the public. Users are attracted and are benefited by these resources as they can execute their work in the cloud without the need to buy powerful computer resources. On the other hand, providers are benefiting by charging users for service access and resource rental.

Simultaneously, the rise of machine learning and especially neural networks has driven users to Cloud Computing for their model's training as they rise on complexity and demand more powerful machines. There is need to manage those resources so that users use 100% of the services they rent. On the other hand, providers aim at better resource allocation so they can reduce power consumption and maximize the analogy between use and service.

The aim of this diploma thesis is to introduce ways to choose suitable resources from a pool of available machines to execute an application, with emphasis to neural network applications. That is why we use the TensorFlow platform, while we introduce ways by which our work can be distributed to a set of available resources with heterogenous architectures (GPUs, CPUs, etc.)

Keywords: Resource Allocation, Manage Resources, Cloud Computing, Optimization, Machine Learning, Neural Networks, Task Scheduling, Tensorflow

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω την επιβλέπουσα της εργασίας μου, Καθηγήτρια της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, κα. Θεοδώρα Βαρβαρίγου, για την ανάθεση ενός τόσο ενδιαφέροντος θέματος, αλλά και για την πολύτιμη βοήθεια και καθοδήγηση που μου προσέφερε κατά την εκπόνησή του.

Επίσης θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα Βρεττό Μουλό, για την άριστη συνεργασία μας και τις γνώσεις και συμβουλές που μου παρείχε καθ' όλα τα στάδια της εργασίας.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου και τους φίλους μου για την συμπαράσταση και υποστήριξη τους καθ' όλη τη διάρκεια των σπουδών μου.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΕΦΑΛΑΙΟ 1:
1.ΕΙΣΑΓΩΓΗ

1.1 Συνοπτική Παρουσίαση

Στη σύγχρονη εποχή της κατακόρυφης ανάπτυξης του υλικού των υπολογιστών (Computer Hardware) και της Υπολογιστικής Νέφους (Cloud Computing), παρατηρείται μία μεγάλη έκρηξη στην προσφορά υπολογιστικών πόρων από τους παρόχους (Cloud Service Providers) προς το ευρύ κοινό. Οι χρήστες δελεάζονται και επωφελούνται από αυτούς τους πόρους καθώς μπορούν να εκτελέσουν τις εργασίες τους στο νέφος χωρίς να χρειάζεται να ξοδέψουν αρκετά χρήματα για την αγορά ισχυρών υπολογιστικών πόρων. Από την άλλη, οι πάροχοι επωφελούνται από την διάθεση των πόρων τους, καθώς χρεώνουν τους χρήστες ανάλογα με το ποσοστό χρήσης.

Ταυτόχρονα, η άνθιση της μηχανικής μάθησης και συγκεκριμένα των νευρωνικών δικτύων έχει οδηγήσει τους χρήστες να στρέφονται στην Υπολογιστική Νέφους για την εκπαίδευση των μοντέλων τους καθώς αυτά αυξάνονται σε πολυπλοκότητα και κατά συνέπεια απαιτούν ισχυρότερα μηχανήματα. Υπάρχει μία επιτακτική ανάγκη για τη διαχείριση των υπολογιστικών πόρων έτσι ώστε οι χρήστες να εκμεταλλεύονται πλήρως τα μηχανήματα που επινοικιάζουν. Από την άλλη, και οι πάροχοι επιδιώκουν την αποτελεσματικότερη κατανομή των πόρων τους έτσι ώστε να μειώνουν το κόστος λειτουργίας και να μεγιστοποιούν την αναλογία χρήσης και λειτουργίας.

Στόχος της παρούσας διπλωματικής εργασίας είναι η παρουσίαση τρόπων με τους οποίους μπορούμε να επιλέξουμε τους κατάλληλους πόρους από μία δεξαμενή διαθέσιμων μηχανημάτων για την εκτέλεση μίας εφαρμογής, με έμφαση στις εφαρμογές νευρωνικών δικτύων. Για το σκοπό αυτό χρησιμοποιούμε την πλατφόρμα TensorFlow, όπου παρουσιάζουμε παράλληλα τους τρόπους με τους οποίους η εργασία μας μπορεί να κατανεμηθεί σε ένα σύνολο από πόρους με ετερογενείς αρχιτεκτονικές (GPUs, CPUs κ.α.).

1.2 Δομή της Διπλωματικής Εργασίας

Η διπλωματική εργασία στην γενική της εικόνα, χωρίζεται λογικά σε δύο ενότητες: στο θεωρητικό υπόβαθρο που είναι απαραίτητο να θεμελιωθεί για την μετέπειτα παρουσίαση των μοντέλων και στην προτεινόμενη προσέγγιση του προβλήματος και την τελική υλοποίηση της. Τα κεφάλαια της διπλωματικής ταξινομούνται στις λογικές αυτές ενότητες ως εξής:

- Θεωρητικό υπόβαθρο: Κεφάλαια 2,3,4,5

- Προσέγγιση προβλήματος και υλοποίηση: Κεφάλαια 6,7

Συγκεκριμένα, το περιεχόμενο αυτών των κεφαλαίων είναι το εξής:

- **Κεφάλαιο 2**
  Το κεφάλαιο 2 αποτελεί μία εισαγωγή στις σύγχρονες έννοιες των μεγάλων δεδομένων (big data) και της υπολογιστικής νέφους (cloud computing).

Παρουσιάζονται τα χαρακτηριστικά τους και οι τρόποι με τους οποίους οι πάροχοι προσφέρουν τις υπηρεσίες τους.

- **Κεφάλαιο 3**
  Στο κεφάλαιο αυτό γίνεται μία παρουσίαση των πόρων που προσφέρονται όπως Κεντρικές Επεξεργαστικές Μονάδες Γενικού Σκοπού (general use CPUs), Μονάδες Επεξεργασίας Γραφικών (Graphics Processing Units - GPUs), Μονάδες Επεξεργασίας Τανυστών (Tensor Processing Units – TPUs), Συστοιχίες Επιτόπια Προγραμματιζόμενων Πυλών (Field Programmable Gate Array – FPGAs) και αναφέρονται τα πλεονεκτήματα χρήσης της κάθε αρχιτεκτονικής.

- **Κεφάλαιο 4**
  Στο Κεφάλαιο 4 γίνεται μία εκτενής παρουσίαση του τομέα της μηχανικής μάθησης (machine learning) και των νευρωνικών δικτύων (neural networks) και αναλύεται το μαθηματικό υπόβαθρο για την κατανόηση τους.

- **Κεφάλαιο 5**
  Στο Κεφάλαιο 5 παρουσιάζεται η πλατφόρμα TensorFlow για την ανάπτυξη εφαρμογών μηχανικής μάθησης και αναλύονται οι τρόποι με τους οποίους τα μοντέλα μπορούν να κατανεμηθούν στους διαθέσιμους πόρους.

- **Κεφάλαιο 6**
  Στο κεφάλαιο αυτό παρουσιάζουμε τον αλγόριθμο για την αποδοτική κατανομή πόρων, εισάγουμε την έννοια του blueprint και εξηγούμε τη διαδικασία δημιουργίας TensorFlow clusters. Κατόπιν δοκιμάζουμε τον αλγόριθμο μας στην κατανομή του δημοφιλούς μοντέλου MNIST για την ταξινόμηση χειρόγραφων ψηφίων.

- **Κεφάλαιο 7**
  Στο τελευταίο κεφάλαιο αξιολογούμε τον αλγόριθμό μας και προτείνουμε τρόπους βελτίωσης του και τροφή για σκέψη.

ΚΕΦΑΛΑΙΟ 2:
2.ΜΕΓΑΛΑ ΔΕΔΟΜΕΝΑ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΗ ΝΕΦΟΥΣ

2.1 Μεγάλα Δεδομένα (Big Data)

Από την ανακάλυψη του παγκόσμιου ιστού το 1994 ο όγκος των δεδομένων έχει αυξηθεί τρομακτικά. Από τη χρήση των μηχανών αναζήτησης και των κοινωνικών δικτύων αλλά και γενικότερα από οποιαδήποτε δραστηριότητα στο διαδίκτυο, άνθρωποι χωρίς ιδιαίτερες γνώσεις προκαλούν την παραγωγή και αποθήκευση τεράστιων όγκων δεδομένων. Σε αυτά μπορούν να προστεθούν δεδομένα που προκύπτουν αυτόματα από αισθητήρες ενσωματωμένους σε συσκευές όπως έξυπνα κινητά, αυτοκίνητα κ.λπ. αλλά και δεδομένα που παράγονται καθημερινά από εικόνες δορυφόρων και επικοινωνιακά δίκτυα.

Αυτή η απίστευτη αύξηση της δημιουργίας δεδομένων σημαίνει ότι ο όγκος της πληροφορίας σε ένα κέντρο δεδομένων μπορεί πλέον να μετριέται σε terabytes (10^{12} bytes), petabytes (10^{15} bytes) ή ακόμα και σε exabytes (10^{18} bytes). Ο όρος *Μεγάλα Δεδομένα* ο οποίος μπήκε πλέον στη συνηθισμένη διάλεκτο μας, αναφέρεται σε τέτοιες ογκώδεις ποσότητες πληροφορίας. Το σύνολο αυτών των δεδομένων ξεπερνάει το τυπικό των παραδοσιακών συστημάτων διαχείρισης βάσεων δεδομένων για συλλογή, διαχείριση και ανάλυση αυτών.

2.1.1 Ορισμός

Ο όρος Μεγάλα Δεδομένα (Big Data) αναφέρεται στον τομέα ο οποίος ασχολείται με τρόπους ανάλυσης και συστηματικής εξόρυξης πληροφορίας, ή διαφορετικά με σύνολα δεδομένων τα οποία είναι πολύ μεγάλα ή πολύπλοκα για να αντιμετωπιστούν με παραδοσιακά λογισμικά επεξεργασίας δεδομένων.

Ένας πρώτος ορισμός αναφέρει τα Μεγάλα Δεδομένα ως το σύνολο της πληροφορίας που χαρακτηρίζεται από υψηλό όγκο, ταχύτητα και ποικιλία με αποτέλεσμα να απαιτείται ειδική τεχνολογία και αναλυτικές μέθοδοι για το μετασχηματισμό της σε τιμές. Ένας άλλος ορισμός παρουσιάζει τα Μεγάλα Δεδομένα ως σύνολα δεδομένων τα οποία χαρακτηρίζονται από τεράστια ποσά (όγκος) από συνεχώς συγχρονιζόμενα δεδομένα (ταχύτητα) σε διάφορες μορφές, όπως αριθμούς, κείμενο, ή εικόνες/videos (ποικιλία).

Στο σύγχρονο περιβάλλον, το μέγεθος των συνόλων δεδομένων που μπορούν να χαρακτηριστούν ως μεγάλα δεδομένα εκτείνεται από terabyte, ή petabytes ως exabytes. Στην πραγματικότητα η έννοια του τι είναι μεγάλα δεδομένα εξαρτάται από τη βιομηχανία, από το πως χρησιμοποιούνται τα δεδομένα, πόσο εμπλέκονται ιστορικά τα δεδομένα και πολλά άλλα χαρακτηριστικά.

2.1.2 Χαρακτηριστικά

Τα μεγάλα δεδομένα έχουν ένα μεγάλο πλήθος χαρακτηριστικών τα οποία παρουσίασε το 2011 το Gartner Group, μία δημοφιλής επιχειρηματικού επιπέδου οργάνωση

στην οποία απευθύνεται η βιομηχανία για να μάθει τις τάσεις. Σύμφωνα με αυτή τα μεγάλα δεδομένα χαρακτηρίζονται από τρία V: volume (όγκος), velocity (ταχύτητα), και variety (ποικιλία). Ορισμένοι ερευνητές προσθέτουν σε αυτά άλλα δύο V: veracity (αξιοπιστία) και value (αξία). Παρακάτω αναπτύσσουμε αυτά τα χαρακτηριστικά:

- *Όγκος (Volume)*
  Ο όγκος των δεδομένων προφανώς αναφέρεται στο μέγεθος των δεδομένων που μπορεί να διαχειριστεί ένα σύστημα. Δεδομένα που κατά κύριο λόγο δημιουργούνται αυτόματα τείνουν να έχουν τεράστιο όγκο. Παραδείγματα περιλαμβάνουν δεδομένα από αισθητήρες, όπως δεδομένα παραγωγής ή επεξεργασίας σε εργοστάσια, δεδομένα από συσκευές σάρωσης, όπως έξυπνες κάρτες και αναγνώστες πιστωτικών καρτών, και συσκευές μέτρησης, όπως έξυπνοι μετρητές ή συσκευές καταγραφής περιβαλλοντικών δεδομένων. Ταυτόχρονα τεράστιο όγκο δεδομένων αποθηκεύουν κατά κύριο λόγο τα κοινωνικά δίκτυα όπως το Twitter και το Facebook αλλά και άλλες πλατφόρμες αποθήκευσης βίντεο όπως το Youtube. Χαρακτηριστικά αναφέρουμε ότι το έτος 2010 οι επιχειρήσεις αποθήκευσαν 13 exabytes (10^{18} bytes) δεδομένων.

- *Ταχύτητα (Velocity)*
  Αναφέρεται στην ταχύτητα που τα δεδομένα δημιουργούνται, συλλέγονται, τροφοδοτούνται και επεξεργάζονται. . Τα Μεγάλα Δεδομένα είναι συνήθως διαθέσιμα σε πραγματικό χρόνο και παράγονται συνεχόμενα σε αντίθεση με τις μικρές ποσότητες δεδομένων. Τα δύο είδη ταχύτητας που σχετίζονται με τα Μεγάλα Δεδομένα είναι η συχνότητα παραγωγής και η συχνότητα χειρισμού, καταγραφής και δημοσίευσης.

- *Ποικιλία (Variety)*
  Ο τύπος και η φύση των δεδομένων. Ως παραδείγματα τύπων μπορούμε να δώσουμε τα δεδομένα διαδικτύου (πχ. κοινωνικά δίκτυα και επιλογές στο διαδίκτυο), τα ερευνητικά δεδομένα (πχ. βιομηχανικές αναφορές), τα δεδομένα θέσης (πχ. δεδομένα κινητών συσκευών), τις εικόνες, τα βίντεο, τα e-mails κλπ. Τα μεγάλα δεδομένα περιλαμβάνουν δομημένα, ημιδομημένα και αδόμητα δεδομένα σε διαφορετικά ποσοστά με βάση το περιεχόμενο. Τα δομημένα δεδομένα χαρακτηρίζονται από ένα τυπικό μοντέλο όπως το σχεσιακό, στο οποίο τα δεδομένα είναι σε μορφή πινάκων που περιέχουν γραμμές και στήλες. Τα αδόμητα δεδομένα αντίθετα δεν χαρακτηρίζονται από κάποιο τυπικό μοντέλο αλλά μπορεί να ταιριάζουν σε μία μορφοποίηση που επιτρέπει καλά ορισμένες ετικέτες που διαχωρίζουν σημασιολογικά στοιχεία.

- *Αξιοπιστία (Veracity)*
  Πρόκειται για ένα πιο σύγχρονο χαρακτηριστικό το οποίο αναφέρεται στην ποιότητα και την καταλληλότητα των δεδομένων όσον αφορά το ακροατήριο στο οποίο απευθύνονται. Για παράδειγμα εάν θέλουμε να αναλύσουμε κάποια δεδομένα καλό θα ήταν πρώτα να περάσουν από κάποιον βαθμό ελέγχου ποιότητας και ανάλυσης αξιοπιστίας. Πολλές πηγές δεδομένων δημιουργούν δεδομένα που είναι αβέβαια, δεν είναι πλήρη, είναι ανακριβή και επομένως η αξιοπιστία τους είναι αμφισβητήσιμη.

- *Αξία (Value)*
  Η εμπορική αξία των δεδομένων που συλλέγονται. Για παράδειγμα πολλές εταιρείες δημιουργούν τις δικές τους πλατφόρμες δεδομένων με σκοπό να αξιοποιούν τα δεδομένα προς όφελος τους.

2.2 Υπολογιστική Νέφους (Cloud Computing)

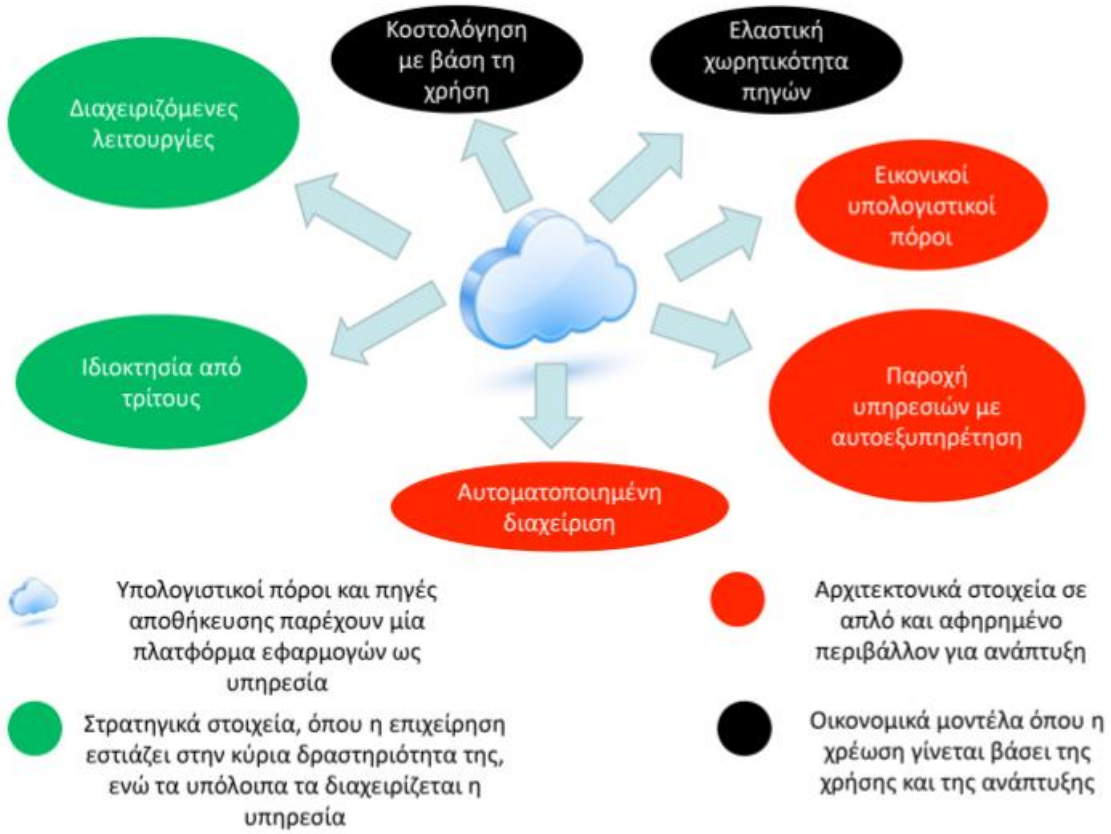
2.2.1 Εισαγωγή

Η υπολογιστική νέφους είναι ένα μοντέλο για τη διευκόλυνση της κατ' απαίτηση πρόσβασης μέσω δικτύου σε μία κοινόχρηστη δεξαμενή παραμετροποιήσιμων υπολογιστικών πόρων (π.χ. δίκτυα, εξυπηρετητές, περιοχές αποθήκευσης, εφαρμογές και υπηρεσίες), που μπορούν γρήγορα να εκχωρηθούν και να απελευθερωθούν με ελάχιστη διαχειριστική προσπάθεια ή αλληλεπίδραση με τον πάροχο των πόρων. Η υπολογιστική νέφους βασίζεται στη διαθεσιμότητα των πόρων, προσφέροντας οφέλη στους χρήστες του, είτε είναι οργανισμοί είτε είναι ιδιώτες. Ωστόσο, η προστασία της ιδιωτικότητας των δεδομένων και η ασφάλεια που προσφέρουν θέλουν αξιολόγηση. Γι' αυτό και απαιτείται προσοχή στους όρους χρήσης των προσφερόμενων στο νέφος υπηρεσιών.

Η σύνδεση του υπολογιστή ή της έξυπνης συσκευής του χρήστη στην πλατφόρμα της υπολογιστικής νέφους γίνεται με τη χρήση εξειδικευμένου λογισμικού. Στην υπολογιστική νέφους, η επεξεργαστική ισχύς εξασφαλίζεται από μεγάλα κέντρα δεδομένων, με εκατοντάδες ή και χιλιάδες εξυπηρετητές και συστήματα αποθήκευσης δεδομένων, που μπορούν στην πράξη να χειριστούν σχεδόν οποιοδήποτε λογισμικό υπολογιστή. Υπάρχει μια σειρά υπηρεσιών που μπορεί να προσφέρονται (π.χ. διαδικτυακό ηλεκτρονικό ταχυδρομείο), ενώ για όσες υπηρεσίες απαιτείται πληρωμή, οι χρήστες μπορούν να διαλέξουν από μια πληθώρα διαφορετικών τρόπων πληρωμής αυτόν που τους διευκολύνει.

Η υπολογιστική νέφους παρέχεται και ως κατ' απαίτηση αυτοεξυπηρετούμενη υπηρεσία (on demand self service), δηλαδή οι πελάτες της, οι οργανισμοί και οι επιχειρήσεις μπορούν να αυτοδιαχειρίζονται τους δικούς τους υπολογιστικούς πόρους. Απαλλάσσει τους χρήστες από την αγορά εξειδικευμένου λογισμικού ή την αγορά και συντήρηση ακριβών και δύσκολων στην παραμετροποίηση εξυπηρετητών. Αντίστοιχα, τους προσφέρει αποθήκευση δεδομένων χωρίς τη χρήση δικού τους υλικού ή λογισμικού, δίνοντάς τους ευελιξία στην αποθήκευση δεδομένων. Με αυτούς τους τρόπους, οι χρήστες εξοικονομούν χρήματα και εκτελούν τις εργασίες τους πιο αποτελεσματικά και αξιόπιστα.

Οι υπηρεσίες της υπολογιστικής νέφους μπορεί να προσφερθούν μέσω διαδικτύου ή μέσω ιδιωτικών δικτύων. Οι πελάτες τραβούν πόρους μέσα από την αντίστοιχη δεξαμενή πόρων, η οποία βρίσκεται σε απομακρυσμένα κέντρα δεδομένων. Είναι προφανές ότι οι υπηρεσίες της υπολογιστικής νέφους θα πρέπει να μπορούν να κλιμακώνονται ανάλογα με τη ζήτηση των πόρων. Εν ολίγοις, θα πρέπει να είναι διαθέσιμες, ακέραιες, ασφαλείς και ανταποκρίσιμες. Τέλος, η χρήση τους θα πρέπει να είναι μετρήσιμη, ώστε οι πελάτες να χρεώνονται ανάλογα με τη χρήση των υπηρεσιών και των πόρων.



Εικόνα 1: Συστατικά της Υπολογιστικής Νέφους

2.2.2 Μοντέλα της Υπολογιστικής Νέφους

Τα κύρια μοντέλα υπηρεσιών υπολογιστικής νέφους είναι τα εξής:

1. *Το Λογισμικό ως Υπηρεσία (Software as a Service/SaaS)*, κατά το οποίο παρέχεται μια προκατασκευασμένη εφαρμογή, μαζί με το απαιτούμενο λογισμικό, λειτουργικό σύστημα, υλικό και δίκτυο. Είναι ένα μοντέλο διανομής λογισμικού στο οποίο οι εφαρμογές φιλοξενούνται από έναν πάροχο υπηρεσιών ή προμηθευτή και διατίθενται στους πελάτες μέσω ενός δικτύου, συνήθως μέσω του διαδικτύου. Παραδείγματα αποτελούν τα Google Apps, Salesforce κ.ά. Στα οφέλη του περιλαμβάνονται:
    - ✓ η ευκολότερη διαχείριση,
    - ✓ οι αυτόματες ενημερώσεις και η διαχείριση επιδιορθώσεων (patch management),
    - ✓ η συμβατότητα, καθώς όλοι οι χρήστες θα έχουν την ίδια έκδοση του λογισμικού,
    - ✓ η ευκολότερη συνεργασία και
    - ✓ η παγκόσμια προσβασιμότητα.

2. *Η Πλατφόρμα ως Υπηρεσία (Platform as a Service/PaaS)*, κατά την οποία παρέχονται το λειτουργικό σύστημα, το υλικό και το δίκτυο, ενώ ο πελάτης εγκαθιστά ή αναπτύσσει το δικό του λογισμικό εφαρμογών. Είναι ένα μοντέλο που

παρέχει εφαρμογές μέσω του διαδικτύου. Σε αυτό, ένας πάροχος υπολογιστικού νέφους παραδίδει το υλικό και τα εργαλεία λογισμικού, συνήθως όσα είναι απαραίτητα για την ανάπτυξη εφαρμογών, στους χρήστες του ως υπηρεσία. Ένας πάροχος Πλατφόρμας ως Υπηρεσίας φιλοξενεί το υλικό και το λογισμικό στη δική του υποδομή. Ως αποτέλεσμα, το μοντέλο αυτό απαλλάσσει τους χρήστες από την εγκατάσταση υλικού στο δικό τους περιβάλλον και τους παρέχει λογισμικό ώστε να αναπτύξουν ή να εκτελέσουν μια νέα εφαρμογή. Παράδειγμα αποτελεί το Apprenda.

3. *Η Υποδομή ως Υπηρεσία (Infrastructure as a Service/IaaS)*, κατά την οποία παρέχονται μόνο το υλικό και το δίκτυο. Ο πελάτης εγκαθιστά ή αναπτύσσει δικά του λειτουργικά συστήματα, λογισμικό και εφαρμογές. Η Υποδομή ως Υπηρεσία είναι μια μορφή που παρέχει εικονικούς υπολογιστικούς πόρους μέσω του διαδικτύου. Σε αυτό, ένας πάροχος φιλοξενεί το υλικό, το λογισμικό, τους εξυπηρετητές, την αποθήκευση και άλλα στοιχεία υποδομής για λογαριασμό των χρηστών του. Οι πάροχοί της φιλοξενούν επίσης τις αιτήσεις των χρηστών και χειρίζονται αιτήματα, συμπεριλαμβανομένης της συντήρησης του συστήματος, της δημιουργίας αντιγράφων ασφαλείας και της ανθεκτικότητας του σχεδιασμού. Παραδείγματα αποτελούν τα Amazon Web Services (AWS), Microsoft Azure και Google Compute Engine (GCE).

2.2.3 Παροχή Υπηρεσιών Νέφους

Οι υπηρεσίες νέφους παρέχονται συνήθως με τις εξής μορφές: *ιδιωτικό* ή *εσωτερικό* (private), *κοινότητας* (community), *δημόσιο* (public) ή *υβριδικό* (hybrid).

Σε γενικές γραμμές, οι υπηρεσίες ενός δημόσιου νέφους προσφέρονται μέσω του διαδικτύου και η διαχείρισή τους γίνεται από συγκεκριμένο πάροχο (provider). Περιλαμβάνουν υπηρεσίες που απευθύνονται στο ευρύ κοινό (ή/και σε επιχειρήσεις, κατά περίπτωση), όπως αυτές της διαδικτυακής αποθήκευσης φωτογραφιών, του ηλεκτρονικού ταχυδρομείου, των δικτυακών τόπων κοινωνικής δικτύωσης κ.ά. Ένα από τα πλεονεκτήματα της χρήσης μιας δημόσιας υπηρεσίας νέφους είναι η εύκολη και ανέξοδη αρχικοποίηση, γιατί το κόστος του υλικού, της εφαρμογής και του εύρους ζώνης καλύπτεται από τον πάροχο. Άλλα πλεονεκτήματα αποτελούν η επεκτασιμότητα, για την κάλυψη των αναγκών, και η μηδενική σπατάλη πόρων, επειδή ο χρήστης πληρώνει μόνο για ό,τι χρησιμοποιεί.

Σε ένα *ιδιωτικό νέφος*, η υποδομή λειτουργεί αποκλειστικά για συγκεκριμένη επιχείρηση ή συγκεκριμένο οργανισμό, και συνήθως η διαχείρισή της γίνεται από αυτόν/αυτήν ή από ένα τρίτο έμπιστο μέρος (third trusted party). Μπορεί να ιδωθεί ως μια αρχιτεκτονική υπολογιστών η οποία βρίσκεται πίσω από κάποιο τείχος προστασίας. Η προώθηση αυτού του μοντέλου έχει σχεδιαστεί για να προσελκύσει έναν οργανισμό ο οποίος θέλει περισσότερο έλεγχο των δεδομένων του από αυτόν που επιτυγχάνεται με τη χρήση ενός πακέτου υπηρεσιών φιλοξενίας κάποιου παρόχου.

Σε ένα *νέφος κοινότητας*, η υπηρεσία είναι κοινόχρηστη από διαφόρους οργανισμούς και διατίθενται μόνο στις ομάδες της κοινότητας. Η διαχείριση και λειτουργία της υποδομής μπορεί να γίνεται από τους ίδιους τους οργανισμούς ή από έναν πάροχο υπηρεσιών νέφους. Στόχος του είναι να δώσει στους συμμετέχοντες οργανισμούς τα οφέλη ενός δημόσιου νέφους, όπως πολυμίσθωση και δομή τιμολόγησης πληρωμής σύμφωνα με τη χρήση (pay-as-you-go), αλλά με την προσθήκη της προστασίας δεδομένων, της ασφάλειας και της συμμόρφωσης της πολιτικής, που σχετίζονται συνήθως με ένα ιδιωτικό νέφος.

Σε ένα *δημόσιο* ή *υβριδικό νέφος*, που είναι συνδυασμός των παραπάνω, ένας οργανισμός παρέχει και διαχειρίζεται κάποιους πόρους στο εσωτερικό του, αλλά διαθέτει και άλλους, που παρέχονται εξωτερικά. Για παράδειγμα, ένας οργανισμός μπορεί να χρησιμοποιήσει μια δημόσια υπηρεσία νέφους, όπως το Amazon Simple Storage Service (Amazon S3), για αρχειοθετημένα δεδομένα, αλλά να εξακολουθεί να διατηρεί τα επιχειρησιακά δεδομένα των πελατών στο εσωτερικό του.

2.2.4 Πλεονεκτήματα, Μειονεκτήματα και Προοπτικές των Υπηρεσιών Νέφους

Οι υπηρεσίες υπολογιστικής νέφους μπορούν να μειώσουν το κόστος και την πολυπλοκότητα λειτουργίας της υποδομής πληροφορικής (IT infrastructure) μιας επιχείρησης ή ενός οργανισμού. Συνήθως, οι χρήστες των υπηρεσιών αυτών δεν θέλουν ή δεν μπορούν να επενδύσουν στις τεχνολογίες πληροφορικής, αγοράζοντας υλικό ή άδειες χρήσης λογισμικού.

Με αυτήν τη μορφή εξωτερικής ανάθεσης (outsourcing), οι επιχειρήσεις ανακτούν τα οφέλη από τα χρήματα που δίνουν σχεδόν άμεσα, ενώ ταυτόχρονα αποκτούν προσαρμοστικότητα, ευελιξία και ευκολία στην ανάπτυξη και διάθεση λύσεων, δηλαδή καθιστούν εφικτό αυτό που ένας οργανισμός ή μια επιχείρηση από μόνοι τους θα αδυνατούσαν να επιτύχουν.

Οι υπηρεσίες υπολογιστικής νέφους κλιμακώνονται εύκολα, προσφέροντας ένα συνεχώς αυξανόμενο αριθμό πόρων, ενώ επιτρέπουν την πρόσβαση σε εφαρμογές και αρχεία από οπουδήποτε μέσω του διαδικτύου. Βέβαια, αυτές οι υπηρεσίες θα πρέπει να παρέχονται όσο το δυνατόν πιο αξιόπιστα. Έτσι, δίνεται η δυνατότητα στους οργανισμούς να απεγκλωβιστούν από την ανάγκη για πόρους και να εστιαστούν στην καινοτομία και στην ανάπτυξη του δικού τους προϊόντος και της δικής τους επιχειρηματικής δράσης. Και όλα αυτά μαζί με τη δυνατότητα για διαχείριση της υποδομής και των υπηρεσιών με καλύτερο τρόπο από τους ίδιους τους φορείς ή από τον πάροχο, ανάλογα με την περίπτωση.

Οι πάροχοι υπηρεσιών νέφους (Cloud Service Providers – CSPs) τείνουν πλέον να χρησιμοποιούν την τεχνική της εικονοποίησης (virtualization) που επιτρέπει σε πολλαπλά λειτουργικά συστήματα να τρέχουν στην ίδια φυσική πλατφόρμα, και δομεί τους εξυπηρετητές (servers) σε εικονικά μηχανήματα (virtual machines – VMs). Τα εικονικά μηχανήματα προσφέρουν με τη σειρά τους στους χρήστες υποδομές, πλατφόρμες και πόρους όπως για παράδειγμα επεξεργαστική ισχύ, μνήμη, χώρο αποθήκευσης κλπ. Το κίνητρο των παρόχων είναι τα οφέλη από τη χρέωση των χρηστών για τις υπηρεσίες νέφους ενώ το κίνητρο των χρηστών είναι εξάλειψη των δαπανών που θα πρόκυπταν από υπολογισμούς, χρόνο και κατανάλωση σε ρεύμα.

Τα τεράστια κόστη σε ενέργεια όσον αφορά το ρεύμα που καταναλώνουν τα κέντρα δεδομένων είναι μία σημαντική πρόκληση. Συγκεκριμένα, υπολογίζεται ότι η κατανάλωση ρεύματος στα κέντρα δεδομένων αναμένεται να φτάσει τα 140 δισεκατομμύρια κιλοβατώρες ετησίως μέχρι το 2020 που θα κοστίσει περίπου 13 δισεκατομμύρια δολάρια σε ετήσιους λογαριασμούς ρεύματος. Επομένως για να αυξηθούν τα περιθώρια κέρδους καθώς επίσης για να μειωθούν η εκπομπές διοξειδίου του άνθρακα στην ατμόσφαιρα, είναι επιτακτική η ανάγκη για την ελαχιστοποίηση της κατανάλωσης ηλεκτρικής ενέργειας στα μεγάλα κέντρα δεδομένων.

ΚΕΦΑΛΑΙΟ 3:
3.ΔΙΑΘΕΣΙΜΟΙ ΥΠΟΛΟΓΙΣΤΙΚΟΙ ΠΟΡΟΙ ΚΑΙ ΔΙΑΧΕΙΡΙΣΗ ΤΟΥΣ

3.1 Εισαγωγή

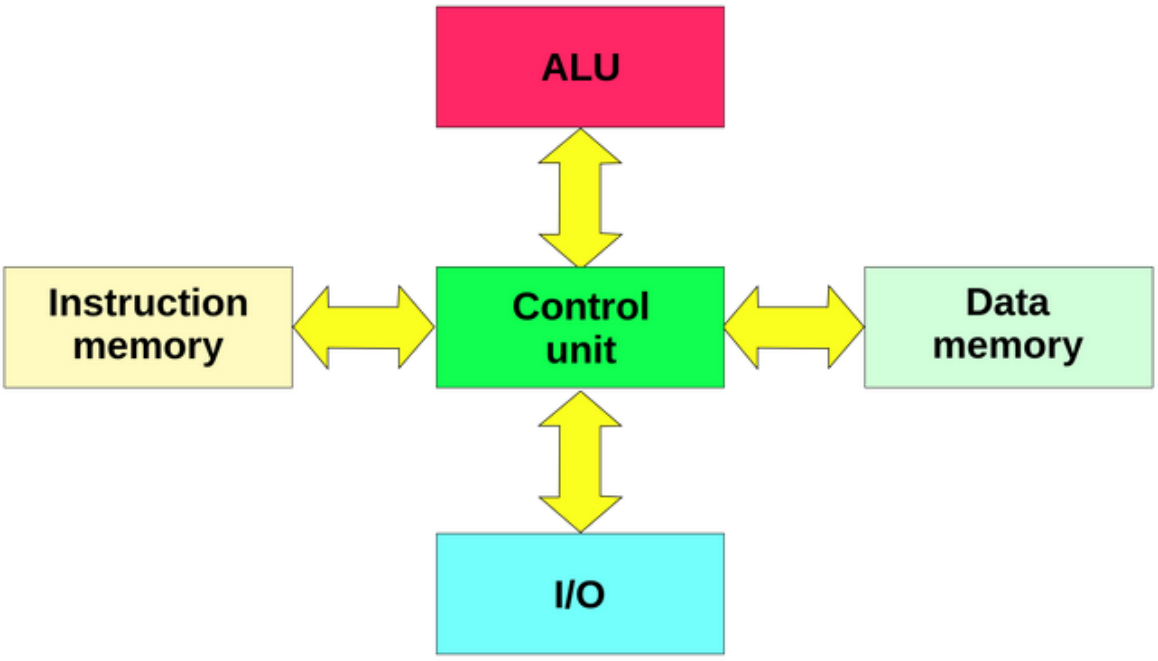
Η μεγάλη ανάπτυξη των Νευρωνικών Δικτύων έχει οδηγήσει τους παρόχους υπηρεσιών νέφους (cloud providers) να προσφέρουν μία πληθώρα από επιλογές όσον αφορά την υπολογιστική πλατφόρμα που θα χρησιμοποιηθεί για την εκπαίδευση μίας εφαρμογης μηχανικής μάθησης. Πάροχοι όπως η AWS, η Alibaba Cloud, η Azure και η Huawei προσφέρουν αρκετές πλατφόρμες όπως γενικού σκοπού κεντρικές μονάδες επεξεργασίας (Central Processing Unit - CPUs), CPUs βελτιστοποιημένες για υπολογισμούς, CPUs βελτιστοποιημένες στο κομμάτι της μνήμης, μονάδες επεξεργασίας γραφικών (Graphics Processing Units - GPUs), μονάδες επεξεργασίας τανυστών (Tensor Processing Units - TPUs) ακόμα και Συστοιχίες Επιτόπια Προγραμματιζόμενων Πυλών (Field Programmable Gate Array – FPGAs). Η επιλογή της κατάλληλης πλατφόρμας για να επιτύχουμε την καλύτερη απόδοση, το χαμηλότερο κόστος ή των συνδυασμό αυτών είναι μία πρόκληση που απαιτεί προσεκτική σκέψη και αναλυτικό σχεδιασμό από τον επιστήμονα δεδομένων.

3.2 Κεντρική Μονάδα Επεξεργασίας (Central Processing Unit - CPU)

Η κεντρική μονάδα επεξεργασίας (CPU) σχεδιάστηκε ως ένας γενικού σκοπού επεξεργαστής που μπορεί να εκτελεί κάθε είδος υπολογισμού βασισμένο σε συγκεκριμένες εντολές. Σε πολύ χαμηλό επίπεδο, η CPU εκτελεί πράξεις όπως ολισθήσεις, προσθέσεις, πολλαπλασιασμούς, αφαιρέσεις, καταχωρήσεις κ.λπ. Αυτές οι εντολές περιμένουν σε μία "ουρά" για εκτέλεση από την αριθμητική και λογική μονάδα (arithmetic and logic unit – ALU). Παρακάτω μπορούμε να δούμε τη βασική αρχιτεκτονική των CPUs που ονομάζεται αρχιτεκτονική Harvard (Εικόνα 2).

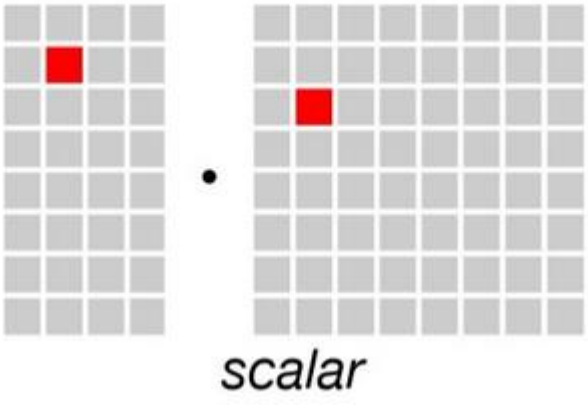
Η *είσοδος / έξοδος (Input / Output – I/O)* είναι το κομμάτι με το οποίο αλληλεπιδρούν οι χρήστες και περιλαμβάνει πληκτρολόγια, οθόνες και άλλες συσκευές εισόδου – εξόδου. Η *μνήμη εντολών (instruction memory)* είναι το μέρος στο οποίο στοιβάζονται οι εντολές που θέλουμε να τρέξουν στην CPU. Η *δομή ελέγχου (control unit)* ενορχηστρώνει την όλη διαδικασία και ορίζει ποια δομή θα τρέξει κάποια συγκεκριμένη δουλειά. Η *μνήμη δεδομένων (data memory)* είναι το μέρος στο οποίο αποθηκεύονται όλα τα απαιτούμενα δεδομένα για τους υπολογισμούς. Τέλος η *αριθμητική και λογική μονάδα (arithmetic and logic unit – ALU)* είναι το μέρος στο οποίο εκτελούνται προσθέσεις, αφαιρέσεις και άλλες λογικές πράξεις.

Η διαδικασία εκτέλεσης υπολογισμών αποτελείται από τα εξής βήματα. Αρχικά η δομή ελέγχου "φέρνει" (fetches) τις εντολές από τη μνήμη εντολών και κατόπιν τις αποκωδικοποιεί (decodes) και τέλος η αριθμητική και λογική μονάδα εκτελεί (executes) την παρούσα πράξη. Ο επεξεργαστής εκτελεί αυτή την αλυσίδα εντολών (fetch – decode - execute) την μία μετά την άλλη, έτσι ώστε να ολοκληρωθεί μία συγκεκριμένη εργασία.



Εικόνα 2: Αρχιτεκτονική Harvard

Οι επεξεργαστές εκτελούν τις εντολές σε κάθε έναν από τους πυρήνες τους σειριακά με ορισμένα νήματα κάθε φορά. Οι νεότεροι επεξεργαστές έχουν περιορισμένο αριθμό πυρήνων που μπορεί να φτάσει τους 40 και έχουν τη δυνατότητα να χειρίζονται δεκάδες εντολές σε κάθε κύκλο με πολύ μεγάλη ταχύτητα. Η μικρότερη μονάδα που μπορεί να χειριστεί ένας επεξεργαστής είναι τα βαθμωτά μεγέθη, δηλαδή 1x1 στοιχεία όπως φαίνεται στην παρακάτω εικόνα:



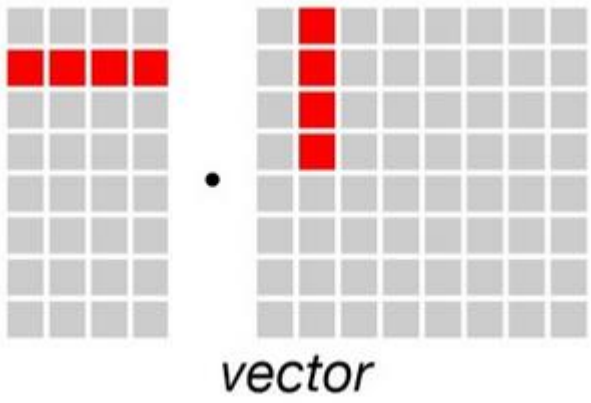
Εικόνα 3: Μικρότερος τύπος δεδομένων στη CPU

3.3 Μονάδα Επεξεργασίας Γραφικών (Graphics Processing Unit - GPU)

Η μονάδα επεξεργασίας γραφικών (GPU) αποτελεί ένα μικροεπεξεργαστικό chip το οποίο έχει σχεδιαστεί για να χειρίζεται γραφικά σε υπολογιστικά περιβάλλοντα. Μπορεί να είναι εφοδιασμένη με εκατοντάδες ή ακόμα και χιλιάδες πυρήνες σε αντίθεση με τη CPU αλλά οι πυρήνες αυτοί "τρέχουν" σε μικρότερες ταχύτητες. Παραδοσιακά οι GPUs χρησιμοποιούνται για την επεξεργασία 3D περιεχομένου όπως παιχνίδια, αλλά τον τελευταίο καιρό έχουν γίνει αρκετά δημοφιλείς στη βιομηχανία, κυρίως γιατί μπορούν να χειριστούν εργασίες που περιλαμβάνουν ταυτόχρονους υπολογισμούς γρηγορότερα από τις CPUs, όπως για παράδειγμα προσομοιώσεις μοντέλων.

Οι GPUs θεωρούνται η βέλτιστη επιλογή για την εκπαίδευση μοντέλων μηχανικής μάθησης καθώς προσφέρονται για την ταυτόχρονη εκτέλεση πολλαπλών εντολών. Συγκεκριμένα, ο μεγάλος αριθμός πυρήνων επιτρέπει την καλύτερη εκτέλεση πολλαπλών παράλληλων εντολών. Επίσης οι υπολογισμοί σε μοντέλα μηχανικής μάθησης χρειάζεται να χειριστούν μεγάλες ποσότητες δεδομένων, καθιστώντας το υψηλό εύρος ζώνης μνήμης (memory bandwidth) των GPUs (που μπορεί να τρέξει μέχρι και 750 GB/s σε αντίθεση με 50 GB/s που προσφέρουν οι παραδοσιακές CPUs) το πλέον καταλληλότερο.

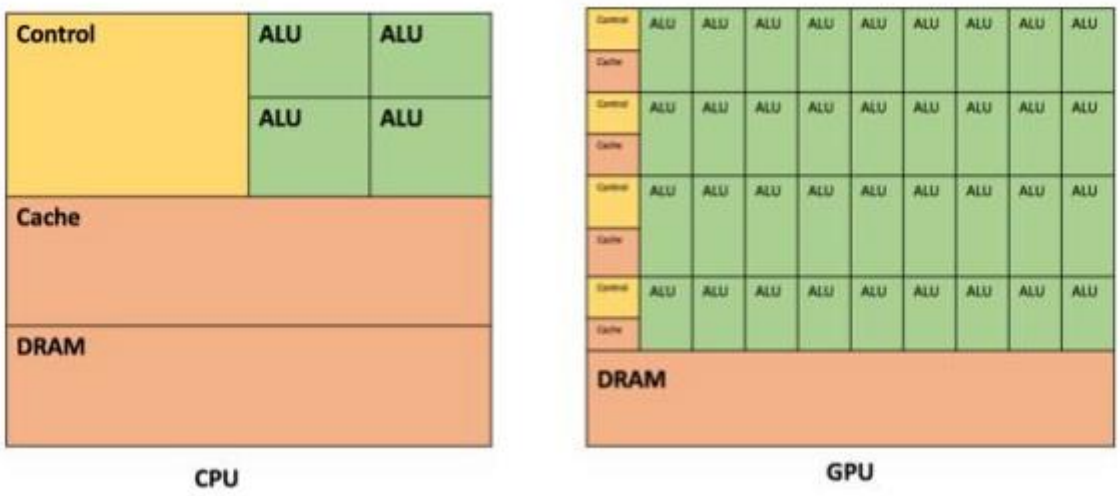
Οι GPUs μπορούν να χειριστούν δεκάδες χιλιάδες εντολές σε κάθε κύκλο συνήθως όμως με μικρότερη ταχύτητα. Η μικρότερη μονάδα που μπορεί να χειριστεί μία GPU είναι τα διανυσματικά μεγέθη δηλαδή 1xN στοιχεία όπως φαίνεται στην παρακάτω εικόνα:



Εικόνα 4: Μικρότερος τύπος δεδομένων στη GPU

Όσον αφορά τον τομέα της μηχανικής μάθησης συγκεκριμένα, το κάθε επίπεδο (layer) του νευρωνικού δικτύου και ειδικά ο κάθε νευρώνας του επιπέδου λαμβάνει πολλαπλές εισόδους από τους νευρώνες του προηγούμενου επιπέδου, στη συνέχεια τις πολλαπλασιάζει με τα βάρη και μετά τα προσθέτει μαζί. Εκτελεί στην ουσία τη διαδικασία που ονομάζεται Πολλαπλασιασμός και Συσσώρευση (Multiply and Accumulation – MAC). Επιπλέον υπάρχουν χιλιάδες τέτοιοι νευρώνες στο νευρωνικό δίκτυο που εκτελούν αυτήν ακριβώς τη διαδικασία. Η ιδιότητα των GPUs να είναι κατάλληλες για την εκτέλεση του MAC τις καθιστούν κατάλληλες για το χειρισμό πολλών παράλληλων εντολών και για το σκοπό αυτό ο φόρτος της μηχανικής μάθησης οδηγείται στις GPUs.

Παρακάτω φαίνεται επιπλέον η διαφοροποίηση των GPUs απο τις CPUs όσον αφορά των αριθμό των πυρήνων (cores):



Εικόνα 5: Διαφοροποίηση CPU και GPU στο κομμάτι των πυρήνων

3.4 Ανάλυση και Σύγκριση CPU και GPU σε εφαρμογές μηχανικής μάθησης

Όσον αφορά τις CPUs το μεγάλο τους πλεονέκτημα είναι η ευκολία στον προγραμματισμό σε οποιαδήποτε γλώσσα ή framework. Με τον τρόπο αυτό μπορεί κανείς να πραγματοποιήσει μία γρήγορη εξερεύνηση χώρου και να τρέξει τις εφαρμογές του. Στην περίπτωση της μηχανικής μάθησης, η εκπαίδευση είναι κατάλληλη για απλά μοντέλα που δεν απαιτούν χρόνο για να ολοκληρωθούν και για μικρά μοντέλα με μικρά αποτελεσματικά σύνολα (batch size). Στην περίπτωση των μεγάλων μοντέλων και των μεγάλων συνόλων δεδομένων ο συνολικός χρόνος εκτέλεσης για τη μηχανική μάθηση μπορεί να είναι πολλές φορές απαγορευτικά μεγάλος.

Στην αντίθετη πλευρά οι GPUs είναι ειδικές επεξεργαστικές μονάδες που σχεδιάστηκαν κυρίως για την επεξεργασία εικόνων και videos. Είναι βασισμένες σε απλούστερα επεξεργαστικά μοντέλα σε σύγκριση με τις CPUs αλλά μπορούν να φιλοξενήσουν έναν πολύ μεγαλύτερο αριθμό πυρήνων καθιστώντας τις κατάλληλες για εφαρμογές στις οποίες τα δεδομένα πρέπει να επεξεργάζονται παράλληλα όπως για παράδειγμα εικονοστοιχεία (pixels) από εικόνες ή videos. Ο προγραμματισμός τους όμως σε συγκεκριμένες γλώσσες όπως η CUDA και το OpenCL παρέχουν μικρότερη ευελιξία σε σύγκριση με τις CPUs.

3.4.1 Διαφορετικά Χαρακτηριστικά

- Οι CPUs έχουν λίγους πολύπλοκους πυρήνες που τρέχουν διεργασίες ακολουθιακά, με ορισμένα νήματα την κάθε φορά ενώ οι GPUs έχουν έναν μεγάλο αριθμό απλούστερων πυρήνων που επιτρέπουν την παράλληλη επεξεργασία σε χιλιάδες νήματα που εκτελούν ταυτόχρονα υπολογισμούς.
- Σε εφαρμογές μηχανικής μάθησης, ο κώδικας τρέχει στην CPU ενώ στην GPU τρέχει κώδικας CUDA.

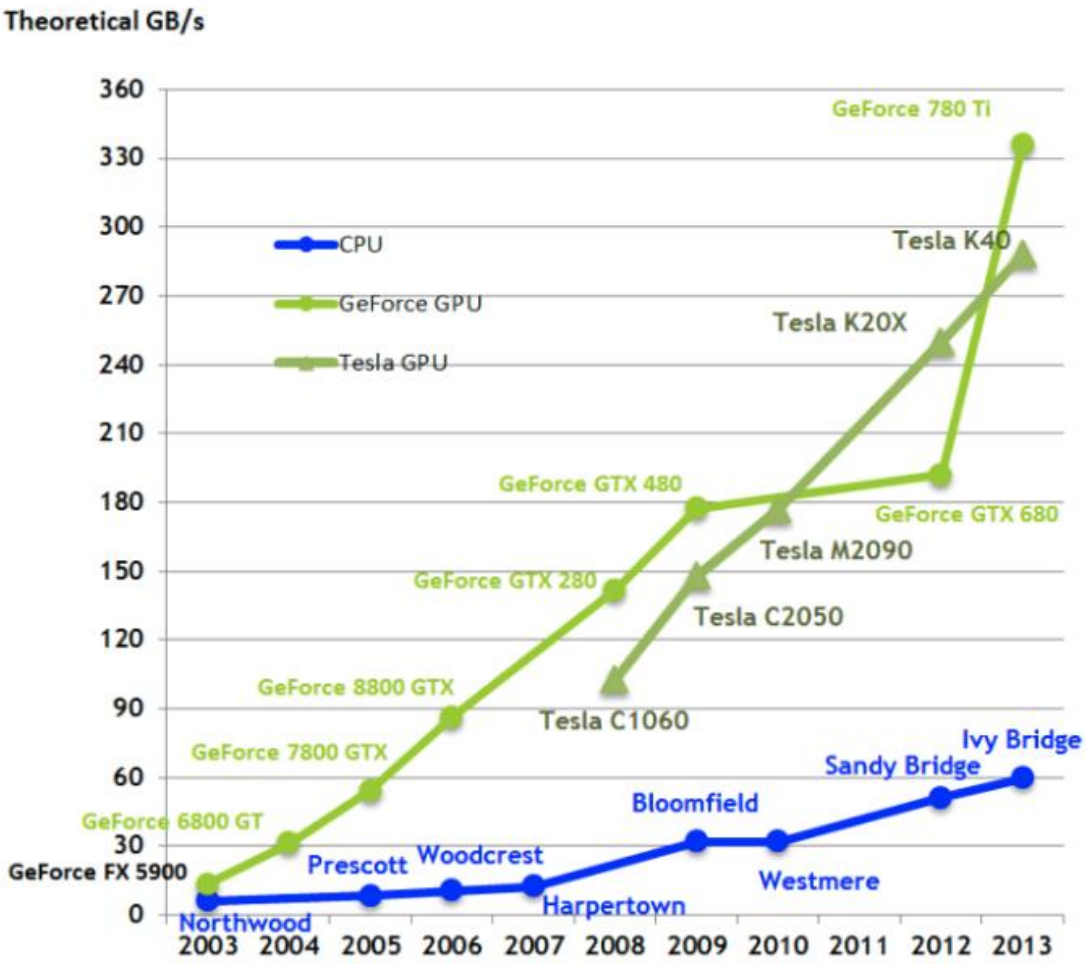
- Η CPU αναθέτει τις πολύπλοκες εργασίες όπως την απόδοση 3D γραφικών (3D Graphics Rendering) και τον υπολογισμό διανυσμάτων στην GPU.
- Ενώ η CPU μπορεί να χειριστεί βέλτιστα πολύπλοκες εργασίες, η GPU μπορεί να παρουσιάσει μία συμφόρηση εύρους ζώνης (bandwidth bottleneck issue) όπως για παράδειγμα σε μεταφορές μεγάλων ποσοτήτων δεδομένων όπου η GPU μπορεί να είναι αργή σε ταχύτητα.
- Οι GPUs βελτιστοποιούνται ως προς το εύρος ζώνης (bandwidth) ενώ οι CPUs ως προς το χρόνο πρόσβασης στη μνήμη (latency – memory access time).

3.4.2 Αναλογίες

Μπορεί κανείς να κάνει την εξής αναλογία παρομοιάζοντας την CPU ως ένα γρήγορο αυτοκίνητο και την GPU ως ένα τεράστιο φορτηγό που χρησιμοποιείται για τη μεταφορά αγαθών. Η CPU (το γρήγορο αυτοκίνητο) μπορεί να μεταφέρει μικρές ποσότητες πακέτων στην προσωρινή μνήμη (random access memory – RAM) ενώ από την άλλη πλευρά η GPU (φορτηγό) μπορεί να μεταφέρει τεράστιες ποσότητες μνήμης με μία διαδρομή αλλά με μικρότερη ταχύτητα. Ακολουθούν οι βασικές παράμετροι που μπορεί κανείς να λάβει υπόψιν για την επιλογή της συσκευής που θα εκπαιδεύσει το μοντέλο του.

- *Εύρος Ζώνης Μνήμης*

  Το εύρος ζώνης είναι ο κύριος λόγος για τον οποίο οι GPUs είναι γρηγορότερες στους υπολογισμούς από τις CPUs. Λόγω των μεγάλων συνόλων δεδομένων, η CPU χρειάζεται αρκετή μνήμη ενώ πραγματοποιεί τη διαδικασία της εκπαίδευσης. Η GPU από την άλλη πλευρά έχει ενσωματωμένη μία μνήμη VRAM. Η μνήμη της CPU μπορεί να χρησιμοποιηθεί και για άλλες εργασίες αλλά η μεταφορά δεδομένων από τη μνήμη της CPU στην GPU μπορεί να είναι μια σημαντική πρόκληση. Ο υπολογισμός μεγάλων και πολύπλοκων εργασιών χρειάζεται αρκετούς κύκλους ρολογιού στη CPU. Ο λόγος είναι ότι η CPU επεξεργάζεται τις δουλειές ακολουθιακά και έχει μικρότερο αριθμό πυρήνων από τον ανταγωνιστή της. Παρ' ότι οι GPUs είναι γρηγορότερες, ο χρόνος που χρειάζεται για να μεταφερθούν μεγάλες ποσότητες δεδομένων από την CPU στην GPU μπορεί να οδηγήσει σε μεγάλη χρονική καθυστέρηση που εξαρτάται και από την αρχιτεκτονική της διαδικασίας. Η καλύτερες CPUs έχουν εύρος ζώνης μνήμης περίπου 50 GB/s ενώ οι καλύτερες GPUs 750 GB/s. Παρακάτω μπορούμε να δούμε μία σύγκριση των πιο γνωστών GPUs και CPUs όσον αφορά το εύρος ζώνης μνήμης στην πάροδο των χρόνων (Εικόνα 6).



Εικόνα 6: Σύγκριση του εύρους ζώνης με το χρόνο

- *Σύνολο Δεδομένων*

  Η εκπαίδευση ενός μοντέλου μηχανικής μάθησης απαιτεί συνήθως ένα μεγάλο σύνολο δεδομένων, το οποίο προφανώς σημαίνει μεγάλες υπολογιστικές διαδικασίες από πλευράς μνήμης. Για να υπολογίσουμε τα δεδομένα αποδοτικά, η GPU είναι η καταλληλότερη επιλογή. Όσο μεγαλύτερος ο αριθμός των υπολογισμών που χρειάζονται, τόσο καταλληλότερη είναι η GPU έναντι της CPU.

- *Παραλληλισμός*

  Γυρίζοντας πίσω στην αναλογία, περιμένοντας τα φορτηγά για να γεμίσουν το φορτίο τους μπορεί να απαιτεί αρκετό χρόνο που μπορούμε να μειώσουμε εάν χρησιμοποιήσουμε περισσότερα φορτηγά ταυτοχρόνως. Ωστόσο, ο χρόνος φόρτωσης (latency) μπορεί να παραμείνει μυστικός καθώς τα φορτηγά απαιτούν περισσότερο χρόνο για να φορτώσουν. (thread parallelism). Το γεγονός αυτό "κρύβει" το latency έτσι ώστε η GPU να προσφέρει υψηλό εύρος ζώνης ενώ κρύβει το latency κάτω από τον παραλληλισμό των νημάτων (threads). Επομένως, για μεγάλες ποσότητες μνήμης, οι GPUs προσφέρουν το καλύτερο εύρος ζώνης μήνης (memory bandwidth) ενώ ταυτόχρονα δεν έχουν μειονεκτήματα λόγω του latency μέσω του παραλληλισμού των νημάτων.

- *Βελτιστοποίηση*

  Η βελτιστοποίηση των εργασιών είναι προφανώς ευκολότερη στην CPU από ότι στην GPU. Οι πυρήνες της CPU αν και λιγότεροι, είναι πολύ πιο δυνατοί από χιλιάδες πυρήνες της GPU. Ο κάθε πυρήνας της CPU μπορεί να ανταπεξέλθει σε διαφορετικές εντολές (MIMD αρχιτεκτονική) ενώ οι πυρήνες της GPU που συνήθως οργανώνονται σε μπλοκ των 32, εκτελούν την ακριβώς ίδια εντολή σε έναν δοσμένο χρόνο παράλληλα (SIMD αρχιτεκτονική). Ο παραλληλισμός σε πυκνά νευρωνικά δίκτυα είναι αρκετά δύσκολος αν λάβει κανείς υπόψιν την προσπάθεια που χρειάζεται. Κατά συνέπεια, σύνθετες τεχνικές βελτιστοποίησης είναι πιο δύσκολο να εφαρμοστούν στην GPU από ότι στην CPU.

- *Κόστος Αποδοτικότητας*

  Είναι προφανές ότι η εκπαίδευση μικρών δικτύων με μικρότερα σύνολα δεδομένων (datasets) όπου ο χρόνος δεν είναι μία παράμετρος, η CPU μπορεί να χρησιμοποιηθεί έναντι της GPU. Το κόστος λειτουργίας της GPU είναι αυστηρά μεγαλύτερο από αυτό λειτουργίας της CPU.

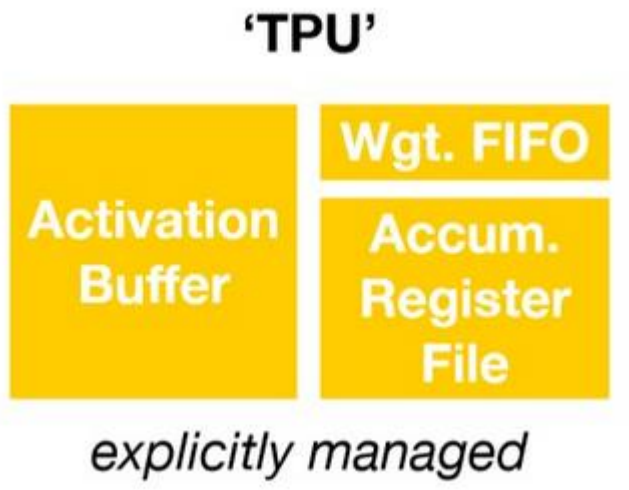
3.5 Μονάδα Επεξεργασίας Τανυστών (Tensor Processing Unit – TPU)

Η μονάδα επεξεργασίας τανυστών (tensor processing unit – TPU) αποτελεί ένα ειδικευμένο για εφαρμογές ολοκληρωμένο κύκλωμα (application specific integrated circuit – ASIC) που λειτουργεί σαν επιταχυντής (accelerator) για εφαρμογές τεχνητής νοημοσύνης (artificial intelligence – AI). Αναπτύχθηκε από την Google με ειδικό σκοπό τις εφαρμογές νευρωνικών δικτύων και μηχανικής μάθησης.

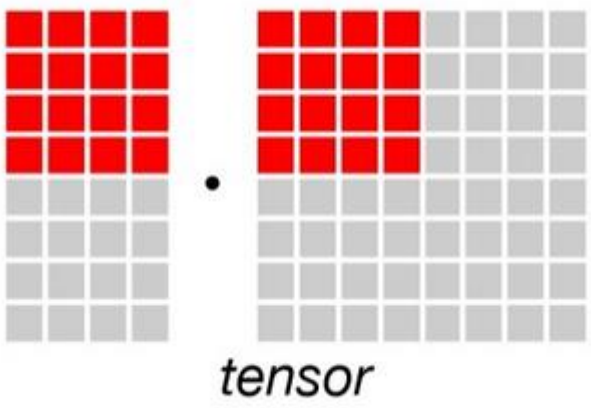
Η TPU ανακοινώθηκε από την Google το 2016, και ειπώθηκε ότι χρησιμοποιούταν από τα κέντρα δεδομένων της για πάνω από έναν χρόνο. Το chip, έχει σχεδιαστεί συγκεκριμένα για το Tensorflow, ένα framework της google το οποίο αποτελεί μία βιβλιοθήκη συμβολικών μαθηματικών που χρησιμοποιείται για εφαρμογές μηχανικής μάθησης όπως τα νευρωνικά δίκτυα.

Οι TPUs σχεδιάστηκαν από την αρχή για να επιτρέπουν τη γρηγορότερη επεξεργασία εφαρμογών. Είναι πολύ γρήγορες στην εκτέλεση πυκνών διανυσμάτων και στις πράξεις πινάκων και είναι σχεδιασμένες να τρέχουν πολύ γρήγορα προγράμματα βασισμένα στο Tensorflow. Ενδείκνυνται για εφαρμογές που κυριαρχούνται από υπολογισμούς πινάκων και για εφαρμογές και μοντέλα χωρίς συγκεκριμένες πράξεις του Tensorflow μέσα στον υπολογιστικό τους γράφο. Αυτό σημαίνει πως έχουν μικρότερη ευελιξία σε σύγκριση με τις CPUs και GPUs και αποτελούν για την ώρα την καταλληλότερη επιλογή για μοντέλα του Tensorflow. Μία TPU μπορεί να χειριστεί εκατοντάδες χιλιάδες εντολές σε κάθε κύκλο ρολογιού αλλά ακόμα δεν έχουν αναπτυχθεί compilers που να επιτρέπουν τον γενικού σκοπού προγραμματισμό πράγμα που μας περιορίζει μόνο στο Tensorflow.

Παρακάτω φαίνονται η αρχιτεκτονική μνήμης μίας TPU και η μικρότερη δομή που υποστηρίζει, δηλαδή ένας τανυστής ή μία NxN μονάδα δεδομένων.



Εικόνα 7: αρχιτεκτονική μνήμης μίας TPU



Εικόνα 8: μικρότερη μονάδα δεδομένων στην TPU

3.6 Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών (Field Programmable Gate Array – FPGA)

Το FPGA ή Field Programmable Gate Array ή Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών αποτελεί έναν τύπο προγραμματιζόμενου ολοκληρωμένου κυκλώματος γενικής χρήσης το οποίο διαθέτει πολύ μεγάλο αριθμό τυποποιημένων πυλών και άλλων ψηφιακών λειτουργιών όπως απαριθμητές, καταχωρητές μνήμης, γεννήτριες PLL κα. Σε ορισμένα από αυτά ενσωματώνονται και αναλογικές λειτουργίες. Κατά τον προγραμματισμό του FPGA, ο οποίος γίνεται πάντοτε ενώ αυτό είναι τοποθετημένο στο τυποποιημένο κύκλωμα, ενεργοποιούνται οι επιθυμητές λειτουργίες και διασυνδέονται μεταξύ τους έτσι ώστε το FPGA να συμπεριφέρεται ως ολοκληρωμένο κύκλωμα με συγκεκριμένη λειτουργία.

Εικόνα 9: FPGA τύπου Stratix IV GX, από την Altera

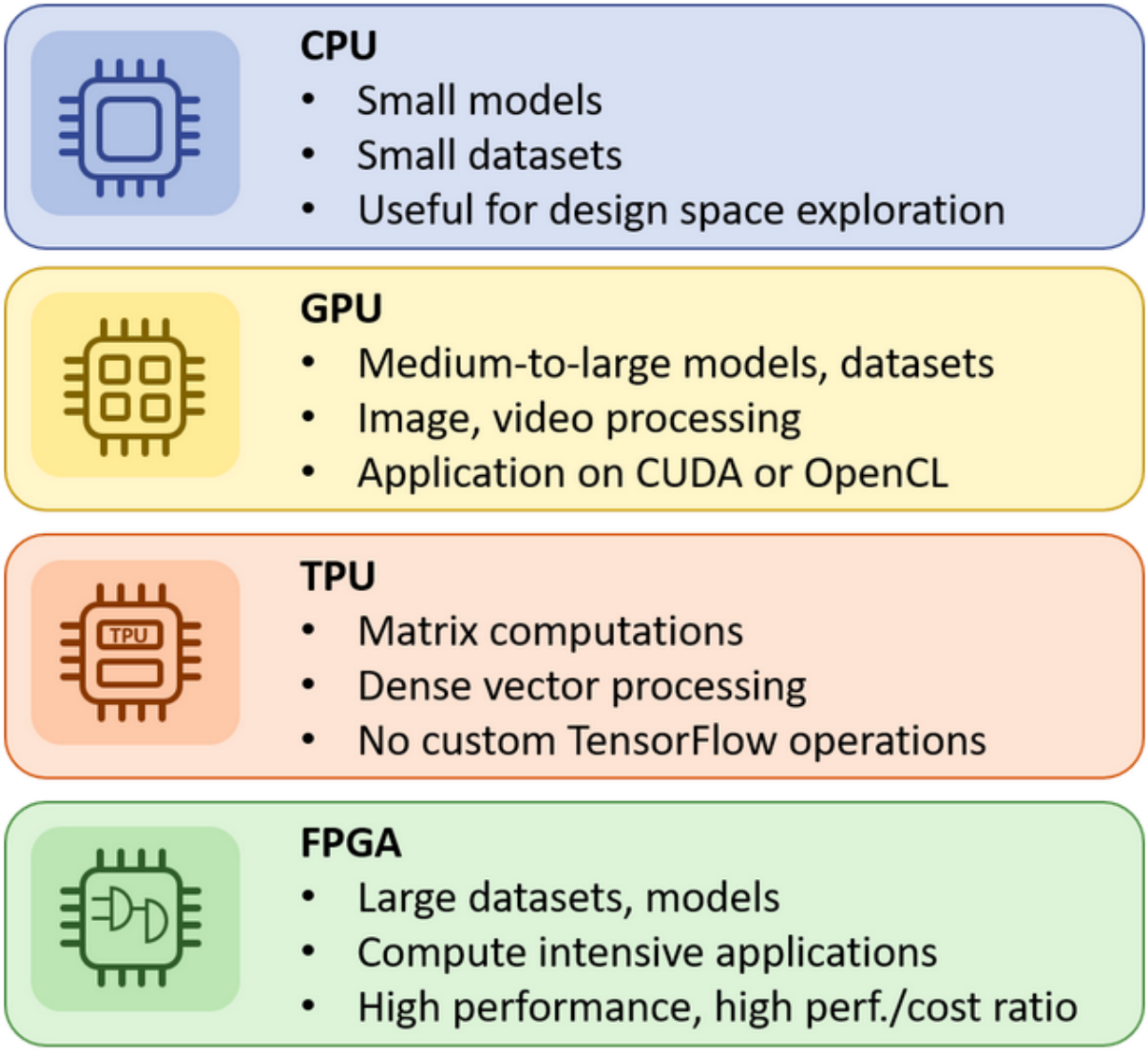
Το FPGA μπορεί να προγραμματιστεί σε μία συγκεκριμένη γλώσσα που ονομάζεται *γλώσσα περιγραφής υλικού* με γνωστότερες την VHDL, AHDL, Verilog κ.α. Στο παρελθόν, τα FPGAs συμπεριφέρονταν ως προγραμματιζόμενα chips που κυρίως χρησιμοποιούνταν για την υλοποίηση λογικών πράξεων και συγκεκριμένων συναρτήσεων. Αντιθέτως, τα σύγχρονα FPGAs αποτελούν μία πολύ ισχυρή επεξεργαστική μονάδα που μπορεί να προγραμματιστεί για να ικανοποιεί απαιτήσεις εφαρμογών. Στην ουσία με τη χρήση των FPGAs μπορούμε να δημιουργήσουμε προσαρμοσμένες αρχιτεκτονικές ειδικευμένες για συγκεκριμένες εφαρμογές. Με αυτόν τον τρόπο μπορούμε να πετύχουμε πολύ μεγάλη απόδοση, χαμηλό κόστος και χαμηλότερη κατανάλωση ισχύος σε αντίθεση με τις προηγούμενες επιλογές των CPU και GPU. Ο προγραμματισμός των FPGAs πραγματοποιείται πλέον με ειδικά λογισμικά όπως το OpenCL και το HLS (High Level Synthesis) που καθιστούν αυτόν πολύ πιο εύκολο από ότι στο παρελθόν. Λόγω αυτού του περιορισμού τα FPGAs προσφέρουν περιορισμένη ευελιξία σε αντίθεση με τις άλλες πλατφόρμες.

Ο καλύτερος τρόπος για να χρησιμοποιηθεί ένα FPGA για την εκπαίδευση ενός μοντέλου είναι με τη χρήση προεπιλεγμένων αρχιτεκτονικών ειδικευμένων για την εφαρμογή που μας ενδιαφέρει. Με τον τρόπο αυτό πετυχαίνουμε καλύτερη απόδοση χωρίς να χρειαστεί να αλλάξουμε τον κώδικά μας. Οι προεπιλεγμένες αρχιτεκτονικές επιτάχυνσης παρέχουν όλα τα διαθέσιμα APIs (Aplication Interface) και τις βιβλιοθήκες για το framework στο οποίο θέλουμε να προγραμματίσουμε (Python, Scala, Java, R και Apache Spark) και το οποίο επιτρέπει την υπερφόρτωση των πιο εντατικών εργασιών και την εκφόρτωση τους στα FPGAs.

3.7 Σύγκριση και Επιλογή

Όταν ο επιστήμονας δεδομένων πρέπει να διαλέξει ποια πλατφόρμα υλικού θα χρησιμοποιήσει για να τρέξει την εφαρμογή του έρχεται σε ένα μεγάλο δίλημμα το οποίο ξεδιαλύνεται με προσοχή στα συγκριτικά αποτελέσματα (bentchmarks) και στη σύγκριση

με άλλες πλατφόρμες. Χρησιμοποιώντας το ίδιο σύνολο δεδομένων και την ίδια υπολογιστική πλατφόρμα, ο επιστήμονας μπορεί να εκτελέσει τις ίδιες εργασίες και να αξιολογήσει την πλέον κατάλληλη πλατφόρμα για αυτές. Γενικότερα όμως θα πρέπει να λαμβάνει υπόψιν του τα πλεονεκτήματα και τα μειονεκτήματα των παραπάνω αρχιτεκτονικών που συγκεντρώνονται στην παρακάτω εικόνα:



Εικόνα 10: Υπολογιστικές Πλατφόρμες και τα βασικά χαρακτηριστικά τους

ΚΕΦΑΛΑΙΟ 4:
4.ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ ΚΑΙ ΝΕΥΡΩΝΙΚΑ ΔΙΚΤΥΑ

4.1 Εισαγωγή

Η Μάθηση (Learning) είναι μία από τις θεμελιώδεις ιδιότητες της νοήμονος συμπεριφοράς του ανθρώπου. Αν και η έννοια της μάθησης, παρά τις μελέτες από πολλά επιστημονικά πεδία, δεν έχει γίνει πλήρως κατανοητή, όπως γίνεται αντιληπτή στην καθημερνή ζωή μπορεί να συνδεθεί με δύο βασικές ιδιότητες:

- την ικανότητα απόκτησης γνώσης μέσω της αλληλεπίδρασης με το περιβάλλον
- την ικανότητα βελτίωσης του τρόπου με τον οποίο εκτελείται μια ενέργεια μέσω της επανάληψης.

Ο άνθρωπος μέσω της παρατήρησης, μπορεί και κατανοεί το περιβάλλον του κατασκευάζοντας απλοποιημένες και αφαιρετικές εκδοχές αυτού, τα λεγόμενα *μοντέλα,* ενώ ταυτόχρονα έχει την δυνατότητα να οργανώνει και να συσχετίζει εμπειρίες, παράγοντας νέες δομές που ονομάζονται *πρότυπα*. Η δημιουργία μοντέλων και προτύπων μέσω ενός συνόλου δεδομένων από ένα υπολογιστικό σύστημα, ονομάζεται Μηχανική Μάθηση.

Ως κλάδος της Τεχνητής Νοημοσύνης, η Μηχανική Μάθηση έχει σαν σκοπό τη δημιουργία συστημάτων ικανών να μαθαίνουν, να βελτιώνουν δηλαδή την απόδοσή τους σε κάποιους τομείς μέσω της αξιοποίησης προηγούμενης γνώσης και εμπειρίας. Η Μηχανική Μάθηση, μας δίνει τη δυνατότητα να αντιμετωπίσουμε εργασίες που είναι πολύ δύσκολο να επιλυθούν με προγράμματα ρητά γραμμένα και σχεδιασμένα από ανθρώπους. Από επιστημονική και φιλοσοφική άποψη, η μελέτη της είναι ενδιαφέρουσα, διότι η ανάπτυξη της κατανόησής μας για αυτήν, συνεπάγεται την ανάπτυξη της κατανόησης μας για τις αρχές που διέπουν γενικά τη νοημοσύνη.

4.2 Θεμελίωση

Σε πρακτικό επίπεδο, όταν μιλάμε για ένα σύστημα Μηχανικής Μάθησης, ουσιαστικά μιλάμε για έναν αλγόριθμο Μηχανικής Μάθησης ο οποίος εφαρμόζεται σε ένα σύνολο δεδομένων (dataset) και έχει την δυνατότητα να μαθαίνει από αυτό. Σύμφωνα με τον Mitchell (1997):

"Ένα πρόγραμμα υπολογιστή λέμε ότι μαθαίνει από την εμπειρία E ως προς κάποια κλάση εργασιών T και μέτρο απόδοσης P, αν η απόδοσή του σε εργασίες από το T, όπως μετριέται από το P, βελτιώνεται μέσω της εμπειρίας E."

Οι κλάσεις εργασιών T συνήθως, περιγράφονται με βάση τον τρόπο που ο αλγόριθμος Μηχανικής Μάθησης επεξεργάζεται ένα πρότυπο (example). Για ένα σύστημα Μηχανικής Μάθησης, πρότυπο είναι ένα στοιχείο από τα δεδομένα (dataset) του

προβλήματος, το οποίο έχει μια συγκεκριμένη αναπαράσταση (representation) την οποία μπορεί να καταλαβαίνει. Για την αναπαράσταση αυτήν, χρησιμοποιείται μια συλλογή χαρακτηριστικών (features). Τυπικά αναπαριστούμε ένα πρότυπο ως ένα διάνυσμα $x \in R^n$ όπου κάθε στοιχείο x_i του διανύσματος, αποτελεί και ένα διαφορετικό χαρακτηριστικό. Έτσι, αν για παράδειγμα θέλουμε να σχεδιάσουμε ένα σύστημα Μηχανικής Μάθησης το οποίο θα μπορεί να αναγνωρίζει τα αντικείμενα σε μια φωτογραφία, η κλάση εργασίας Τ είναι το πρόβλημα ταξινόμησης - κατηγοριοποίησης (classification) των φωτογραφιών, πρότυπα είναι οι φωτογραφίες οι οποίες έχουν ως χαρακτηριστικά τα pixels τους. Αυτά τα χαρακτηριστικά θα δοθούν, για κάθε πρότυπο, ως είσοδος στον αλγόριθμο.

Για να αξιολογήσουμε τις ικανότητες ενός αλγορίθμου Μηχανικής Μάθησης, πρέπει να σχεδιάσουμε ένα ποσοτικό μέτρο της απόδοσής του. Συνήθως αυτό το μέτρο απόδοσης είναι συγκεκριμένο για την εργασία που εκτελείται από το σύστημα. Για εργασίες όπως η ταξινόμηση - κατηγοριοποίηση , συχνά μετράμε την *ακρίβεια (accuracy) με* την οποία ταξινομεί το μοντέλο. Η ακρίβεια είναι το ποσοστό των παραδειγμάτων για τα οποία το μοντέλο παράγει τη σωστή έξοδο σε σχέση με τον αρχικό αριθμό των παραδειγμάτων που κλήθηκε να επεξεργαστεί.

Η βασικότερη φάση κάθε αλγόριθμου Μηχανικής Μάθησης είναι η φάση της εκπαίδευσης (training phase), όπου ο αλγόριθμος χρησιμοποιεί ως είσοδο ένα σύνολο δεδομένων πάνω στα οποία εκπαιδεύεται (training set), προς επίτευξη του σκοπού του, δηλαδή τη δημιουργία νέας γνώσης. Την εκπαίδευση ακολουθεί η φάση της πιστοποίησης-ελέγχου (validation/testing phase) της παραγόμενης νέας γνώσης. Σε αυτήν την φάση υπολογίζεται η ακρίβεια και η αποδοτικότητα του αλγορίθμου με τη βοήθεια δεδομένων ελέγχου (test set) και εξάγονται όλα τα συμπεράσματα σχετικά με τον τρόπο που οι διάφορες παράμετροι επηρεάζουν το τελικό μοντέλο. Τέλος, ο αλγόριθμος περνάει στο στάδιο εφαρμογής (application phase) δηλαδή η νέα γνώση δίνεται προς χρήση σε εφαρμογές στις οποίες αυτή είναι απαραίτητη προκειμένου να λυθούν πραγματικά προβλήματα.

4.3 Είδη Μηχανικής Μάθησης

Γενικότερα, ο τομέας της Μηχανικής Μάθησης αναπτύσσει τρεις τρόπους μάθησης, ανάλογους με τους τρόπους με τους οποίους μαθαίνει ο άνθρωπος: την επιβλεπόμενη μάθηση, τη μη επιβλεπόμενη μάθηση και την ενισχυτική μάθηση. Ωστόσο ιδιαίτερη αναφορά αξίζει να κάνουμε και στην βαθιά μάθηση αν και η τελευταία αποτελεί μια ειδική κατηγορία της μηχανικής μάθησης και κυρίως των νευρωνικών δικτύων και δεν μπορεί να χαρακτηριστεί ως μια επιπλέον κατηγορία της παραπάνω γενικής κατηγοριοποίησης.

4.3.1 Επιβλεπόμενη Μάθηση

Στην *επιβλεπόμενη μάθηση (Supervised Learning) ή μάθηση με παραδείγματα,* δοσμένων των μεταβλητών εισόδου Χ (ανεξάρτητες) και των μεταβλητών εξόδου Υ (εξαρτημένες), το σύστημα καλείται να μάθει την συνάρτηση-στόχο Φ, η οποία χρησιμοποιείται για την πρόβλεψη των Υ μέσω των Χ, δηλαδή την σχέση $Y = \Phi(X)$. Η συνάρτηση αυτή αποτελεί έκφραση του μοντέλου που περιγράφει τα δεδομένα του εκάστοτε προβλήματος. Ονομάζεται επιβλεπόμενη μάθηση καθώς είναι μια διαδικασία εκπαίδευσης που μπορεί να παρομοιαστεί με ένα "δάσκαλο" που επιτηρεί την μάθηση του συστήματος και διορθώνει τις προβλέψεις του με βάσει τις γνωστές – επιθυμητές εξόδους (Υ). Στην επιβλεπόμενη μάθηση διακρίνονται δυο κατηγορίες προβλημάτων:

- Τα προβλήματα *ταξινόμησης (classification)*, που σχετίζονται με την δημιουργία μοντέλων που προβλέπουν (κατηγοριοποιούν) μεταξύ διακριτών κλάσεων.
- Τα προβλήματα *παρεμβολής (regression)*, που σχετίζονται με την δημιουργία μοντέλων που προβλέπουν αριθμητικές τιμές σε συνεχή διαστήματα.

4.3.2 Μη Επιβλεπόμενη Μάθηση

Στην *μη επιβλεπόμενη μάθηση (Unsupervised Learning) ή μάθηση από παρατήρηση*, το σύστημα, δοσμένων μόνο των μεταβλητών εισόδου X, πρέπει να διερευνήσει και να ανακαλύψει συσχετίσεις για τα δοσμένα δεδομένα, δημιουργώντας κατάλληλα πρότυπα, χωρίς όμως να γνωρίζει τις επιθυμητές εξόδους. Τα προβλήματα στην μη επιβλεπόμενη μάθηση ομαδοποιούνται σε δύο κατηγορίες:

- Στα προβλήματα *ομαδοποίησης (clustering)*: Σε ένα πρόβλημα ομαδοποίησης, ο αλγόριθμός μας προσπαθεί να ανακαλύψει εγγενείς ομαδοποιήσεις μεταξύ των δεδομένων στα οποία εφαρμόζεται (π.χ. ομαδοποιήσεις των πελατών με βάση την αγοραστική τους συμπεριφορά).
- Στα προβλήματα *συσχέτισης (association)*: Σε ένα πρόβλημα συσχέτισης, ο αλγόριθμός μας προσπαθεί να ανακαλύψει κανόνες (rules) που χαρακτηρίζουν ένα μέρος των δεδομένων μας (π.χ. από ένα μεγάλο αριθμό αντικειμένων ζητείται να βρεθεί για ένα υποσύνολο τους, ποια αντικείμενα αγοράζονται μαζί)

4.3.3 Ενισχυτική Μάθηση

Στην *ενισχυτική μάθηση (Reinforcement Learning)*, ο αλγόριθμος μαθαίνει μια στρατηγική ενεργειών μέσα από άμεση αλληλεπίδραση με το περιβάλλον. Χρησιμοποιείται κυρίως σε προβλήματα Σχεδιασμού (Planning), όπως για παράδειγμα ο έλεγχος κίνησης ρομπότ και η βελτιστοποίηση εργασιών σε εργοστασιακούς χώρους.

4.3.4 Βαθιά Μάθηση

Ένας ιδιαίτερος κλάδος της μηχανικής μάθησης, *η βαθιά μάθηση (deep learning)*, έχει αποδειχθεί ιδιαίτερα αποτελεσματικός τα τελευταία χρόνια. Η βαθιά μάθηση είναι μια οικογένεια αλγορίθμων εκμάθησης αναπαράστασης που χρησιμοποιεί σύνθετες αρχιτεκτονικές νευρωνικών δικτύων με μεγάλο αριθμό κρυφών επιπέδων, το καθένα από τα οποία αποτελείται από απλούς αλλά μη γραμμικούς μετασχηματισμούς (νευρώνες) τους οποίους εφαρμόζει στα δεδομένα εισόδου. Δεδομένου ότι τα δομικά αυτά στοιχεία μετασχηματισμού παρουσιάζονται σε πολύ μεγάλο πλήθος στις αρχιτεκτονικές αυτές, δίνεται η δυνατότητα για την διαμόρφωση πολύ σύνθετων μοντέλων τα οποία μπορούν να επιλύσουν περίπλοκα και απαιτητικά προβλήματα ταξινόμησης, παλινδρόμησης και πολλές άλλες μαθησιακές εργασίες.

Υπάρχουν αρκετοί λόγοι που οδήγησαν στην ανάπτυξη της βαθιάς μάθησης και στην τοποθέτησή της στο κέντρο του τομέα της μηχανικής μάθησης τις τελευταίες

δεκαετίες. Ένας λόγος, ίσως και ο κυριότερος, αντιπροσωπεύεται ασφαλώς από την πρόοδο στον τομέα του υλικού (hardware), με τη διαθεσιμότητα νέων επεξεργαστών, όπως μονάδες επεξεργασίας γραφικών (GPU), οι οποίες μείωσαν σημαντικά τον χρόνο που απαιτείται για την εκπαίδευση τέτοιων σύνθετων δικτύων, δεδομένης της υψηλής υπολογιστικής τους πολυπλοκότητας. Ένας άλλος λόγος είναι η συνεχώς αυξανόμενη διαθεσιμότητα σε σύνολα δεδομένων για την κατάρτιση ενός τέτοιου συστήματος, που απαιτείται για την εκπαίδευση αρχιτεκτονικών ορισμένου βάθους και με μεγάλες διαστάσεις στα δεδομένα εισόδου.

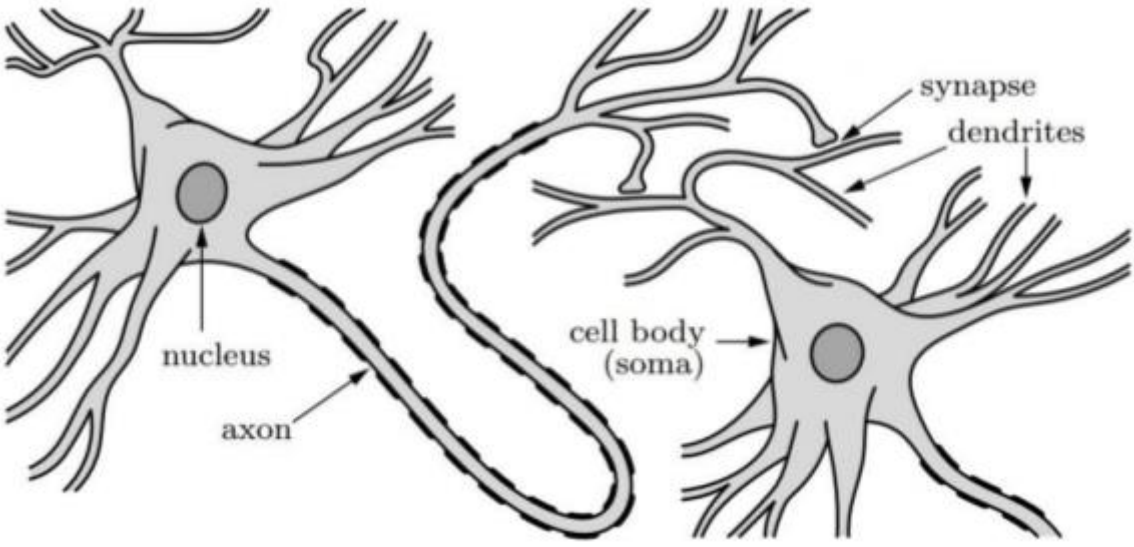
Η βαθιά μάθηση, ως προϊόν των νευρωνικών δικτύων, βασίζεται στον τρόπο με τον οποίο ο ανθρώπινος εγκέφαλος επεξεργάζεται πληροφορίες και μαθαίνει, ανταποκρινόμενος σε εξωτερικά ερεθίσματα. Αποτελεί ένα μοντέλο μηχανικής μάθησης, αποτελούμενο από διάφορα επίπεδα αναπαράστασης, όπου τα βαθύτερα επίπεδα λαμβάνουν ως είσοδο τα αποτελέσματα των προηγούμενων, δημιουργώντας όλο και πιο αφηρημένες αναπαραστάσεις της πληροφορίας.

4.4 Νευρωνικά Δίκτυα

Τα Τεχνητά Νευρωνικά Δίκτυα (ΤΝΔ) είναι μια περίπτωση συστήματος Μηχανικής Μάθησης και αποτελούν υπολογιστικά-αλγοριθμικά μοντέλα επεξεργασίας δεδομένων, τα οποία αποτελούνται από ένα πλήθος τεχνητών νευρώνων και έχουν σαν σκοπό τη προσέγγιση και προσομοίωση της δομής και της λειτουργίας του ανθρώπινου εγκεφάλου.

4.4.1 Βιολογικά Νευρωνικά Δίκτυα

Ο εγκέφαλος είναι ένα εξαιρετικά πολύπλοκο, μη γραμμικό, παράλληλο σύστημα επεξεργασίας πληροφοριών, στο οποίο ο άνθρωπος οφείλει την ικανότητα του να σκέφτεται, να θυμάται και να επιλύει προβλήματα. Όπως είναι γνωστό από την Βιολογία, η δομική μονάδα του εγκεφάλου είναι οι νευρώνες. Ο τυπικός βιολογικός νευρώνας είναι ένα μεγάλο σε μέγεθος κύτταρο, το οποίο αποτελείται από το σώμα, τους δενδρίτες και τον άξονα.



Εικόνα 11: Αναπαράσταση Βιολογικού Νευρώνα

Πιο συγκεκριμένα, το σώμα περιλαμβάνει τον πυρήνα του νευρώνα, οι δενδρίτες αποτελούν τα τμήματα μέσω των οποίων ο νευρώνας λαμβάνει τα σήματα από τους γειτονικούς του, ενώ ο άξονας είναι η έξοδός του και το μέσο σύνδεσής του με άλλους νευρώνες. Σε ένα νευρωνικό δίκτυο, ένας νευρώνας συνδέεται με χιλιάδες άλλους, μέσω σημείων σύνδεσης που ονομάζονται συνάψεις. Καθώς η πληροφορία στο νευρωνικό δίκτυο μεταδίδεται μέσω ηλεκτρικών παλμών, οι συνάψεις έχουν την δυνατότητα να επιταχύνουν ή επιβραδύνουν την ροή των ηλεκτρικών φορτίων προς το σώμα του νευρώνα, καθορίζοντας έτσι το σήμα που θα φτάσει σε αυτόν. Τα ηλεκτρικά σήματα που εισέρχονται στο σώμα του μέσω των δενδριτών, συνδυάζονται και αν το αποτέλεσμα ξεπερνά κάποια τιμή κατωφλίου, ο νευρώνας πυροδοτείται και το σήμα διαδίδεται προς τους υπόλοιπους νευρώνες.

4.4.2 Τεχνητά Νευρωνικά Δίκτυα

Τα Τεχνητά Νευρωνικά Δίκτυα (ΤΝΔ) επεξεργάζονται τις πληροφορίες με παρόμοιο τρόπο που κάνει ο ανθρώπινος εγκέφαλος. Όπως ένα βιολογικό έτσι και ένα τεχνητό νευρωνικό δίκτυο, αποτελείται από ένα μεγάλο αριθμό εξαιρετικά διασυνδεδεμένων στοιχείων επεξεργασίας και αποθήκευσης, τους τεχνητούς νευρώνες, που εργάζονται παράλληλα για την επίλυση συγκεκριμένου προβλήματος. Οι δυνατότητες και ο τρόπος λειτουργίας των νευρώνων σε συνδυασμό με αυτήν την αρχιτεκτονική, δίνει στα ΤΝΔ κάποιες βασικές ιδιότητες, χάρη στις οποίες, αυτά βρίσκονται ανάμεσα στις πιο δημοφιλείς και αποδοτικές μορφές συστημάτων μηχανικής μάθησης. Αυτές οι ιδιότητες είναι:

- Ικανότητα μάθησης μέσω της εκπαίδευσης τους με παραδείγματα-δεδομένα, με αποτέλεσμα να βελτιώνουν την εμπειρία τους και να προσαρμόζονται στο περιβάλλον τους.
- Ικανότητα γενίκευσης της γνώσης τους πάνω στα δεδομένα που εκπροσωπούν το πρόβλημα που πρέπει να επιλύσουν.
- Αυτό-οργάνωση, καθώς ένα τεχνητό νευρωνικό δίκτυο μπορεί να δημιουργήσει τη δική του οργάνωση των πληροφοριών που λαμβάνει κατά τη διάρκεια της εκμάθησης.
- Κατανεμημένος και παράλληλος υπολογισμός.
- Ανοχή σε σφάλματα και θορυβώδεις εισόδους, με την έννοια ότι τόσο η κακή λειτουργία ή απώλεια ενός νευρώνα όσο και ελλιπείς-θορυβώδεις είσοδοι στο ΤΝΔ δεν επηρεάζουν την λειτουργία και ην απόδοσή του.

4.4.2.1 Μοντέλο Τεχνητού Νευρώνα

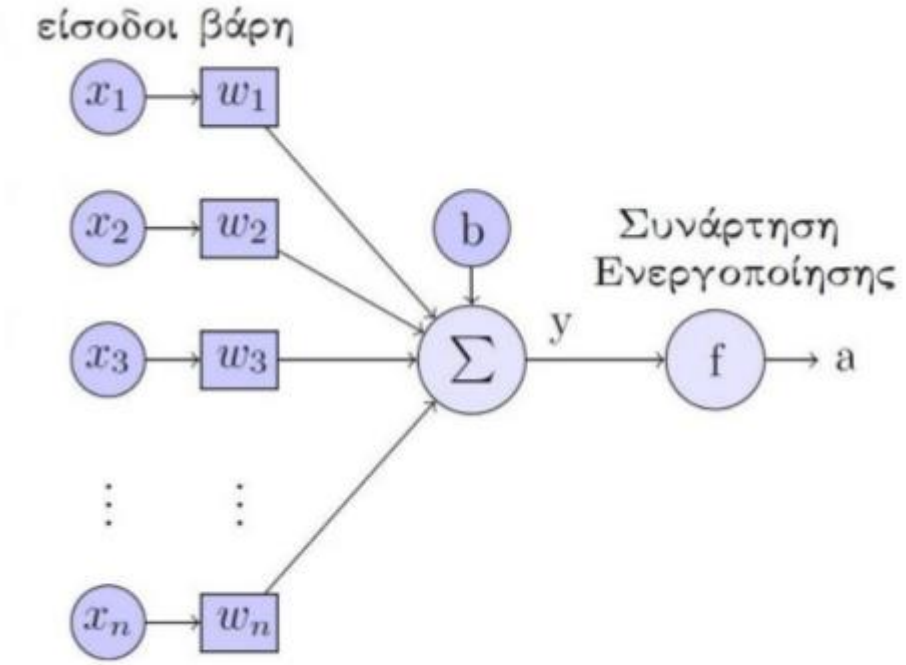
Όπως αναφέρθηκε παραπάνω, στην γενική τους εικόνα, οι νευρώνες σε ένα νευρωνικό δίκτυο (τεχνητό ή βιολογικό), διασυνδέονται μεταξύ τους μέσω συνάψεων και πυροδοτούνται όταν ένας γραμμικός συνδυασμός των εισόδων τους υπερβαίνει ένα κατώφλι.

Ο τεχνητός νευρώνας είναι ένα μαθηματικό μοντέλο αποθήκευσης και επεξεργασίας της πληροφορίας, το οποίο αποτελεί μια αφηρημένη προσέγγιση του βιολογικού νευρώνα. Πιο συγκεκριμένα αποτελείται από ένα πλήθος καναλιών εισόδου **n**, τις συνάψεις, μέσω των οποίων δέχεται τις εισόδους του $\mathbf{x_i}$. Κάθε σύναψή του συσχετίζεται με ένα αριθμητικό βάρος $\mathbf{W_i}$, τα λεγόμενα συναπτικά βάρη (σε ένα σύνθετο ΤΝΔ οι είσοδοι μπορεί να είναι

τόσο δεδομένα που τροφοδοτούνται απευθείας στον νευρώνα όσο και έξοδοι από άλλους νευρώνες). Οι είσοδοι αφού πολλαπλασιαστούν με τα αντίστοιχα συναπτικά βάρη, πηγαίνουν σε έναν αθροιστή όπου και αθροίζονται, μαζί με μια αριθμητική τιμή πόλωσης **b** (bias). Το αποτέλεσμα , τροφοδοτείται σε μια συνάρτηση ενεργοποίησης , η έξοδος της οποία αποτελεί και την τελική έξοδο, γνωστή και ως ενεργοποίηση **α** του νευρώνα. Η μαθηματική έκφραση της λειτουργίας του νευρώνα είναι η εξής:

$$y = \sum_{i=1}^{n} w_i x_i + b = \sum_{i=0}^{n} w_i x_i = \boldsymbol{W}^T \boldsymbol{X}$$

Έξοδος = α = f(y)



Εικόνα 12: Μαθηματικό Μοντέλο Τεχνητού Νευρώνα

Αυτό το μοντέλο νευρώνα αποτελεί το βασικό μοντέλο νευρώνα που δομεί κάθε είδους νευρωνικό. Η φιλοσοφία λειτουργίας αυτού το μοντέλου είναι η εξής:

- Τα βάρη κάθε σύναψης καθορίζουν το βαθμό συμμετοχής κάθε εισόδου, δηλαδή το πόσο πολύ ή το πόσο λίγο κάθε είσοδος θα επηρεάσει την ενεργοποίηση του νευρώνα.
- Η τιμή πόλωσης αποτελεί το αριθμητικό κατώφλι που θα πρέπει να ξεπεράσει η συνολική είσοδος του νευρωνικού προκειμένου ο νευρώνας να πυροδοτηθεί-ενεργοποιηθεί. Ωστόσο, τόσο η τιμή του κατωφλίου, όσο και το αν τελικά ο

νευρώνας θα ενεργοποιηθεί ή όχι, θα καθοριστούν από την συνάρτηση ενεργοποίησης.

- Η τιμή $y = \sum_{i=1}^{n} W_i x_i + b$

του νευρώνα, μπορεί να πάρει πρακτικά τιμές από -∞ έως +∞, με αποτέλεσμα ο νευρώνας να μην μπορεί να ξέρει τα πραγματικά της όρια και κατ' επέκταση να μην υπάρχει κάποιο μέτρο σύγκρισης που θα καθορίζει αν αυτός θα ενεργοποιηθεί ή όχι. Το πρόβλημα αυτό, το λύνει η συνάρτηση ενεργοποίησης η οποία ουσιαστικά φράζει την έξοδο, δρα δηλαδή ως φίλτρο που την αντιστοιχίζει σε ένα συγκεκριμένο αριθμητικό διάστημα. Η πιο απλή συνάρτηση ενεργοποίησης είναι η βηματική (step function) αλλά καθώς δεν είναι διαφορίσιμη (κάτι το οποίο παίζει σημαντικό ρόλο στην διαδικασία εκπαίδευσης και την απόδοση ενός νευρωνικού δικτύου) δεν χρησιμοποιείται σχεδόν ποτέ. Κάποιες από τις πιο δημοφιλείς συναρτήσεις ενεργοποίησης (Εικόνα 13) είναι οι:

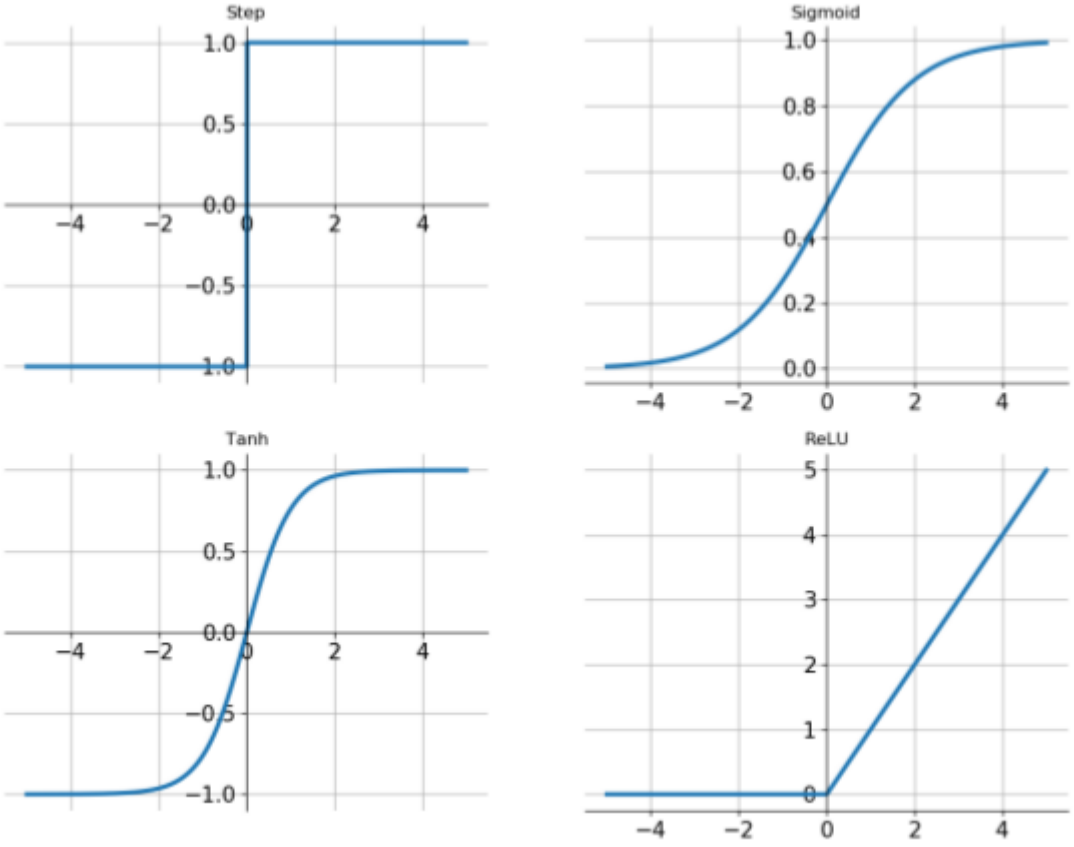
- Βηματική (Step): $$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$
- Υπερβολική Εφαπτομένη: $$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$
- Σιγμοειδής (Sigmoid): $$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$
- Rectified Linear Unit (ReLU): $$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

4.4.2.2 Δομές Τεχνητών Νευρωνικών Δικτύων

Το μοντέλο τεχνητού νευρώνα που παρουσιάστηκε παραπάνω, αποτελεί τον βασικό τεχνητό νευρώνα που δομεί την πλειοψηφία των τεχνητών νευρωνικών δικτύων. Με κριτήριο το πώς είναι συνδεδεμένοι οι νευρώνες μεταξύ τους, τα ΤΝΔ χωρίζονται σε δύο βασικές κατηγορίες:

- Στα **ΤΝΔ πρόσθιας τροφοδότησης (feedforward)**. Τα νευρωνικά δίκτυα πρόσθιας τροφοδότησης αποτελούν την πιο απλή αρχιτεκτονική νευρωνικών δικτύων. Σε αυτά τα δίκτυα οι νευρώνες οργανώνονται σε επίπεδα και κάθε νευρώνας συνδέεται μόνο με τους νευρώνες του μπροστινού γειτονικού του επιπέδου. Έτσι, η ροή της πληροφορίας σε αυτά τα δίκτυα είναι μονής κατεύθυνσης. Ένα ΤΝΔ πρόσθιας τροφοδότησης ουσιαστικά αναπαριστά μια συνάρτηση των τρεχουσών εισόδων του.
- Στα **ΤΝΔ ανατροφοδότησης ή αναδρομικά ΤΝΔ (feedback ή recurrent)**. Στα αναδρομικά νευρωνικά δίκτυα, σε αντίθεση με τα ΤΝΔ πρόσθιας τροφοδότησης, πληροφορία δεν ρέει μονόδρομα, αλλά οι έξοδοι του δικτύου ανατροφοδοτούνται πίσω στις εισόδους. Αυτό σημαίνει ότι τα επίπεδα ενεργοποίησης του δικτύου

σχηματίζουν ένα δυναμικό σύστημα το οποίο μπορεί να ευσταθεί πλήρως, να παρουσιάζει ταλαντώσεις ή ακόμα και χαοτική συμπεριφορά.



Εικόνα 13: Συναρτήσεις Ενεργοποίησης

4.4.2.3 Ανάκληση και Εκπαίδευση

Ο κύκλος ζωής ενός τεχνητού νευρωνικού δικτύου περιγράφεται από δύο βασικές λειτουργίες: την ανάκληση και την εκπαίδευση. Η ανάκληση (recall), είναι η διαδικασία κατά την οποία το νευρωνικό δίκτυο, για συγκεκριμένο διάνυσμα εισόδου, υπολογίζει το αντίστοιχο διάνυσμα εξόδου. Η εκπαίδευση (training) ή μάθηση του νευρωνικού, είναι η διαδικασία τροποποίησης της τιμής των βαρών του δικτύου, προκειμένου να τα προσαρμόσει με τέτοιο τρόπο ώστε για συγκεκριμένο διάνυσμα εισόδου, να παράγει την επιθυμητή έξοδο.

4.4.3 Τεχνητά Νευρωνικά Δίκτυα Πρόσθιας Τροφοδότησης

Στα ΤΝΔ πρόσθιας τροφοδότησης, η πληροφορία ρέει μονόδρομα από την είσοδο του νευρωνικού μέχρι την έξοδό του, χωρίς να ενεργοποιείται κανένας νευρώνας παραπάνω από μία φορά για συγκεκριμένη είσοδο. Σε αυτά υπάρχει ένα επίπεδου εισόδου, ένα επίπεδο εξόδου και προαιρετικά ένα η περισσότερα ενδιάμεσα επίπεδα. Παρακάτω θα εξετάσουμε τις βασικές περιπτώσεις ΤΝΔ πρόσθιας τροφοδότησης.

4.4.3.1 Αισθητήρας Perceptron και Εκπαίδευση

Ο αισθητήρας perceptron αποτελεί την πιο απλή τοπολογία ΤΝΔ πρόσθιας τροφοδότησης χωρίς κρυφά επίπεδα, καθώς αποτελείται μονάχα από έναν τεχνητό νευρώνα ο οποίος δομεί το επίπεδο εξόδου και στον οποίο τροφοδοτούνται όλες οι είσοδοι. Η λειτουργία του perceptron περιγράφεται πλήρως από την λειτουργία του απλού τεχνητού νευρώνα που παρουσιάστηκε παραπάνω.

Για την εκπαίδευση του perceptron χρησιμοποιείται επιβλεπόμενη μάθηση. Η έξοδος του αισθητήρα perceptron συγκρίνεται με τις τιμές στόχους t(p) κάθε προτύπου εκπαίδευσης p και μέσω του υπολογισμού του σφάλματος εξόδου υπολογίζονται και αναπροσαρμόζονται κατάλληλα τα συναπτικά βάρη w_0, w_1, ... , w_n των εισόδων προκειμένου στο τέλος να παραχθεί η επιθυμητή έξοδος. Έτσι με τυχαία αρχικοποίηση του διανύσματος συναπτικών βαρών $\mathbf{W} = [w_0, w_1, \dots, w_n]$, συνάρτηση ενεργοποίησης f και δοσμένων των προτύπων εκπαίδευσης $\mathbf{X^0}, \mathbf{X^1}, \dots, \mathbf{X^P}$ και των αντίστοιχων στόχων $t^0, t^1, \dots, t^P$ ο επαναληπτικός αλγόριθμος εκπαίδευσης του perceptron είναι:

Μέχρι να συμπληρωθεί ο προκαθορισμένος αριθμός εποχών εκπαίδευσης ή δεν γίνει καμία μεταβολή στα συναπτικά βάρη:

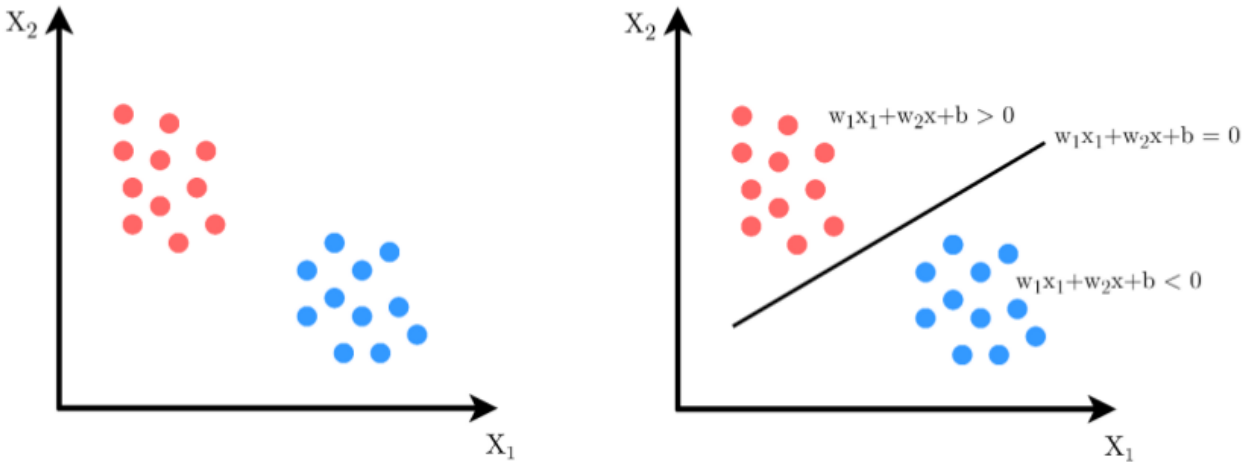
Για κάθε πρότυπο εισόδου $\boldsymbol{X}^{\boldsymbol{i}}$ με στόχο $\boldsymbol{t}^{\boldsymbol{i}}$:

1. Υπολόγισε την έξοδο $\boldsymbol{a} = \boldsymbol{f}\left(\sum_{\boldsymbol{j}=\boldsymbol{0}}^{\boldsymbol{n}} \boldsymbol{w_i x_j}^{\boldsymbol{i}}\right)$
2. Εάν $\boldsymbol{a} = \boldsymbol{t}$, δεν γίνεται καμία αλλαγή στα συναπτικά βάρη
3. Εάν $\boldsymbol{a} \neq \boldsymbol{t}$, τότε μετέβαλλε τα βάρη κατά την ποσότητα $\boldsymbol{\Delta W} = \boldsymbol{d} \cdot \boldsymbol{t} - \boldsymbol{a} * \boldsymbol{X}$
   Τα βάρη προσαρμόζονται ως εξής: $: \boldsymbol{W\ k} = \boldsymbol{W\ k-1}\ \boldsymbol{\Delta W}$, όπου $\boldsymbol{W\ k}$ τα βάρη της επανάληψης κ

Στον παραπάνω αλγόριθμο, όπως και σε κάθε αλγόριθμο εκπαίδευσης ενός νευρωνικού δικτύου, ως εποχή ονομάζεται ένας πλήρης κύκλος χρήσης όλων των προτύπων εκπαίδευσης, ενώ η παράμετρος d ονομάζεται ρυθμός εκπαίδευσης (learning rate) και καθορίζει το ρυθμό μεταβολής των βαρών του νευρωνικού δικτύου.

4.4.3.2 Δίκτυο perceptron ως γραμμικός διαχωριστής

Σε ένα πρόβλημα ταξινόμησης μεταξύ δύο κλάσεων, στόχος του perceptron είναι να ταξινομήσει σωστά ένα σύνολο παραδειγμάτων σε μια από τις κλάσεις C_1 , C_2 . Καθένα από



αυτά τα παραδείγματα έχει πρότυπο με διάνυσμα χαρακτηριστικών $\mathbf{X} = [x_1, x_2, ..., x_n]^T$, το οποίο αποτελεί την είσοδο του νευρώνα. Εξετάζοντας την ενεργοποίηση του νευρώνα $a = f(\mathbf{W}^T\mathbf{X})$, παρατηρούμε ότι η εξίσωση $\mathbf{W}^T\mathbf{X} = \mathbf{0}$ ορίζει ένα υπερεπίπεδο στο n-διάστατο χώρο εισόδου, όπου αν η είσοδος $\mathbf{X}$ βρίσκεται στην μια πλευρά αυτού του επιπέδου, δηλαδή $y > 0$, τότε η βηματική ενεργοποίηση του νευρώνα είναι 1, ενώ αν $y < 0$ η ενεργοποίηση είναι 0. Έτσι το perceptron μπορεί να λειτουργήσει σαν γραμμικός διαχωριστής μεταξύ των κλάσεων C_1 , C_2 , αν οι κλάσεις αυτές, στον n-διάστατο χώρο που ορίζουν τα παραδείγματά τους, μπορούν να διαχωριστούν γραμμικά, είναι δηλαδή γραμμικά διαχωρίσιμες.

Εικόνα 14: Ταξινόμηση γραμμικά διαχωρίσιμων κλάσεων

4.4.3.3 Πολυεπίπεδα Δίκτυα Perceptron (Multi-Layer Perceptron)

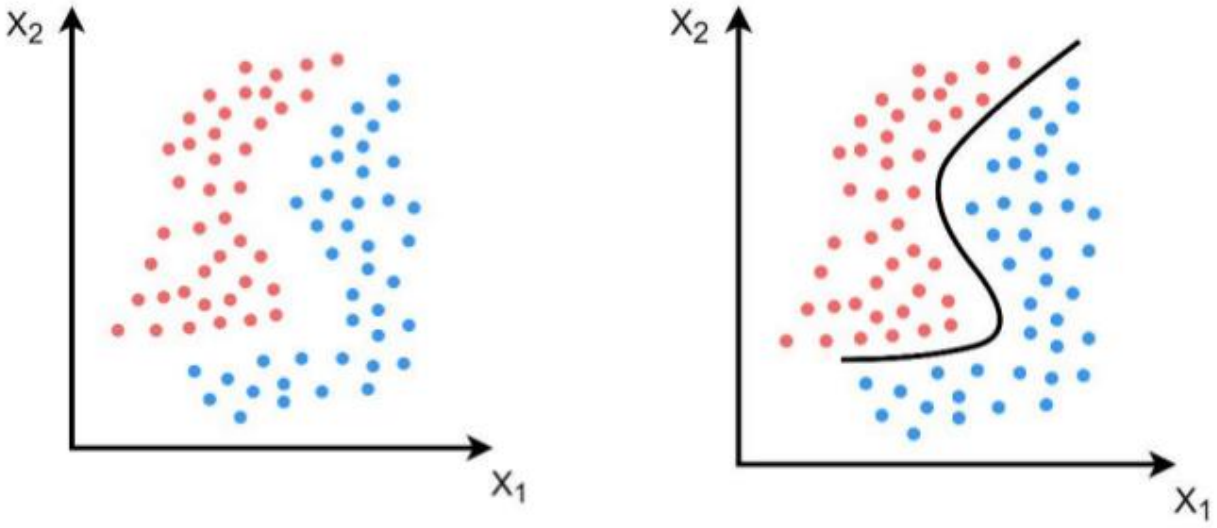
Όπως φάνηκε παραπάνω, οι δυνατότητες των δικτύων με μόνο έναν νευρώνα, περιορίζονται στο να μπορούν αναπαριστούνε μόνο επίπεδες επιφάνειες, με αποτέλεσμα τα δίκτυα αυτά να είναι ικανά να διαχωρίσουν με αρκετά μεγάλη ακρίβεια κλάσεις, οι οποίες όμως θα πρέπει να είναι γραμμικά διαχωρίσιμες.

Παρ' όλα αυτά, στην πλειοψηφία των προβλημάτων ταξινόμησης, ένα ΤΝΔ καλείται να ταξινομήσει πρότυπα τα οποία ανήκουν σε κλάσεις που διαχωρίζονται μεταξύ τους μη γραμμικά (Εικόνα 15). Για την αντιμετώπιση του προβλήματος αυτού, προστίθενται στο δίκτυο περισσότεροι νευρώνες, δημιουργώντας έτσι, πολυεπίπεδα ΤΝΔ, δηλαδή πιο σύνθετες δομές οι οποίες παράγουν και πιο σύνθετες, μη γραμμικές, επιφάνειες διαχωρισμού των κλάσεων. Τα δίκτυα αυτά είναι γνωστά ως πολυεπίπεδα δίκτυα perceptron (Multi-layer Perceptron ή MLP).

Εικόνα 15: Παράδειγμα μη γραμμικά διαχωρίσιμων κλάσεων

Οι νευρώνες στα πολυεπίπεδα δίκτυα perceptron ομαδοποιούνται και στοιβάζονται σε στρώματα - επίπεδα. Οι νευρώνες του ίδιου επιπέδου χρησιμοποιούν την ίδια συνάρτηση ενεργοποίησης, ενώ μεταξύ τους δεν υπάρχουν συνδέσεις. Συνδέσεις υπάρχουν μόνο μεταξύ των νευρώνων γειτονικών επιπέδων. Έτσι σχηματίζονται:

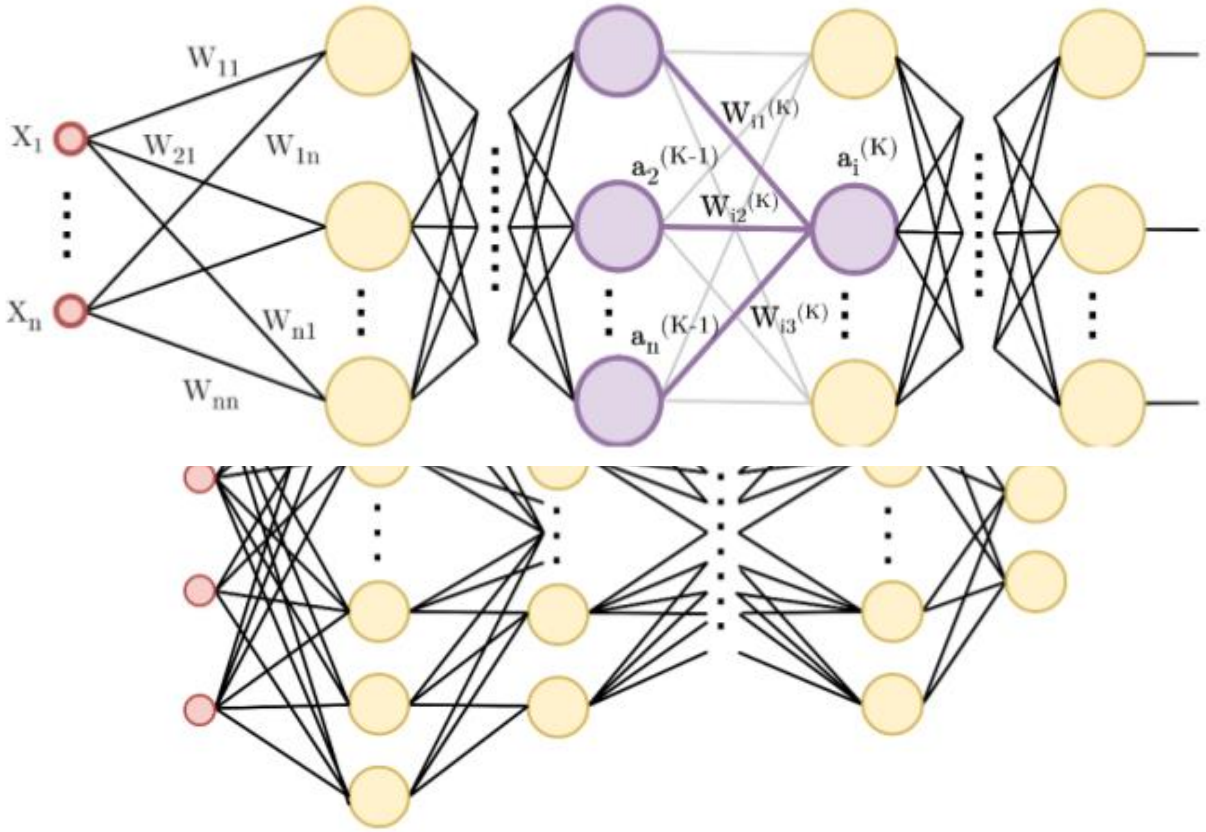
- Τα *κρυφά επίπεδα (hidden layers)*. Τα επίπεδα αυτά είναι ουσιαστικά ομάδες νευρώνων που βρίσκονται ανάμεσα στην είσοδο και το επίπεδο εξόδου. Ένα πολυεπίπεδο δίκτυο perceptron μπορεί να έχει ένα ή περισσότερα κρυφά επίπεδα. Αυτό που τα ξεχωρίζει από το επίπεδο εξόδου, είναι ότι η έξοδος των νευρώνων των κρυφών επιπέδων αποτελεί πάντα είσοδο για τους νευρώνες των επόμενων επιπέδων (κρυφά ή επίπεδο εξόδου).
- Το *επίπεδο εξόδου (output layer)*. Το επίπεδο αυτό, όπως γίνεται κατανοητό και από το όνομα, αποτελείται από νευρώνες, η έξοδος των οποίων αποτελεί και την τελική έξοδο του νευρωνικού.



Εικόνα 16: Παράδειγμα πολυεπίπεδου τεχνητού νευρωνικού δικτύου

4.4.3.4 Ανάκληση ΤΝΔ Πρόσθιας Τροφοδότησης

Για να περιγράψουμε την ανάκληση των πολυεπίπεδων δικτύων perceptron, δηλαδή το πως υπολογίζεται η έξοδος του δικτύου δοσμένου του διανύσματος εισόδου,



βασιζόμαστε στο μαθηματικό μοντέλο του απλού τεχνητού νευρώνα και χρησιμοποιούμε την εξής σημειογραφία:

- N_i, το πλήθος νευρώνων σε κάθε επίπεδο i
- $w_{ij}^{(K)}$, το συναπτικό βάρος από τον νευρώνα j του επιπέδου K – 1 στο νευρώνα i του επιπέδου K
- $f_i^{(K)}$, την συνάρτηση ενεργοποίησης του νευρώνα του επιπέδου K
- $\alpha_i^{(K)}$, την έξοδο-ενεργοποίηση του νευρώνα i του επιπέδου K

Έτσι η ενεργοποίηση του νευρώνα του επιπέδου K υπολογίζεται ως εξής (Εικόνα 17):

$$\alpha_i^{(K)} = f_i^{(K)} \left(\sum_{j=1}^{N_{K-1}} w_{ij}^{(K)} \, a_j^{(K-1)} + b_i^{(K)} \right)$$

Εικόνα 17: Ενεργοποίηση του νευρώνα i στο επίπεδο K του δικτύου

Με βάση τα παραπάνω, η διαδικασία υπολογισμού της εξόδου του πολυεπίπεδου ΤΝΔ, ορίζεται με τον εξής αλγόριθμο:

4.4.3.5 Εκπαίδευση Πολυεπίπεδου Δικτύου Πρόσθιας Τροφοδότησης με τον αλγόριθμο Ανάστροφης Μετάδοσης Σφάλματος (Back-Propagation)

Στα πολυεπίπεδα νευρωνικά δίκτυα, όπως και στην περίπτωση του απλού perceptron, η διαδικασία της εκπαίδευσης βασίζεται στην κατάλληλη προσαρμογή των βαρών του δικτύου, προκειμένου η τελική έξοδος να προσεγγίζει όσο τον δυνατόν περισσότερο την αναμενόμενη έξοδο. Ο αλγόριθμος εκπαίδευσης των πολυεπίπεδων ΤΝΔ στηρίζεται στην εξής ιδέα: έχοντας ορίσει μια αντικειμενική συνάρτηση που θα περιγράφει το σφάλμα μεταξύ της εξόδου του δικτύου και της αναμενόμενης εξόδου, προσπαθούμε να ελαχιστοποιήσουμε την συνάρτηση αυτή ως προς τα βάρη του δικτύου, προσαρμόζοντάς τα ανάλογα με το πόσο συνεισέφεραν στο συνολικό σφάλμα.

Η πιο γνωστή και σε ευρεία χρήση μέθοδος, για την εύρεση του ελάχιστου μιας συνάρτησης σφάλματος στα νευρωνικά δίκτυα, είναι η μέθοδος κατάβασης δυναμικού (gradient descent). Η μέθοδος αυτή, είναι ένας επαναληπτικός αλγόριθμος βελτιστοποίησης πρώτης τάξης, ο οποίος βασίζεται στην παρατήρηση ότι αν μια συνάρτηση πολλών μεταβλητών J(x) ορίζεται και είναι συνεχής στην περιοχή γύρω από ένα σημείο , τότε η συνάρτηση αυτή ελαχιστοποιείται γρηγορότερα προς την κατεύθυνση ελάττωσης της κλίσης της,

Για κάθε επίπ $-\nabla_{\boldsymbol{x}} J(\boldsymbol{x}) = -\left[\frac{\partial J(\boldsymbol{x})}{x_1}, \frac{\partial J(\boldsymbol{x})}{x_2}, \ldots, \frac{\partial J(\boldsymbol{x})}{x_n}\right]$

Για κ

Υπολόγισε τη έξοδο του νευρώνα: $\boldsymbol{a}_i^{(\cdot\cdot)} = \boldsymbol{f}_i^{(\cdot\cdot)}\left(\sum_{j=1}^{\cdot\cdot J_{K-1}} \boldsymbol{w}_{ij}^{(K)}\, \boldsymbol{a}_j^{(K-1)} + \boldsymbol{b}_i^{(K)}\right)$

Αποθήκευσε την στο διάνυσμα: $\left[\boldsymbol{a}_1^{(K)}, \boldsymbol{a}_2^{(K)}, \ldots, \boldsymbol{a}_{N_K}^{(K)}\right]$

Επέστρεψε ως έξοδο το διάνυσμα: $\left[\boldsymbol{a}_1^{(L)}, \boldsymbol{a}_2^{(L)}, \ldots, \boldsymbol{a}_{N_L}^{(L)}\right]$

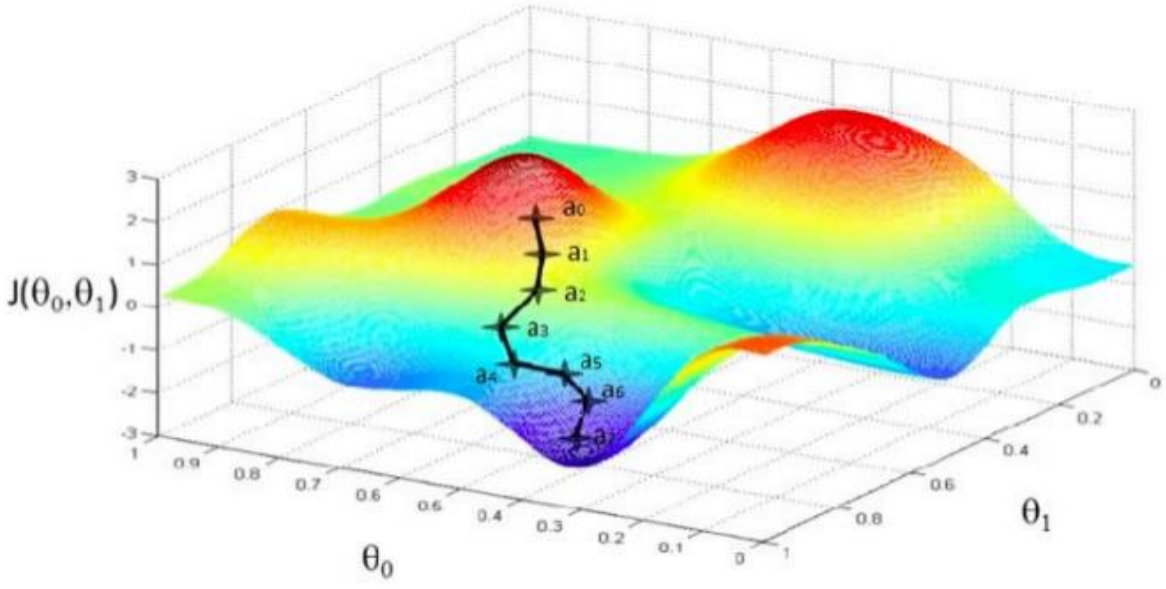
Εικόνα 18: Αναπαράσταση εύρεσης ελαχίστου με την μέθοδο κατάβασης δυναμικού

Δηλαδή, ακολουθώντας την αρνητική κλίση της επιφάνειας σφάλματος, κατευθυνόμαστε γρηγορότερα προς το ελάχιστο της. Σε κάθε βήμα του αλγορίθμου, το επόμενο σημείο προς την κατεύθυνση του ελαχίστου υπολογίζεται ως εξής:

$$\alpha_{n+1} = a_n - \eta \nabla J \ a_n$$

Με γ αρκετά μικρό έτσι ώστε : $J \ a_n \ \geq \ J \ a_{n+1}$

Στην περίπτωση των νευρωνικών οι μεταβλητές της συνάρτησης σφάλματος J



αποτελούν τα βάρη του δικτύου και επομένως η κλίσης της θα είναι:

$$\nabla_{\boldsymbol{W}} J = \frac{\partial J}{\partial \boldsymbol{W}} = \left(\frac{\partial J(\boldsymbol{W})}{w_1}, \frac{\partial J(\boldsymbol{W})}{w_2}, \ldots, \frac{\partial J(\boldsymbol{W})}{w_n} \right)$$

Παρ' όλα αυτά η μέθοδος αυτή δεν μπορεί να εφαρμοστεί αυτούσια σε δίκτυα τα οποία έχουν κρυφά επίπεδα, καθώς δεν είναι γνωστή η επιθυμητή έξοδος σε κάθε νευρώνα των εσωτερικών επιπέδων του δικτύου, παρά μόνο στο επίπεδο εξόδου. Η λύση για το πρόβλημα αυτό, δίνεται από τον αλγόριθμο ανάστροφης μετάδοσης σφάλματος (back propagation), τον πιο γνωστό αλγόριθμο εκπαίδευσης τεχνητών νευρωνικών δικτύων και ο οποίος βασίζεται στην μέθοδο κατάβασης δυναμικού, επιτρέποντας όμως να καθοριστεί το ποσοστό του συνολικού σφάλματος που αντιστοιχεί στα βάρη κάθε νευρώνα, ακόμα και αν αυτός βρίσκεται σε κρυφό επίπεδο. Παρακάτω διατυπώνεται μαθηματικά και περιγράφεται ο αλγόριθμος ανάστροφης μετάδοσης σφάλματος.

Αλγόριθμος Ανάστροφης Μετάδοσης Σφάλματος

Δίνεται ένα νευρωνικό δίκτυο perceptron πολλών επιπέδων, με αριθμό επιπέδων K και ένα σύνολο δεδομένων εισόδου και επιθυμητής εξόδου (σύνολο δεδομένων εκπαίδευσης) πάνω στο οποίο το νευρωνικό δίκτυο θα εκπαιδευτεί:

$$(\boldsymbol{x}^{1}, \boldsymbol{d}^{1}), (\boldsymbol{x}^{2}, \boldsymbol{d}^{2}), \ldots, (\boldsymbol{x}^{P}, \boldsymbol{d}^{P})$$

όπου, $\boldsymbol{x}^{(i)} = \left[x_1^i, x_2^i, \ldots, x_n^i\right], \quad \boldsymbol{d}^{(i)} = \left[d_1^i, d_2^i, \ldots, d_m^i\right]$

Χρησιμοποιώντας ως συνάρτηση σφάλματος, την συνάρτηση summed squared error (SSE) μεταξύ εξόδου του δικτύου και αναμενόμενης εξόδου, η αντικειμενική συνάρτηση που προκύπτει είναι:

$$J_{tot} = \frac{1}{2}\sum_{i=1}^{p}\|\boldsymbol{d}^{i} - \boldsymbol{y}^{i}\|^2 = \frac{1}{2}\sum_{i=1}^{P}\sum_{j=1}^{m}\left(d_j^i - y_j^i\right)^2 \quad (1)$$

όπου, $\boldsymbol{y} = [y_1, y_2, \ldots, y_m] = [a_1^{(K)}, a_2^{(K)}, \ldots, a_m^{(K)}] = \boldsymbol{a}^{K}$

η έξοδος του δικτύου που αποτελείται από τις ενεργοποιήσεις των νευρώνων του επιπέδου εξόδου.

Για τον καθορισμό της συνεισφοράς του επιπέδου εξόδου στο συνολικό σφάλμα:

Υπολογίζουμε μεταβολή του συνολικού σφάλματος ως προς τις τιμές των βαρών των ενώσεων μεταξύ του επιπέδου εξόδου και του προηγούμενου επιπέδου:

$$\nabla J_{tot} = \frac{\partial J_{tot}}{\partial \boldsymbol{W}^{(K)}} = \frac{\partial}{\partial \boldsymbol{W}^{(K)}}\left(\frac{1}{2}\sum_{z=1}^{p}\|\boldsymbol{d}^{z} - \boldsymbol{y}^{z}\|^2\right) = \frac{1}{2}\sum_{z=1}^{p}\frac{\partial(\boldsymbol{d}^{z} - \boldsymbol{y}^{z})^2}{\partial \boldsymbol{W}^{(K)}} \quad (2)$$

όπου $\mathbf{W}^{(K)}$ συμβολίζουμε το διάνυσμα βαρών που χαρακτηρίζουν τις συνδέσεις του επιπέδου K - 1 με το επιπέδου εξόδου K (επίπεδο εξόδου). Αν επικεντρωθούμε μόνο σε ένα πρότυπο του συνόλου δεδομένων εκπαίδευσης, τότε το σφάλμα εκεί είναι:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(K)}} = \frac{1}{2}\frac{\partial \|\boldsymbol{d} - \boldsymbol{y}\|^2}{\partial \boldsymbol{W}^{(K)}} = -\|\boldsymbol{d} - \boldsymbol{y}\| \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{W}^{K}} = -(\boldsymbol{d} - \boldsymbol{a}^{K})\frac{\partial \boldsymbol{a}^{K}}{\partial \boldsymbol{W}^{K}}$$

Η ενεργοποίηση όμως ενός νευρώνα i στο επίπεδο K είναι:

$$a_i^{K} = f\left(u_i^{K}\right),$$

$$u_i^{K} = \sum_{v=1}^{m} w_{iv}^{K}\, a_v^{K-1} + b_i^{(K)}$$

Όπου $w_{ij}^{(K)}$ το βάρος σύνδεσης του νευρώνα j του στρώματος K – 1 με το νευρώνα i του στρώματος K. Έτσι χρησιμοποιώντας τον κανόνα της αλυσίδας προκύπτει:

$$\frac{\partial J}{\partial \boldsymbol{W}^{K}} = (\boldsymbol{d} - \boldsymbol{a}^{K})\frac{\partial \boldsymbol{a}^{K}}{\partial \boldsymbol{W}^{K}} = (\boldsymbol{d} - \boldsymbol{a}^{K})\frac{\partial \boldsymbol{a}^{K}}{\partial \boldsymbol{u}^{K}}\frac{\partial \boldsymbol{u}^{K}}{\partial \boldsymbol{W}^{K}} = (\boldsymbol{d} - \boldsymbol{a}^{K})\frac{\partial f(\boldsymbol{u}^{K})}{\partial \boldsymbol{u}^{K}}\frac{\partial \boldsymbol{u}^{K}}{\partial \boldsymbol{W}^{K}}$$

$$= (\boldsymbol{d} - \boldsymbol{a}^{K}) f'(\boldsymbol{u}^{(K)}) \boldsymbol{a}^{K-1}$$

Και τελικά,

$$\frac{\partial J}{\partial \boldsymbol{W}^{K}} = \boldsymbol{\delta}^{K}\, \boldsymbol{a}^{K-1}, \qquad \mu\varepsilon \quad \boldsymbol{\delta}^{K} = (\boldsymbol{d} - \boldsymbol{a}^{K}) f'\left(\boldsymbol{u}_{i}^{K}\right) \qquad (3)$$

Όπου $\boldsymbol{\delta}^{K} = (\boldsymbol{d} - \boldsymbol{a}^{K}) f'\left(\boldsymbol{u}_{i}^{K}\right)$

ονομάζεται παράγοντας δ και ουσιαστικά αποτελεί τον ρυθμό μεταβολής του σφάλματος ως προς την είσοδο των νευρώνων του επιπέδου K. Από την (1),
αθροίζοντας τα $\frac{\partial J}{\partial \boldsymbol{W}^{K}}$
για κάθε πρότυπο του συνόλου δεδομένων εκπαίδευσης, παίρνουμε την συνεισφορά του επιπέδου εξόδου στο συνολικό σφάλμα.

Για τον καθορισμό της συνεισφοράς των κρυφών επιπέδων στο συνολικό σφάλμα:

Όπως και προηγουμένως υπολογίζουμε μεταβολή του συνολικού σφάλματος ως προς τις τιμές των βαρών, τα οποία όμως τώρα αφορούν τις ενώσεις μεταξύ των κρυφών επιπέδων. Έτσι με παρόμοιο τρόπο όπως και πριν και με χρήση του κανόνα της αλυσίδας, για τα βάρη $\mathbf{W}^{K-1}$ μεταξύ των επιπέδων K-1 και K έχουμε :

$$\frac{\partial J}{\partial \boldsymbol{W}^{K-1}} = (\boldsymbol{d} - \boldsymbol{a}^{K})\frac{\partial \boldsymbol{a}^{K}}{\partial \boldsymbol{W}^{K-1}} = (\boldsymbol{d} - \boldsymbol{a}^{K})\frac{\partial \boldsymbol{a}^{K}}{\partial \boldsymbol{u}^{K}}\frac{\partial \boldsymbol{u}^{K}}{\partial \boldsymbol{W}^{K-1}} =$$

$$= (\boldsymbol{d} - \boldsymbol{a}^{K}) f'(\boldsymbol{u}^{K})\frac{\partial \boldsymbol{u}^{K}}{\partial \boldsymbol{W}^{K-1}} = \boldsymbol{\delta}^{K}\frac{\partial \boldsymbol{u}^{K}}{\partial \boldsymbol{a}^{K-1}}\frac{\partial \boldsymbol{a}^{K-1}}{\partial \boldsymbol{W}^{K-1}} = \boldsymbol{\delta}^{K}\frac{\partial \boldsymbol{u}^{K}}{\partial \boldsymbol{a}^{K-1}}\frac{\partial \boldsymbol{a}^{K-1}}{\partial \boldsymbol{u}^{K-1}}\frac{\partial \boldsymbol{u}^{K-1}}{\partial \boldsymbol{W}^{K-1}}$$

$$= \boldsymbol{\delta}^{K}\boldsymbol{W}^{K} f'(\boldsymbol{u}^{(K-1)})\frac{\partial \boldsymbol{u}^{K-1}}{\partial \boldsymbol{W}^{K-1}} = \boldsymbol{\delta}^{K}\boldsymbol{W}^{K} f'l(\boldsymbol{u}^{(K-1)})\boldsymbol{a}^{K-2}$$

Και τελικά,

1. Αρχικοποίηση του δικτύου με τυχαία βάρη
2. Για κάθε εποχή εκπαίδευσης επανέλαβε:
   2.1. Για κάθε πρότυπο εκπαίδευσης από 1 έως P επανέλαβε:
      2.1.1. Υπολόγισε τις εξόδους-ενεργοποιήσεις κάθε επιπέδου του δικτύου
      2.1.2. Υπολόγισε τους παράγοντες δ για τους m νευρώνες του επιπέδου εξόδου (K) με βάση την (3)
      2.1.3. Ανάστροφα, για κάθε κρυφό επίπεδο του δικτύου από K-1 ως 1 επανέλαβε:
         2.1.3.1. Υπολόγισε τους παράγοντες δέλτα για όλους τους νευρώνες του κρυφού επιπέδου, με βάση την (4).
      2.1.4. Ανανέωσε όλα τα βάρη του δικτύου με βάση την (5)
   2.2. Υπολόγισε το σφάλμα εξόδου
   2.3. Αν το σφάλμα είναι μικρότερο από ε επέστρεψε τα βάρη του δικτύου.
   2.4. Αν η εποχή είναι η τελευταία επέστρεψε το σφάλμα.

Η παραπάνω σχέση $w_{ij}^{l}(k+1) = w_{ij}^{l}(k) - \gamma \frac{\partial J}{\partial w_{ij}^{l}(k)}$ (5) αφορά τον καθορισμό της συνεισφοράς των κρυφών επιπέδων στο σφάλμα του δικτύου και επομένως εφαρμόζεται ανάλογα και για τα βάρη $\mathbf{W}^{K-2}$, $\mathbf{W}^{K-3}$ κ.ο.κ.

Από τις σχέσεις (3), (4) πλέον γνωρίζουμε τον τρόπο με τον οποίο μπορεί να υπολογιστεί η μεταβολή του σφάλματος σε σχέση με κάθε βάρος του νευρωνικού δικτύου και επομένως με χρήση της μεθόδου κατάβασης δυναμικού μπορούμε να μεταβάλλουμε τα βάρη με τέτοιον τρόπο ώστε η συνάρτηση σφάλματος να ελαχιστοποιείται. Έτσι, η μεταβολή των βαρών προς την κατεύθυνση αρνητικής κλίσης είναι:

Όπου $w_{ij}^{l}(k)$ είναι το βάρος σύνδεσης του νευρώνα j του επιπέδου l – 1 με τον νευρώνα i του επίπεδου l στην χρονική στιγμή (επανάληψη) και ένας αριθμός αρκετά μικρός που ονομάζεται ρυθμός μάθησης.

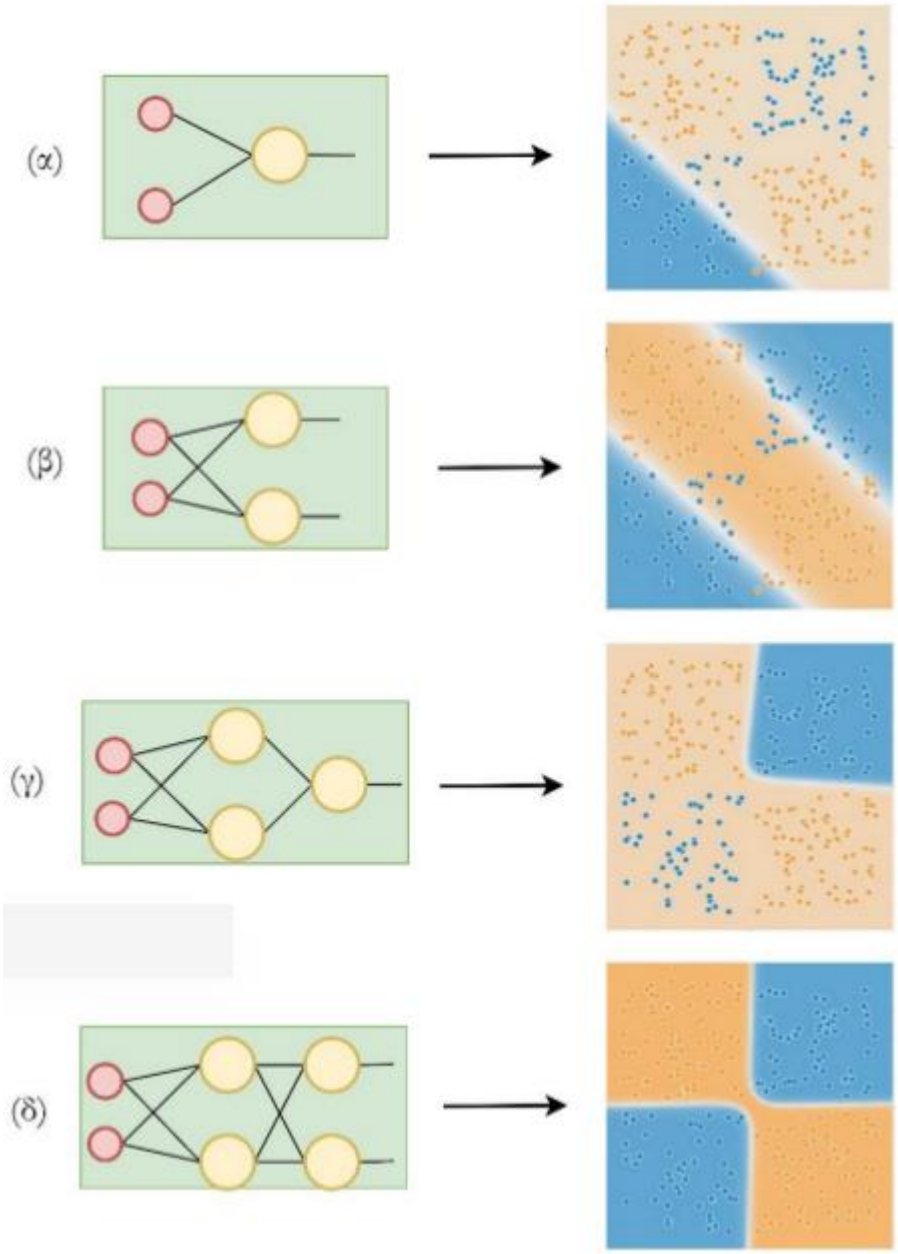
Με βάση τα παραπάνω ο αλγόριθμος ανάστροφης μετάδοσης σφάλματος διατυπώνεται ως εξής:

Η προσπάθεια του αλγορίθμου ανάστροφης μετάδοσης σφάλματος για ελαχιστοποίηση, μπορεί να θεωρηθεί σαν μια αναζήτηση του ολικού ελαχίστου της συνάρτησης σφάλματος, η οποία έχει σαν παραμέτρους τις τιμές των βαρών. Η διόρθωση που γίνεται κάθε φορά προσπαθεί να ελαχιστοποιήσει το σφάλμα διαλέγοντας να κάνει εκείνες τις αλλαγές που φαίνεται να το μειώνουν τοπικά.

4.4.3.6 ΤΝΔ Πρόσθιας Τροφοδότησης Πολλών Επιπέδων ως καθολικοί προσεγγιστές

Σύμφωνα με το Θεώρημα Καθολικής Προσέγγισης που διατυπώνεται στην μαθηματική θεωρία των τεχνητών νευρωνικών δικτύων, ένα πολυεπίπεδο δίκτυο perceptron (MLP) με δύο επίπεδα (ένα κρυφό επίπεδο και το επίπεδο εξόδου) με πεπερασμένο αριθμό νευρώνων, είναι ικανό να προσεγγίσει οποιαδήποτε συνεχή συνάρτηση με οσοδήποτε μικρό σφάλμα, διατηρώντας μονάχα, ήπιους περιορισμούς σχετικά με τις συναρτήσεις ενεργοποίησης που χρησιμοποιεί οι οποίες θα πρέπει να είναι φραγμένες, μονότονα αύξουσες και συνεχείς π.χ. η σιγμοειδής συνάρτηση.

Στα πλαίσια του θεωρήματος αυτού, εξετάζουμε μια τυχαία κατανομή παραδειγμάτων, καθένα από τα οποία έχει πρότυπο δύο διαστάσεων (x_1, x_2) και ανήκει σε μία από τις δύο μη γραμμικά διαχωρίσιμες κλάσεις. Στην Εικόνα 19 αναπαριστούμε στο επίπεδο τα αποτελέσματα ταξινόμησης τεσσάρων διαφορετικών ΤΝΔ που έχουν εκπαιδευτεί πάνω σε αυτό το σύνολο δεδομένων, σχεδιάζοντας τις διαχωριστικές επιφάνειες που δημιουργεί το καθένα .



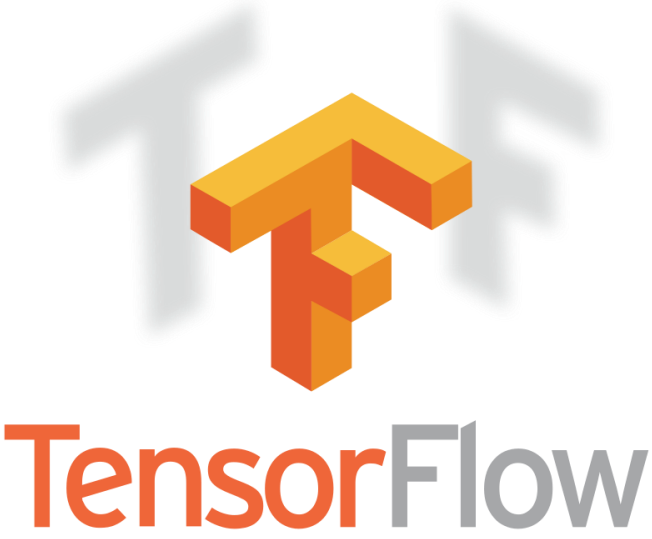
Εικόνα 19: Απεικόνιση διαχωριστικών επιφανειών που δημιουργούν πολυεπίπεδα δίκτυα perceptron με διαφορετικό πλήθος νευρώνων και κρυφών επιπέδων

ΚΕΦΑΛΑΙΟ 5:
5.Η ΠΛΑΤΦΟΡΜΑ TENSORFLOW ΚΑΙ Η ΚΑΤΑΝΟΜΗ ΠΟΡΩΝ ΣΕ ΑΥΤΗ

5.1 Εισαγωγή

Το TensorFlow είναι μία βιβλιοθήκη λογισμικού που έχει αναπτυχθεί από την Google Brain Team του ερευνητικού οργανισμού Machine Learning Intelligence της Google, με σκοπό να διευκολύνει την διεξαγωγή έρευνας και την υλοποίηση αλγορίθμων και συστημάτων μηχανικής μάθησης. Συνδυάζει την υπολογιστική άλγεβρα και τεχνικές βελτιστοποίησης για να εκτελεί εύκολα μαθηματικές εκφράσεις που σε άλλες περιπτώσεις θα απαιτούσαν πολύ μεγάλο χρόνο λόγω της υπολογιστικής τους πολυπλοκότητας. Η επίσημη ιστοσελίδα και το λογότυπο φαίνονται παρακάτω:

www.tensorflow.org



Εικόνα 20: Λογότυπο του Tensorflow

Το TensorFlow περιλαμβάνει τα παρακάτω χρήσιμα χαρακτηριστικά:

- Τον εύκολο ορισμό, την βελτιστοποίηση και τον αποδοτικό υπολογισμό μαθηματικών εκφράσεων που περιλαμβάνουν πίνακες υψηλών διαστάσεων (τανυστές, tensors).
- Την εύκολη κατασκευή και τον προγραμματισμό μοντέλων βαθιάς μάθησης και την άμεση χρήση τεχνικών μηχανικής μάθησης μέσω των προγραμματιστικών βιβλιοθηκών που υποστηρίζονται.
- Την εύκολη χρήση των μονάδων υπολογισμού γραφικών (GPU) για την επιτάχυνση των υπολογισμών καθώς και αυτοματοποίηση της διαχείρισης και βελτιστοποίηση της μνήμης και των δεδομένων που χρησιμοποιούνται. Επιπλέον παρέχεται η δυνατότητα ο ίδιος κώδικας να εκτελεστεί και σε άλλες συσκευές όπως προφανώς μονάδες επεξεργαστικής ισχύος (CPUs) αλλά και μονάδες επεξεργασίας τανυστών (TPUs – Tensor Processing Units), σε ολοκληρωμένες πλακέτες όπως το Raspberry Pi αλλά και σε συνδυασμό αυτών.
- Δυνατότητα υψηλής κλιμάκωσης των υπολογισμών σε πολλά μηχανήματα και υποστήριξη τεράστιων συνόλων δεδομένων.

Το Tensorflow οφείλει τη μεγάλη του δημοτικότητα στην άριστη υποστήριξη προς την πλευρά των χρηστών αλλά και στις πολλές έτοιμες βιβλιοθήκες μηχανικής μάθησης που περιλαμβάνει. Ακόμα συμπεριλαμβάνει έναν μεγάλο αριθμό από αλγορίθμους μηχανικής μάθησης και βαθιάς μάθησης. Οι χρήστες μπορούν να εκπαιδεύσουν και να τρέξουν βαθιά νευρωνικά δίκτυα για την ομαδοποίηση ψηφίων που έχουν γραφτεί με το χέρι, την αναγνώριση εικόνων, την ενσωμάτωση λέξεων και τη δημιουργία πολλών ακολουθιακών μοντέλων.

5.2 Προγραμματιστικό Μοντέλο

Το TensorFlow στον πυρήνα του είναι βασισμένο στο προγραμματιστικό πρότυπο της Ροής Δεδομένων (dataflow programming), σύμφωνα με το οποίο ένα πρόγραμμα μοντελοποιείται και αναπαρίσταται ως κατευθυνόμενος γράφος των δεδομένων που ρέουν μεταξύ πράξεων- λειτουργιών. Υιοθετώντας τον Γράφο Ροής Δεδομένων (Data Flow Graph) ως μοντέλο εκτέλεσης, καταφέρνει να διαχωρίσει την διαδικασία σχεδιασμού της ροής δεδομένων (κατασκευή γράφου και ροής) από την εκτέλεση του (CPU, GPU ή συνδυασμός τους), χρησιμοποιώντας μια μοναδική προγραμματιστική διεπαφή η οποία αποκρύπτει κάθε πολυπλοκότητα.

5.2.1 Γράφοι Ροής Δεδομένων (Data Flow Graphs)

Μια εφαρμογή μηχανικής μάθησης είναι το αποτέλεσμα επαναλαμβανόμενων υπολογισμών σύνθετων μαθηματικών εκφράσεων. Στο TensorFlow, ένας υπολογισμός περιγράφεται με την χρήση των Γράφων Ροής Δεδομένων, όπου κάθε κόμβος στον γράφο αναπαριστά ένα στιγμιότυπο μιας μαθηματικής πράξης και κάθε ακμή του γράφου αποτελεί

έναν πολυδιάστατο πίνακα-σύνολο δεδομένων (τανυστής) πάνω στον οποίο εφαρμόζονται οι μαθηματικές πράξεις αυτές. Πιο συγκεκριμένα, οι κόμβοι και οι ακμές διαχειρίζονται από το TensorFlow ως εξής:

- **Κόμβος (node):** Κάθε κόμβος αναπαριστά ένα στιγμιότυπο μιας μαθηματικής πράξης. Κάθε τέτοια πράξη έχει τουλάχιστον μία είσοδο και καμία ή περισσότερες εξόδους.
- **Ακμή (edge):** Στο TensorFlow οι ακμές είναι δύο τύπων:
  - *Απλές Ακμές:* Οι απλές ακμές μεταφέρουν τανυστές όπου η έξοδος μιας πράξης (κόμβος) γίνεται είσοδος σε επόμενη πράξη.
  - *Ειδικές Ακμές:* Οι ακμές αυτές δεν μεταφέρουν δεδομένα μεταξύ της εξόδου ενός κόμβου και της εισόδου κάποιου άλλου αλλά αντίθετα αναπαριστούν μια εξάρτηση ελέγχου μεταξύ δύο κόμβων. Αν υποθέσουμε ότι έχουμε δύο κόμβους Α, Β, η ειδική ακμή που τους συνδέει, υποδεικνύει ότι ο Β θα αρχίσει την λειτουργία του αφού τελειώσει ο Α την δικιά του. Έτσι, οι ακμές αυτές χρησιμοποιούνται στον Γράφο Ροής Δεδομένων για να δηλώσουν την εξάρτηση και ποια ενέργεια από δύο κόμβους εφαρμόστηκε στα δεδομένα (τανυστές) πρώτα.
- **Πράξη (Operation):** Η πράξη αντιπροσωπεύει έναν αφηρημένο υπολογισμό π.χ. άθροισμα, γινόμενο πινάκων κλπ. Μια πράξη διαχειρίζεται τανυστές και είναι πολυμορφική, δηλαδή μπορεί να δράσει πάνω σε τανυστές διαφορετικών τύπων. Για παράδειγμα, η πρόσθεση μεταξύ δύο τανυστών που έχουν ως στοιχεία ακεραίους των 32 bit (int32 tensors), μπορεί να εφαρμοστεί σε τανυστές με αριθμούς κινητής υποδιαστολής (float tensors) κ.ο.κ.
- **Πυρήνας (Kernel):** Ο πυρήνας καθορίζει την πραγματική υλοποίηση της πράξης, έχοντας διαφορετικές υλοποιήσεις (για την ίδια πράξη) ανάλογα με την συσκευή στην οποία θα εκτελεστεί (CPU, GPU κλπ.)
- **Συνεδρία (Session):** Κάθε φορά που ένα πρόγραμμα-πελάτης πρέπει να επικοινωνήσει με το TensorFlow, πρέπει να δημιουργηθεί μια συνεδρία μεταξύ αυτού του προγράμματος (συνήθως ένα πρόγραμμα σε γλώσσα Python ή σε όποια άλλη γλώσσα υποστηρίζεται αντίστοιχη διεπαφή) και του περιβάλλοντος λειτουργίας C++ του Tensorflow. Αφού η συνεδρία δημιουργηθεί για τον συγκεκριμένο πελάτη, δημιουργείται ένα αρχικός γράφος ο οποίος είναι άδειος. Η συνεδρία, που ουσιαστικά αποτελεί ένα αντικείμενο της Python (tf.Session), παρέχει πρόσβαση σε συσκευές στο τοπικό μηχάνημα καθώς και σε απομακρυσμένες συσκευές μέσω του κατανεμημένου περιβάλλοντος λειτουργίας του Tensorflow. Επιπλέον αποθηκεύει σε κρυφή μνήμη πληροφορίες σχετικά με τον Γράφο (αντικείμενο tf.Graph), παρέχοντας έτσι την δυνατότητα να τρέχουμε αποδοτικά τους ίδιους υπολογισμούς πολλές φορές.

5.2.2 Τα Πλεονεκτήματα των Γράφων Ροής Δεδομένων

Το TensorFlow βελτιστοποιεί τους υπολογισμούς του βάση της συνδεσιμότητας του γράφου. Κάθε γράφος διατηρεί το δικό του σύνολο εξαρτήσεων από κόμβους. Όταν η είσοδος του κόμβου Β επηρεάζεται από την έξοδο του κόμβου Α, λέμε ότι ο Β είναι εξαρτώμενος από τον κόμβο Α. Ονομάζουμε την εξάρτηση άμεση όταν οι δύο εξαρτώμενοι κόμβοι συνδέονται κατευθείαν με ακμή, ενώ έμμεση σε κάθε άλλη περίπτωση.

Το γεγονός ότι μπορούμε πάντα να αναγνωρίζουμε εύκολα τα σύνολα εξαρτήσεων για κάθε κόμβο σε έναν γράφο, αποτελεί βασικό χαρακτηριστικό για το πώς θα δομήσουμε τους υπολογισμούς βάση γράφου. Η δυνατότητα εντοπισμού εξαρτήσεων μεταξύ των

μονάδων του μοντέλου μας, μας επιτρέπει τόσο να διανείμουμε τους υπολογισμούς μας μεταξύ των διαθέσιμων πόρων όσο και να αποφεύγουμε την εκτέλεση περιττών υπολογισμών άσχετων υποσυνόλων, με αποτέλεσμα ταχύτερο, αποδοτικότερο και πιο αποτελεσματικό τρόπο υπολογισμού. Συνοπτικά τα πλεονεκτήματα της χρήσης Γράφου Ροής Δεδομένων, τα οποία εκμεταλλεύεται το TensorFlow κατά την εκτέλεση των προγραμμάτων είναι:

- **Παραλληλισμός:** Χρησιμοποιώντας σαφώς ορισμένες ακμές που αντιπροσωπεύουν εξαρτήσεις μεταξύ των λειτουργιών, είναι εύκολο για το σύστημα να εντοπίσει λειτουργίες που μπορούν να εκτελούνται παράλληλα.
- **Κατανεμημένη Εκτέλεση:** Χρησιμοποιώντας σαφώς ορισμένες ακμές που αντιπροσωπεύουν τις τιμές που ρέουν μεταξύ των λειτουργιών των κόμβων, είναι δυνατό για το TensorFlow να χωρίσει και να κατανείμει το πρόγραμμά σας σε πολλαπλές συσκευές (CPUs, GPUs κλπ.) συνδεδεμένες σε διαφορετικές μηχανές. Το TensorFlow εισάγει την απαραίτητη επικοινωνία και τον συντονισμό μεταξύ των συσκευών.
- **Μεταγλώττιση:** Ο μεταγλωττιστής XLA του TensorFlow μπορεί να χρησιμοποιήσει τις πληροφορίες από τον Γράφο Ροής Δεδομένων για να δημιουργήσει γρηγορότερο κώδικα, όπως για παράδειγμα, συγχωνεύοντας γειτονικές λειτουργίες.
- **Φορητότητα:** Ο Γράφος Ροής Δεδομένων είναι μια αναπαράσταση ανεξάρτητη από την γλώσσα του κώδικα που έχει χρησιμοποιηθεί για την δημιουργία του μοντέλου μας. Παρέχεται έτσι η δυνατότητα να δημιουργήσουμε έναν Γράφο Ροής Δεδομένων στη Python και στην συνέχεια να το αποθηκεύσουμε και να το επαναφέρουμε σε ένα πρόγραμμα C ++.

5.3 Τανυστές, Γράφοι, Συνεδρίες

Όταν κατασκευάζουμε έναν κόμβο του γράφου, στην πραγματικότητα δημιουργούμε ένα στιγμιότυπο μιας πράξης. Αυτές οι πράξεις δεν παράγουν πραγματικές τιμές (μέχρι την στιγμή που ο γράφος θα εκτελεστεί), αλλά αναφορές στα μελλοντικά αποτελέσματα που πρόκειται να υπολογίσουν, τις οποίες παρέχουν σαν «σκυτάλη» στην είσοδο των επόμενων κόμβων. Το TensorFlow έχει σχεδιαστεί έτσι ώστε να δημιουργείται ο αρχικός σκελετός του γράφου με όλα τα συστατικά του και μέχρι το σημείο εκείνο να μην ρέει καμία πληροφορία ή να μην εκτελείται κανένας υπολογισμός. Μόνο κατά την εκτέλεση, όταν δημιουργούμε τη συνεδρία, τα δεδομένα εισέρχονται στο γράφημα ως τανυστές και πραγματοποιούνται οι υπολογισμοί. Με αυτόν τον τρόπο, οι υπολογισμοί μπορούν να είναι πολύ πιο αποδοτικοί, λαμβάνοντας υπόψη ολόκληρη τη δομή γραφημάτων.

Σε αυτήν την ενότητα θα περιγράψουμε και θα εξηγήσουμε του τανυστές και στην συνέχεια θα δημιουργήσουμε έναν απλό γράφο που υλοποιεί μια στοιχειώδη αριθμητική παράσταση και θα τον εκτελέσουμε. Η υλοποίηση θα γίνει στην γλώσσα Python, η οποία αποτελεί και την γλώσσα με την πιο δημοφιλή και πλούσια προγραμματιστική διεπαφή (API) όσον αφορά το TensorFlow.

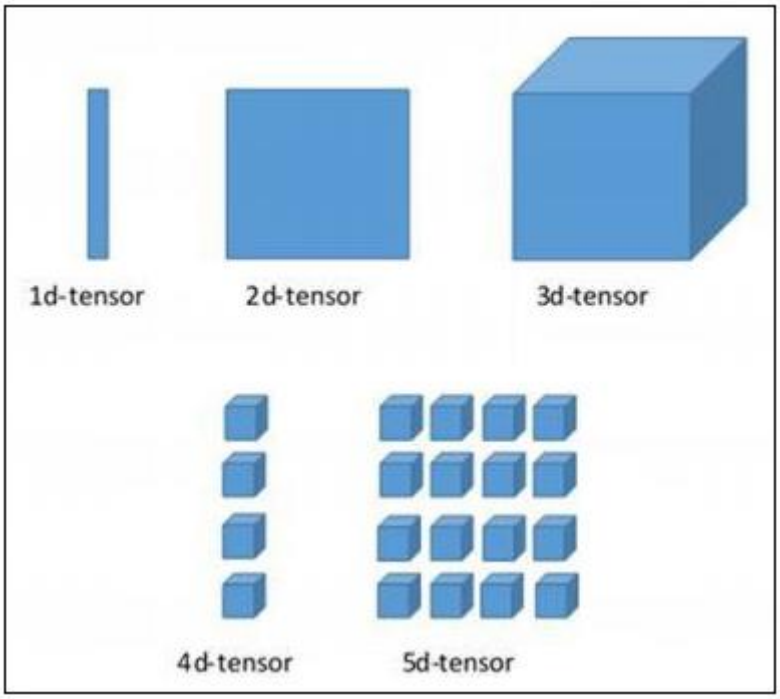
5.3.1 Τανυστές (Tensors)

Οι τανυστές είναι η βασική δομή δεδομένων στο Tensorflow. Όπως έχει ήδη αναφερθεί, οι τανυστές αναπαριστούν τις ακμές συνδέσεων στο Γράφο Ροής Δεδομένων ή διαφορετικά, την πληροφορία που ρέει μέσα στις ακμές. Ένας τανυστής ουσιαστικά

αποτελεί έναν πολυδιάστατο πίνακα ή λίστα και προσδιορίζεται από τρία βασικά χαρακτηριστικά: τον βαθμό (rank), το σχήμα (shape) και τον τύπο (type).

- Η διάσταση κάθε τανυστή περιγράφεται από μια μονάδα η οποία ονομάζεται **βαθμός (rank)**. Έτσι ένας απλός δισδιάστατος πίνακας, είναι ένας τανυστής με βαθμό 2, ενώ ένα διάνυσμα είναι ένα τανυστής με βαθμό 1.
- Το **σχήμα (shape)** ενός τανυστή περιγράφει το μέγεθος της κάθε διάτασης του. Για παράδειγμα, σε έναν πίνακα 2 διαστάσεων, δηλώνει το πλήθος των γραμμών και των στηλών που έχει ο πίνακας αυτός.
- Τέλος, ο **τύπος (type)** του τανυστή, αποτελεί τον τύπο των στοιχείων (int32, float κλπ.) που περιέχει ο τανυστής.

Παρακάτω φαίνεται μία γραφική απεικόνιση των τανυστών από μία έως πέντε διαστάσεις:



Εικόνα 21: Γραφική Απεικόνιση Τανυστών

5.3.2 Σταθερές, Μεταβλητές και Placeholders

Το TensorFlow προκειμένου να μπορεί να εισάγει δεδομένα μέσα στον γράφο καθώς και να τα αποθηκεύσει χρησιμοποιεί τρείς τύπους κόμβων καθένας με διαφορετική λειτουργία και χρησιμότητα. Οι κόμβοι αυτοί είναι οι εξής:

- **tf.constant:** η λειτουργία του κόμβου αυτού είναι παρόμοια με την λειτουργία της σταθεράς σε ένα πρόγραμμα. Ο κόμβος αυτός κατά την δημιουργία του, αρχικοποιείται με μια τιμή η οποία παραμένει σταθερή από την αρχή μέχρι το τέλος λειτουργίας του γράφου.

  Ένα παράδειγμα δημιουργίας ενός κόμβου constant:

```
>>> a=tf.constant(<αρχική-τιμή>, dtype=<προαιρετικός τύπος>)
```

- **tf.placeholder:** η λειτουργία του κόμβου αυτού είναι να τροφοδοτεί δεδομένα στον γράφο. Για παράδειγμα, κατά την εκπαίδευση ενός νευρωνικού δικτύου, ο κόμβος αυτός έχει σαν ρόλο να δέχεται τα δεδομένα από το σύνολο εκπαίδευσης και να τα προωθεί μέσα στον γράφο. Αυτό επιτυγχάνεται καθώς όπως κάθε τανυστής, έτσι και ο κόμβος placeholder μπορεί να χρησιμοποιηθεί ως είσοδος σε επόμενο κόμβο. Κατά την κατασκευή του, θα πρέπει υποχρεωτικά να δηλώνεται ο τύπος των δεδομένων που δέχεται.

  ```
  >>> a = tf.placeholder(dtype=<τύπος_δεδομένων>)
  ```

- **tf.Variable:** Ο κόμβος αυτός, όπως και μια μεταβλητή σε ένα πρόγραμμα, αποθηκεύει και διατηρεί δεδομένα, τα οποία μπορούν να μεταβάλλονται κατά την διάρκεια εκτέλεσης του γράφου. Η προσθήκη μιας μεταβλητής στον γράφο γίνεται με την κατασκευή ενός στιγμιότυπου της κλάσης Variable του TensorFlow. Κατά την κατασκευή της, απαιτείται η αρχικοποίησης της με μία τιμή, η οποία μπορεί να είναι ένας τανυστής οποιουδήποτε τύπου και σχήματος.

  ```
  >>> a = tf.Variable(<αρχική-τιμή>, name=<προαιρετικό-όνομα>)
  ```

  Μετά την κατασκευή της, ο τύπος και το σχήμα της μένουν σταθερά και δεν μπορούν να μεταβληθούν. Η τιμή της μπορεί να αλλάξει χρησιμοποιώντας την μέθοδο assign ή κάποια παρόμοια μέθοδο. Όπως κάθε τανυστής, έτσι και οι μεταβλητές που έχουν κατασκευαστεί με την κλάση Variable, μπορούν να χρησιμοποιηθούν και ως είσοδοι σε επόμενους κόμβους.

5.3.3 Δημιουργία Γράφου

Πριν ξεκινήσουμε να δημιουργούμε οποιαδήποτε μορφή γράφου θα πρέπει πρώτα στο πρόγραμμα μας να εισάγουμε την βιβλιοθήκη του TensorFlow για την Python.

```
>>> import tensorflow as tf
```

Αμέσως μετά την εισαγωγή της βιβλιοθήκης, αρχικοποιείται ένας προκαθορισμένος γράφος, ο οποίος αρχικά είναι κενός. Ό,τι κόμβους δημιουργούμε στην συνέχεια θα συσχετίζονται αυτόματα και θα προστίθενται σε αυτόν τον γράφο.

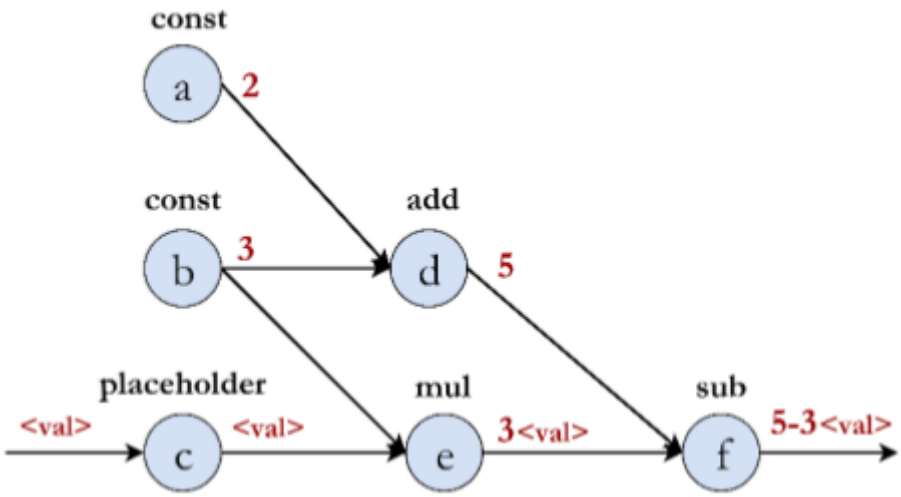
Αρχικά, με την βοήθεια των τύπων κόμβων που παρουσιάσαμε παραπάνω για την διαχείριση των δεδομένων στο γράφο, θα δημιουργήσουμε τρεις κόμβους στους οποίους θα αναθέσουμε τιμές και θα τους συνδέσουμε με επόμενους κόμβους που θα παρουσιαστούν παρακάτω:

```
>>> a = tf.constant(2, dtype=tf.int32)
>>> b = tf.constant(3, dtype=tf.int32)
>>> c = tf.placeholder(dtype=tf.int32, shape=[1])
```

Εδώ, οι κόμβοι a και b έχουν σταθερή τιμή, ενώ ο c είναι placeholder και θα λάβει την είσοδο του κατά την εκτέλεση του γράφου.

Στην συνέχεια, καθένας από τους επόμενους κόμβους παίρνει ως είσοδο την έξοδο κάποιου από τους προηγούμενους κόμβους, δομώντας έτσι έναν γράφο που αναπαριστά μια απλή αριθμητική παράσταση:

```
>>> d = tf.add(a,b)
>>> e = tf.multiply(b,c)
>>> f = tf.subtract(d,e)
```



Εικόνα 22: Παράδειγμα Γράφου στο TensorFlow

5.3.4 Δημιουργία και εκτέλεση Συνεδρίας

Μετά την ολοκλήρωση της περιγραφής και της κατασκευής του γράφου, είμαστε έτοιμοι να εκτελέσουμε του υπολογισμούς που αναπαρίστανται από αυτόν. Για τον σκοπό αυτό θα πρέπει να δημιουργήσουμε μια συνεδρία και στην συνέχεια να την εκτελέσουμε. Αυτό το επιτυγχάνουμε με τον παρακάτω κώδικα:

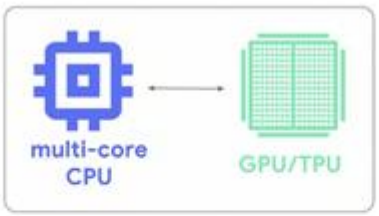
```
>>> sess = tf.Session()
>>> out = sess.run(f, {c:7})
>>> sess.close()
>>> print(out)
>>> -16
```

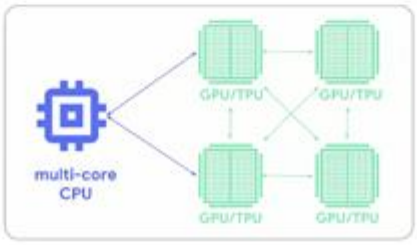
Ένα αντικείμενο Session είναι κομμάτι της προγραμματιστικής διεπαφής (API) του Tensorflow το οποίο δημιουργεί την επικοινωνία μεταξύ των αντικειμένων της Python μαζί

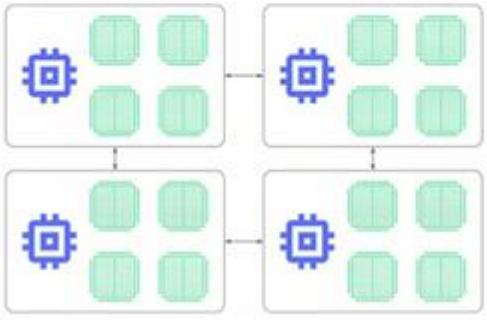
με τα δεδομένα που έχουμε από τη δικιά μας πλευρά και του πραγματικού υπολογιστικού συστήματος στο οποίο έχει δεσμευτεί η μνήμη για τα αντικείμενα που εμείς ορίσαμε, έχουν αποθηκευτεί ενδιάμεσες μεταβλητές και από το οποίο τελικά μας επιστρέφονται τα αποτελέσματα. Η συνεδρία αφού δημιουργηθεί, στην συνέχεια εκτελείται καλώντας την μέθοδο .run(). Εκεί είναι που εισάγεται ο κόμβος του γράφου του οποίου επιθυμούμε να υπολογίσουμε την έξοδο, καθώς και τα δεδομένα που θέλουμε να τροφοδοτήσουμε στον γράφο, στην περίπτωση που στο σύνολο εξαρτήσεων υπάρχει και κόμβος placeholder. Όταν κληθεί, η μέθοδος αυτή υπολογίζει ένα σύνολο υπολογισμών στο γράφο με τον ακόλουθο τρόπο: ξεκινάει από τον κόμβο του οποίου η έξοδος ζητήθηκε και στην συνέχεια με βάση τις εξαρτήσεις μεταξύ των κόμβων, ξεδιπλώνει προς τα πίσω τον γράφο καθορίζοντας το ποιος κόμβος πρέπει να υπολογιστεί. Στην περίπτωση μας, ζητήσαμε να υπολογιστεί ο κόμβος f όταν στον κόμβο c δίνουμε τον ακέραιο αριθμό 7 μέσω της εντολής: sess.run(f, {c:7}). Αφού ολοκληρωθούν οι υπολογισμοί μας, είναι καλή πρακτική να κλείνουμε την συνεδρία προκειμένου να αποδεσμεύουμε τους πόρους του συστήματος.

5.4 Κατανεμημένα Συστήματα στο TensorFlow

Το TensorFlow δίνει τη δυνατότητα στους χρήστες να εκτελούν τα μοντέλα και τους υπολογισμούς τους σε μία πληθώρα από συσκευές (CPUs, GPUs κλπ.). Οι χρήστες, μπορούν να εκτελέσουν τα προγράμματά τους σε ένα μηχάνημα με μία απλή υπολογιστική συσκευή, είτε σε ένα μηχάνημα με πολλαπλές υπολογιστικές συσκευές, είτε τελικά σε πολλαπλά μηχανήματα εφοδιασμένα με πολλαπλές υπολογιστικές συσκευές συνδεδεμένα σε ένα κοινό δίκτυο όπως φαίνεται στην παρακάτω εικόνα:







Εικόνα 23: Συσκευές και Μηχανήματα

Υπάρχει ένας μεγάλος αριθμός μεθόδων με τις οποίες μπορεί κανείς να κατανέμει την διαδικασία της εκπαίδευσης. Το είδος της προσέγγισης εξαρτάται από το μέγεθος του μοντέλου, την ποσότητα των δεδομένων εκπαίδευσης και φυσικά τις διαθέσιμες συσκευές. Οι βασικότερες θα αναπτυχθούν παρακάτω:

5.4.1 Παραλληλισμός Μοντέλου (Model Parallelism)

Σε αυτή τη στρατηγική (που είναι γνωστή και ως Παραλληλισμός Δικτύου) εκμεταλλευόμαστε το γεγονός ότι το μοντέλο μας είναι πολύ μεγάλο για να χωρέσει στη μνήμη ενός μηχανήματος και έτσι κατανέμουμε διαφορετικά τμήματα του γράφου σε διαφορετικές συσκευές για να τρέξουν παράλληλα. Διαφορετικοί παράμετροι θα "ζούνε" σε διαφορετικά μηχανήματα και οι διαδικασίες εκπαίδευσης και ενημέρωσης θα συμβαίνουν εκεί. Ένας βασικός τρόπος για να εφαρμόσουμε αυτή τη στρατηγική είναι να αναθέσουμε το πρώτο επίπεδο του μοντέλου στο πρώτο μηχάνημα, το δεύτερο στο δεύτερο κ.ο.κ. Υπάρχει όμως περίπτωση η συγκεκριμένη στρατηγική να μην είναι πάντα βέλτιστη καθώς σε πολυεπίπεδα μοντέλα, τα πιο "βαθιά" επίπεδα θα πρέπει να περιμένουν για τα αρχικά κατά τη διάρκεια της πρόσθιας τροφοδότησης (forward pass) και αντίστοιχα τα αρχικά επίπεδα θα πρέπει να περιμένουν τα "βαθιά" κατά τη διάρκεια της ανάστροφης μετάδοσης σφάλματος (backpropagation). Σε γενικές γραμμές είναι η πιο περίπλοκη στρατηγική.

5.4.2 Παραλληλισμός Δεδομένων (Data Parallelism)

Σε αυτή τη στρατηγική, τα δεδομένα εκπαίδευσης διαχωρίζονται σε πολλαπλά υποσύνολα και κάθε ένα από αυτά θα τρέξει στο ίδιο αντίγραφο μοντέλου σε διαφορετικό κόμβο – μηχάνημα κάθε φορά (κόμβος εργάτης – worker node). Το σύνολο αυτών των κόμβων θα πρέπει να συγχρονίσουν τις παραμέτρους τους στο τέλος της υπολογιστικής διαδικασίας έτσι ώστε η εκπαίδευση να μην διαφέρει από το αν έτρεχε σε μία μόνο συσκευή. Αυτό σημαίνει ότι κάθε συσκευή θα πρέπει να στείλει όλες τις αλλαγές σε όλα τα μοντέλα σε όλες τις υπόλοιπες συσκευές.

Μία σημαντική ιδιότητα της παραπάνω στρατηγικής είναι ότι βελτιώνεται ανάλογα με το μέγεθος των διαθέσιμων δεδομένων εκπαίδευσης και μία δεύτερη είναι ότι απαιτείται λιγότερη επικοινωνία μεταξύ των κόμβων, καθώς επωφελείται από τον υψηλό αριθμό υπολογισμών για τα βάρη. Ένα σημαντικό μειονέκτημα είναι πως το μοντέλο θα πρέπει να "χωράει" σε κάθε κόμβο με αποτέλεσμα να μην μπορούμε να χρησιμοποιήσουμε τη συγκεκριμένη διαδικασία σε πολύ μεγάλα μοντέλα.

Παρακάτω φαίνεται σχηματικά η στρατηγική αυτή:

Εικόνα 24: Παραλληλισμός Δεδομένων

Όσον αφορά τη διαδικασία ενημέρωσης του μοντέλου, μπορούν να χρησιμοποιηθούν δύο κοινές προσεγγίσεις, η σύγχρονη και η ασύγχρονη κατανεμημένη εκπαίδευση που περιγράφονται παρακάτω:

5.4.2.1 Ασύγχρονη Εκπαίδευση (Asynchronous Training)

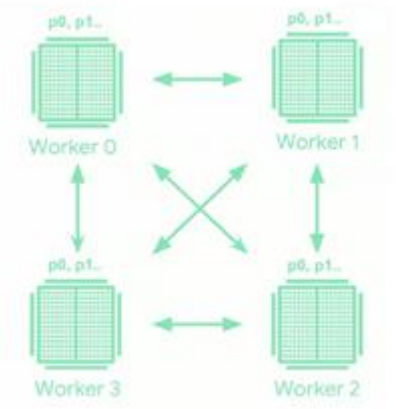
Σε αυτή την προσέγγιση ορίζουμε κάποια μηχανήματα ως Κόμβους Παραμέτρων (Parameter Servers) (στην παρακάτω εικόνα φαίνονται με μπλε χρώμα), τα οποία αποθηκεύουν τις παραμέτρους του μοντέλου. Τα υπόλοιπα μηχανήματα ορίζονται ως Εργάτες (Workers) και έχουν στη δικαιοδοσία τους το φόρτο της υπολογιστικής διαδικασίας. Κάθε ένας Worker "κατεβάζει" τις παραμέτρους του μοντέλου από τους Parameter Servers και στη συνέχεια υπολογίζει τη συνάρτηση σφάλματος (loss function) και τους παράγοντες δ (gradients) και στη συνέχεια τα στέλνει πίσω στους Parameter Servers οι οποίοι με τη σειρά τους ενημερώνουν τις παραμέτρους του μοντέλου. Ο κάθε Worker εκτελεί την παραπάνω διαδικασία ανεξάρτητα, με αποτέλεσμα η παρούσα προσέγγιση να ενδείκνυται για εκτέλεση σε έναν μεγάλο αριθμό από Workers. Το μειονέκτημα της διαδικασίας αυτής είναι προφανώς ότι οι Workers δεν μπορούν να είναι πάντα συγχρονισμένοι με αποτέλεσμα να αργεί η σύγκλιση του αλγορίθμου.

Εικόνα 25: Ασύγχρονη Εκπαίδευση

5.4.2.2 Σύγχρονη Εκπαίδευση (Synchronous Training)

Σε αυτή την προσέγγιση, ο κάθε Κόμβος Εργάτης (Worker node) κρατάει ένα αντίγραφο των παραμέτρων για τον ίδιο με αποτέλεσμα να μην χρειάζονται πλέον Κόμβοι Παραμέτρων (Parameter Servers). Ο κάθε Κόμβος Εργάτης υπολογίζει με τη σειρά του τους παράγοντες δ (gradients), βασισμένος σε ένα υποσύνολο των αρχικών δεδομένων εκπαίδευσης και μόλις τελειώσει με τους υπολογισμούς, συνεννοείται με τους υπόλοιπους

ώστε όλοι μαζί να ενημερώσουν τις παραμέτρους του μοντέλου. Όλοι οι Κόμβοι Εργάτες είναι συγχρονισμένοι, που σημαίνει ότι ο νέος γύρος εκπαίδευσης δεν μπορεί να ξεκινήσει μέχρι κάθε εργάτης να έχει λάβει τα ανανεωμένα gradients και να έχει ενημερώσει το μοντέλο του. Σύμφωνα με τα παραπάνω, η παρούσα προσέγγιση ενδείκνυται για γρήγορη εκπαίδευση σε ένα μηχάνημα με πολλαπλές συσκευές ή σε ένα μικρό αριθμό μηχανημάτων. Σχηματικά η σύγχρονη εκπαίδευση φαίνεται παρακάτω:



Εικόνα 26: Σύγχρονη Εκπαίδευση

5.5 Το DistributionStrategy API

Όπως είναι προφανές, το TensorFflow έχει αναπτύξει ένα διαθέσιμο API με το οποίο οι χρήστες μπορούν να κατανέμουν την εκπαίδευση ενός μοντέλου σε πολλαπλά μηχανήματα και συσκευές. Τα βασικά του πλεονεκτήματα είναι:

- Η ευκολία στη χρήση
- Η καλή απόδοση
- Η εύκολη εναλλαγή μεταξύ των διαφορετικών υποστηριζόμενων στρατηγικών

Το παραπάνω API μπορεί να χρησιμοποιηθεί παράλληλα με άλλα υψηλού επιπέδου APIs, όπως για παράδειγμα το Keras. Οι χρήστες από την πλευρά τους αρκεί να κάνουν μερικές προσθήκες στα ήδη υπάρχοντα προγράμματά τους έτσι ώστε να επωφεληθούν από αυτό.

5.5.1 Διαθέσιμες Στρατηγικές

Mirrored Strategy

Η παραπάνω στρατηγική υποστηρίζει σύγχρονη κατανεμημένη εκπαίδευση σε πολλαπλές συσκευές σε ένα μόνο μηχάνημα (συνήθως πολλαπλές μονάδες επεξεργασίας γραφικών – GPUs). Δημιουργεί ένα αντίγραφο του μοντέλου για κάθε συσκευή. Η κάθε μεταβλητή στο μοντέλο "καθρεπτίζεται" σε όλα τα αντίγραφα του μοντέλου και μαζί όλες οι μεταβλητές δομούν μία ολική μεταβλητή που ονομάζεται MirroredVariable. Αυτές οι μεταβλητές συγχρονίζονται μεταξύ τους με ταυτόχρονες ενημερώσεις.

Η ενημέρωση των μεταβλητών μεταξύ των συσκευών πραγματοποιείται με τη χρήση αποδοτικών αλγορίθμων all – reduce. Οι αλγόριθμοι αυτοί συγκεντρώνουν τους τανυστές και τους έχουν διαθέσιμους για κάθε συσκευή. Ανάλογα με τον τύπο της επικοινωνίας μεταξύ των συσκευών μπορούν να χρησιμοποιηθούν διαφορετικοί αλγόριθμοι, αλλά η προεπιλογή όσον αφορά τις GPUs είναι η βιβλιοθήκη NVIDIA NCCL σαν εκτέλεση all – reduce.

Ο πιο απλός τρόπος για να εισάγει κανείς την παραπάνω στρατηγική στον κώδικά του φαίνεται παρακάτω:

```
mirrored_strategy = tf.distribute.MirroredStrategy()
```

Η παραπάνω γραμμή κώδικα θα δημιουργήσει ένα στιγμιότυπο της στρατηγικής MirroredStrategy το οποίο θα χρησιμοποιήσει όλες τις διαθέσιμες GPUs που είναι ορατές στο TensorFlow και θα εφαρμόσει το NCCL για την επικοινωνία μεταξύ των συσκευών.

Multi Worker Mirrored Strategy

Η παραπάνω στρατηγική αποτελεί ένα υπερσύνολο της προηγούμενης. Υλοποιεί σύγχρονη κατανεμημένη εκπαίδευση μεταξύ πολλαπλών εργατών (worker nodes) και ο κάθε ένας από αυτούς μπορεί να έχει πολλαπλές συσκευές GPU. Όμοια με τα προηγούμενα, δημιουργεί αντίγραφα όλων των μεταβλητών σε κάθε συσκευή μεταξύ όλων των εργατών.

Για να κρατάει τις μεταβλητές συγχρονισμένες χρησιμοποιεί CollectiveOps ως την all – reduce μέθοδο επικοινωνίας σε πολλαπλούς εργάτες. Ένα collective op είναι μία απλή πράξη στον γράφο του TensorFlow η οποία μπορεί να επιλέξει αυτόματα έναν all – reduce αλγόριθμο στο περιβάλλον του TensorFlow με βάση το υλικό, την τοπολογία δικτύου και το μέγεθος των τανυστών.

Ο πιο απλός τρόπος για να εφαρμόσουμε αυτή τη στρατηγική είναι με τον παρακάτω κώδικα:

```
multiworker_strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
```

Μία βασική διαφορά για τη δημιουργία της παραπάνω στρατηγικής, σε αντίθεση με την εκπαίδευση σε πολλαπλές GPUs σε ένα μηχάνημα , είναι η οργάνωση των πολλαπλών Εργατών. Για το λόγο αυτό χρησιμοποιείται η μεταβλητή περιβάλλοντος TF_CONFIG ως ρυθμιστής για την διαμόρφωση του cluster.

Central Storage Strategy

Η παραπάνω στρατηγική υποστηρίζει επίσης τη σύγχρονη κατανεμημένη εκπαίδευση. Οι μεταβλητές δεν "καθρεπτίζονται" πλέον, αλλά αντίθετα αποθηκεύονται στην CPU και οι πράξεις "καθρεπτίζονται" σε όλες τις ορατές GPUs. Εάν υπάρχει μόνο μία

GPU όλες οι μεταβλητές και οι πράξεις θα αποθηκευτούν σε αυτή. Οι χρήστες μπορούν να εισάγουν την παραπάνω στρατηγική στον κώδικα τους ως εξής:

```
central_storage_strategy = tf.distribute.experimental.CentralStorageStrategy()
```

Parameter Server Strategy

Σε αυτή τη μέθοδο, υποστηρίζονται οι κόμβοι παραμέτρων (parameter servers) για την εκπαίδευση σε πολλαπλά μηχανήματα. Κάποια μηχανήματα ορίζονται ως εργάτες (workers) και κάποια ως parameter servers. Οι υπολογισμοί κατανέμονται σε όλες τις GPUs όλων των εργατών.

Σε κώδικα μοιάζει με τις υπόλοιπες στρατηγικές:

```
ps_strategy = tf.distribute.experimental.ParameterServerStrategy()
```

TPU Strategy

Η τελευταία υποστηριζόμενη μέθοδος επιτρέπει την εκπαίδευση σε Μονάδες Επεξεργασίας Τανυστών (TPUs), δηλαδή εξειδικευμένα ASICs, σχεδιασμένα από την Google για την επιτάχυνση των εφαρμογών μηχανικής μάθησης. Η μέθοδος υποστηρίζει σύγχρονη κατανεμημένη εκπαίδευση και μπορούμε να την εισάγουμε στον κώδικα μας με τον παρακάτω τρόπο:

```
cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(
    tpu=tpu_address)
tf.config.experimental_connect_to_cluster(cluster_resolver)
tf.tpu.experimental.initialize_tpu_system(cluster_resolver)
tpu_strategy = tf.distribute.experimental.TPUStrategy(cluster_resolver)
```

Απαιτείται βέβαια η πρόσβαση σε μονάδες TPU συνήθως μέσω της πλατφόρμας Google Colab ή του TensorFlow Research Cloud ή μέσω Cloud TPUs.

ΚΕΦΑΛΑΙΟ 6:
6.ΠΡΟΣΕΓΓΙΣΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ

6.1 Αποδοτική Κατανομή Πόρων

Το πρόβλημα που καλούμαστε να λύσουμε και το οποίο βασίζεται στο θεωρητικό υπόβαθρο που αναπτύχθηκε στα προηγούμενα κεφάλαια, είναι η αποδοτικότερη κατανομή των διαθέσιμων πόρων ανάλογα με την εργασία που πρέπει να εκτελεστεί. Οι διαθέσιμοι πόροι, βρίσκονται σε απομακρυσμένα μηχανήματα στα οποία έχουμε πρόσβαση μέσω του πρωτοκόλλου SSH. Τα στοιχεία των μηχανημάτων όπως η διεύθυνση IP, το username, ο κωδικός πρόσβασης και οι διαθέσιμοι πόροι καταγράφονται από τους χρήστες σε ένα ειδικό JSON αρχείο που καλείται blueprint. Με βάση αυτό το αρχείο δημιουργούμε ένα tensorflow cluster το οποίο περιέχει τα μηχανήματα που είναι κατάλληλα για την εργασία που θέλουμε να κατανέμουμε σε αυτά. Τέλος, επιλέγουμε την κατάλληλη στρατηγική που μας ενδιαφέρει και κατανέμουμε κατόπιν την εργασία στους διαθέσιμους πόρους του cluster.

6.2 Διαθέσιμα Μηχανήματα και Blueprint

Τα διαθέσιμα μηχανήματα που έχουμε πρόσβαση μπορεί να έχουν διαφορετικούς διαθέσιμους πόρους και διαφορετικές δυνατότητες. Μπορεί να υπάρχουν μηχανήματα χωρίς GPUs είτε μηχανήματα με πολλαπλές GPUs. Ο τρόπος με τον οποίο συγκεντρώνουμε τα μηχανήματα αυτά είναι ένα JSON αρχείο που ονομάζουμε blueprint. Σε αυτό αναγράφονται τα στοιχεία των μηχανημάτων, δηλαδή η IP τους, το username με το οποίο έχουμε πρόσβαση σε αυτά, ο κωδικός πρόσβασης καθώς και αν είναι εφοδιασμένα ή όχι με GPUs και ο αριθμός αυτών. Επίσης αναγράφεται εάν είναι αποδοτικά ώς προς την ενέργεια (energy efficient). Παρακάτω φαίνεται τυπικά η μορφή του blueprint:

```
{
     "connections": [
          {
               "IP": "xxx.xxx.xxx.xxx",
               "Password": "pas1",
               "GPUS": n1
               "energy efficient": no
          },
          {
               "IP": "yyy.yyy.yyy.yyy",
               "Password": "pas2",
               "GPUS": n2
               "energy efficient": yes
          },
          {
               "IP": "zzz.zzz.zzz.zzz",
               "Password": "pas2",
               "GPUS": n3
               "energy efficient": no
          }
     ]
}
```

Ο τρόπος με τον οποίο έχουμε πρόσβαση στα μηχανήματα αυτά είναι το πρωτόκολλο SSH (Secure Shell), ένα ασφαλές δικτυακό πρωτόκολλο που επιτρέπει τη μεταφορά δεδομένων μεταξύ δύο υπολογιστών.

6.3 Δημιουργία του Cluster και επιλογή Στρατηγικής

Πριν ξεκινήσουμε την κατανομή της εργασίας στα μηχανήματα, θα πρέπει να επιλέξουμε ποια από αυτά είναι κατάλληλο για τη συγκεκριμένη εργασία. Δίνεται η δυνατότητα στο χρήστη να προσδιορίσει το είδος της εργασίας και με βάση αυτή να επιλεγούν τα κατάλληλα μηχανήματα. Υπάρχουν τέσσερις διαθέσιμες επιλογές οι οποίες προσδιορίζονται με τη μορφή arguments στο βασικό μας πρόγραμμα και περιγράφονται παρακάτω:

- **-all**
  Η συγκεκριμένη επιλογή χρησιμοποιεί όλα τα διαθέσιμα μηχανήματα και όλους τους διαθέσιμους πόρους. Προφανώς έχει τη μεγαλύτερη κατανάλωση σε ισχύ.

- **-gpu**
  Η επιλογή αυτή χρησιμοποιεί όλες τις διαθέσιμες GPUs και είναι κατάλληλη για εκπαίδευση νευρωνικών δικτύων καθώς προσφέρει δυνατότητα παραλληλισιμότητας.

- **-cpu**
  Εδώ χρησιμοποιούμε τα μηχανήματα που δεν είναι εφοδιασμένα με GPUs. Η επιλογή ενδείκνυται για μικρές υπολογιστικές διαδικασίες όπου προτιμάμε ταχύτητα από παραλληλισμό.

- **-energy**
  Η τελευταία μέθοδος χρησιμοποιεί τις συσκευές με τη μικρότερη κατανάλωση ενέργειας και έχει επομένως το μικρότερο κόστος λειτουργίας. Στην περίπτωση αυτή χάνουμε πιθανότατα σε απόδοση, αλλά κερδίζουμε σε οικονομία.

Αφού ο χρήστης προσδιορίσει τη μέθοδο που θα χρησιμοποιηθεί και κατά συνέπεια το σύνολο των μηχανημάτων, θα πρέπει στη συνέχεια να δημιουργηθεί η αναπαράσταση του cluster σε μορφή κατάλληλη για το TensorFlow. Στην πραγματικότητα, ένα cluster στο TensorFlow είναι ένα σύνολο από εργασίες - Tasks που συμμετέχουν στην κατανεμημένη εκτέλεση του γράφου. Κάθε Task συνδέεται με έναν Server, ο οποίος περιλαμβάνει έναν “master” που δημιουργεί τις συνεδρίες και έναν “worker” που εκτελεί τις πράξεις στο γράφο. Ένα cluster μπορεί να διαιρεθεί επίσης σε μία ή περισσότερες δουλειές - “jobs”, όπου το κάθε job περιέχει ένα ή περισσότερα tasks.

Η διαδικασία που πρέπει να ακολουθήσει ο χρήσης για να δημιουργήσει ένα cluster είναι να ξεκινήσει έναν Tensorflow Server για κάθε task σε αυτό. Τυπικά κάθε task τρέχει σε διαφορετικό μηχάνημα, αλλά μπορούμε να έχουμε πολλαπλά tasks σε ένα μηχάνημα, όταν αυτό περιέχει πολλαπλές συσκευές (GPUs). Θα πρέπει λοιπόν για κάθε task να γίνουν οι ακόλουθες ενέργειες σε κώδικα:

- Δημιουργία του **tf.train.ClusterSpec** το οποίο περιγράφει όλα τα tasks στο cluster. Αυτό θα είναι το ίδιο για όλα τα tasks.
- Δημιουργία του **tf.train.Server** στο οποίο περνάμε το ClusterSpec στον constructor και προσδιορίζουμε το παρών task με ένα job name και ένα task index.

Παρακάτω φαίνεται η μορφή του tf.train.Clusterspec και τι προσδιορίζει:

```
tf.train.ClusterSpec({
      "local": ["localhost:2222", "localhost:2223"]
      })
```

Εδώ προσδιορίζονται δύο tasks που ανήκουν στο ίδιο job που ονομάζουμε local και "ζουν" στο τοπικό μηχάνημα localhost στα ports 2222 και 2223.

```
/job:local/task:0
/job:local/task:1
```

```
tf.train.ClusterSpec({
    "worker": [
        "worker0.example.com:2222",
        "worker1.example.com:2222",
        "worker2.example.com:2222"
    ],
    "ps": [
        "ps0.example.com:2222",
        "ps1.example.com:2222"
    ]})
```

Εδώ φαίνεται μία αναπαράσταση που θα χρησιμοποιήσουμε, η οποία περιλαμβάνει δύο jobs το worker job και το ps – parameter server job με τρία και δύο tasks αντίστοιχα.

```
/job:worker/task:0
/job:worker/task:1
/job:worker/task:2
/job:ps/task:0
/job:ps/task:1
```

Όσον αφορά το tf.train.Server θα πρέπει να προσδιοριστεί σε κάθε task όπως φαίνεται παρακάτω για το πρώτο παράδειγμα μας:

```
 # In task 0:
cluster = tf.train.ClusterSpec({"local": ["localhost:2222", "localhost:2223"]})
server = tf.train.Server(cluster, job_name="local", task_index=0)
```

Δημιουργούμε δηλαδή σε κάθε task το ClusterSpec και το περνάμε στον Server έτσι ώστε να μπορεί να επικοινωνεί με τα υπόλοιπα tasks.

Βλέπουμε λοιπον, ότι όσο πιο μεγάλο είναι το cluster μας, απαιτείται μεγαλύτερος κόπος για την αρχικοποίηση του αφού θα πρέπει να ακολουθούμε την παραπάνω διαδικασία για κάθε μηχάνημα. Απαιτείται επομένως μία αυτοματοποιημένη διαδικασία που θα εκτελεί τις παραπάνω ενέργειες.

6.4 Προτεινόμενη Λύση

Η λύση που προτείνουμε περιλαμβάνει ένα διαθέσιμο API το οποίο λαμβάνει σαν παραμέτρους ορισμένα δεδομένα από τους χρήστες και αυτοματοποιεί τη διαδικασία δημιουργίας του cluster καθώς και την κατανομή του μοντέλου.

Οι χρήστες αρκεί να δημιουργήσουν το blueprint με τα διαθέσιμα μηχανήματα και συσκευές, και να δώσουν ως παράμετρο στο API το πρόγραμμα που θέλουν να κατανέμουν και μία από τις τέσσερις παραμέτρους που αναπτύχθηκαν νωρίτερα. Στη συνέχεια το API θα δημιουργήσει το κατάλληλο Cluster, θα συνδεθεί μέσω πρωτοκόλλου SSH στα διαθέσιμα μηχανήματα και θα κατανέμει το μοντέλο μας σε αυτά.

Η όλη διαδικασία πραγματοποιείται αυτόματα, εξοικονομώντας χρόνο από τους προγραμματιστές και προσφέρει αμεσότητα και ευκολία χρήσης. Στην ουσία, ο μόνος κόπος που απαιτείται είναι η δημιουργία του blueprint, και από έκει και πέρα με μία απλή εκτέλεση μίας εντολής αυτοματοποιείται δουλειά που σε άλλη περίπτωση θα απαιτούσε κόπο, χρόνο και σκέψη.

ΚΕΦΑΛΑΙΟ 7:
7.ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΚΑΙ ΑΠΟΤΕΛΕΣΜΑΤΑ

7.1 Επιλογή των Μηχανημάτων και Προετοιμασία

Για την υλοποίηση της προτεινόμενης μεθόδου επιλέξαμε ορισμένα μηχανήματα του εργαστηρίου με ετερογένεια ως προς τους υπολογιστικούς τους πόρους. Συγκεκριμένα επιλέξαμε μηχανήματα με στατικές διευθύνσεις IP στο υποδίκτυο 147.102.19.0/24. Αναλυτικά τα μηχανήματα και τα χαρακτηριστικά τους ακολουθούν παρακάτω:

- 147.102.19.131
  CPU: Intel i5 – 2400 3.1GHz
  GPU: -
  RAM: 8GB
  Storage: 500GB

- 147.102.19.132
  CPU: Intel i5 – 2400 3.1GHz
  GPU: -
  RAM: 8GB
  Storage: 500GB

- 147.102.19.133
  CPU: AMD FX 8120 8 cores 3.1GHz
  GPU: -
  RAM: 32GB
  Storage: 2TB

- 147.102.19.4
  CPU: AMD Phenom II X6 1090T 6 cores 3.2GHz
  GPU: Nvidia GeForce RTX 2060 6GB
  RAM: 12GB
  Storage: 2TB

- 147.102.19.7
  CPU: intel i7 - 3820 4 cores 3.6GHz
  GPU: Nvidia GeForce GTX 650 1GB
  RAM: 32GB
  Storage: 200GB

Όπως μπορούμε να δούμε, τα τρία πρώτα μηχανήματα δεν διαθέτουν GPUs και συγκεκριμένα τα δύο πρώτα είναι εφοδιασμένα με την CPU Intel i5 – 2400 3.1GHz, που λόγω της μικρής κατανάλωσης ισχύος μπορούν να χαρακτηριστούν και ενεργειακά αποδοτικά (energy efficient). Τα δύο τελευταία μηχανήματα είναι εφοδιασμένα με GPUs

και συγκεκριμένα το 147.102.19.4 με την ισχυρή Nvidia GeForce RTX 2060 6GB ενώ το 147.102.19.7 με την πιο παλιάς γενιάς Nvidia GeForce GTX 650 1GB.

Γνωρίζοντας τα παραπάνω μπορούμε να δημιουργήσουμε το blueprint που θα χαρακτηρίζει την παραπάνω διάταξη μηχανημάτων. Συγκεκριμένα:

```
{
     "connections": [
          {
               "IP": "147.102.19.131",
               "Password": "pas131",
               "GPUS": 0
               "energy efficient": yes
          },
          {
               "IP": "147.102.19.132",
               "Password": "pas132",
               "GPUS": 0
               "energy efficient": yes
          },
          {
               "IP": "147.102.19.133",
               "Password": "pas133",
               "GPUS": 0
               "energy efficient": no
          }
          {
               "IP": "147.102.19.4",
               "Password": "pas4",
               "GPUS": 1
               "energy efficient": no
          }
          {
               "IP": "147.102.19.7",
               "Password": "pas7",
               "GPUS": 1
               "energy efficient": no
          }
     ]
}
```

Στη συνέχεια θα πρέπει να επιτρέψουμε την επικοινωνία μεταξύ των μηχανημάτων, έτσι ώστε σε περίπτωση εκπαίδευσης νευρωνικού δικτύου να μπορούν να συγχρονίζουν τις παραμέτρους τους. Για το σκοπό αυτό επιτρέπουμε από το firewall του κάθε μηχανήματος να δέχεται πακέτα από τα υπόλοιπα τέσσερα. Το firewall που χρησιμοποιούμε είναι το UFW (Uncomplicated FireWall) και η εντολή που εκτελέσαμε ήταν:

```
sudo ufw allow from xxx.xxx.xxx.xxx
```

Κατόπιν θα πρέπει να εγκαταστήσουμε στα μηχανήματα τα απαραίτητα εργαλεία για να τρέξει το Tensorflow. Στα πρώτα τρία μηχανήματα που διαθέτουν μόνο CPUs αρκεί να

εγκαταστήσουμε απλά το TensorFlow και η έκδοση που επιλέγουμε είναι η stable 1.15.0-rc2. Στα υπόλοιπα δύο μηχανήματα το TensorFlow θα χρειαστεί υποστήριξη GPU οπότε είναι απαραίτητο να εγκατασταθούν ακόμη οι NVIDIA GPU Drivers (επιλέγουμε τους πιο πρόσφατους 435.21), το CUDA Toolkit (επιλέγουμε το 10.1) και τέλος το cuDNN SDK. Με την εγκατάσταση των παραπάνω εργαλείων τα μηχανήματα είναι έτοιμα για χρήση.

Μία τελευταία εκκρεμότητα είναι η εύκολη πρόσβαση στα πέντε μηχανήματα μέσω πρωτοκόλλου SSH από τρίτο υπολογιστή από τον οποίο θα ελέγχεται η διαδικασία. Χρειαζόμαστε λοιπόν πρόσβαση χωρίς συνθηματικό (password – less). Η λύση στο πρόβλημα μας είναι ο αλγόριθμος RSA που μας επιτρέπει να έχουμε πρόσβαση σε άλλα μηχανήματα με ένα ζεύγος κλειδιών: δημόσιο και ιδιωτικό (public and private). Δημιουργούμε επομένως ένα ζεύγος κλειδιών στον τρίτο υπολογιστή και "ανεβάζουμε" το δημόσιο κλειδί στα πέντε απομακρυσμένα μηχανήματα. Μπορούμε πλέον με το ιδιωτικό μας κλειδί να έχουμε πρόσβαση σε αυτά εύκολα χωρίς την ανάγκη συνθηματικών (passwords) και είμαστε έτοιμοι να μοντελοποιήσουμε τη διαδικασία και να εκτελέσουμε τα πειράματά μας.

7.2 Μοντελοποίηση της διαδικασίας και πείραμα

Ο βασικός μας κώδικας που εκτελεί τον αλγόριθμο κατανομής πόρων ονομάζεται resource_optimiser.py και βρίσκεται στο παράρτημα 1, στο τέλος της παρούσας διπλωματικής. Η διαδικασία που ακολουθούμε εδώ είναι η ακόλουθη: αρχικά επιλέγουμε το μοντέλο νευρωνικού δικτύου που θέλουμε ο αλγόριθμος μας να κατανέμει στους κατάλληλους πόρους των μηχανημάτων. Για το σκοπό αυτό επιλέγουμε ένα πρόβλημα ταξινόμησης εικόνων (image classification) που βασίζεται στην επιβλεπόμενη μάθηση με τη βοήθεια συνελικτικών νευρωνικών δικτύων (CNNs – Convolutioal Neural Networks). Πρόκειται για το γνωστό πρόβλημα ταξινόμησης χειρόγραφων ψηφίων (handwritten digits) όπου ως σύνολο δεδομένων (dataset) χρησιμοποιούμε το MNIST (Modified National Institute of Standards and Technology database). Για την εισαγωγή και την εκπαίδευση του μοντέλου χρησιμοποιούμε το Tensorflow και το Keras. Ο κώδικας για την εκπαίδευση του νευρωνικού δικτύου για οποιονδήποτε αριθμό εργατών (workers) βρίσκεται στο παράρτημα 2, στο τέλος της παρούσας διπλωματικής (MNIST_keras.py).

Στη συνέχεια της διαδικασίας, δίνουμε ως παραμέτρους (arguments) στον κώδικα resource_optimiser.py το blueprint που δημιουργήσαμε στην προηγούμενη ενότητα (blueprint.json), τον κώδικα εκπαίδευσης του δικτύου (MNIST_keras.py) και έναν από τους χαρακτηρισμούς που αναπτύχθηκαν στην ενότητα 6.3 και κατόπιν ξεκινάει η διαδικασία της κατανεμημένης εκπαίδευσης σε συγκεκριμένους πόρους. Παρακάτω φαίνονται οι εντολές που χρησιμοποιήσαμε για την πειραματική μας διαδικασία:

```
python3 resource_optimiser.py blueprint.json MNIST_keras.py -gpus

python3 resource_optimiser.py blueprint.json MNIST_keras.py -cpus

python3 resource_optimiser.py blueprint.json MNIST_keras.py -all

python3 resource_optimiser.py blueprint.json MNIST_keras.py -energy
```

Η βασική διαφορά των τεσσάρων παραπάνω εντολών είναι το cluster που δημιουργείται. Στην πρώτη περίπτωση θα περιλαμβάνει τα μηχανήματα 147.102.19.4 και 147.102.19.7 καθώς είναι τα μόνα με διαθέσιμες GPUs. Στην δεύτερη περίπτωση θα

περιλαμβάνει τα μηχανήματα 147.102.19.131, 147.102.19.132 και 147.102.19.133 καθώς είναι τα μόνα με αποκλειστικές συσκευές τις CPUs. Στην τρίτη περίπτωση θα περιλαμβάνει όλα τα μηχανήματα και στην τέταρτη και τελευταία θα περιλαμβάνει τα μηχανήματα με καλύτερη ενεργειακή απόδοση (energy efficient) δηλαδή τα 147.102.19.131 και 147.102.19.132.

Από τον ορισμό του cluster και μετά, αυτό που μένει είναι η κατανομή των διαθέσιμων πόρων του συγκεκριμένου cluster για την εκπαίδευση του μοντέλου μας. Την δουλειά αυτή την αναλαμβάνει το API tf.distribute.Strategy του Tensorflow και συγκεκριμένα η στρατηγική tf.distribute.experimental.MultiWorkerMirroredStrategy που περιγράφηκε στην ενότητα 5.4.1 και είναι κατάλληλη για σύγχρονη εκπαίδευση σε μηχανήματα με πολλαπλούς εργάτες (workers). Όπως ειπώθηκε και στην ενότητα 5 η στρατηγική αυτή δημιουργεί αντίγραφα όλων των μεταβλητών των επιπέδων του μοντέλου και τα τοποθετεί σε κάθε συσκευή κατά μήκος όλων των εργατών. Κατόπιν χρησιμοποιεί CollectiveOps, συγκεκριμένες εντολές του TensorFlow για συλλογική επικοινωνία έτσι ώστε να κρατάει τις μεταβλητές σε συγχρονισμό.

Στη συνέχεια ακολουθούν αριθμητικά στοιχεία από την εκτέλεση των πειραμάτων.

7.3 Αποτελέσματα Πειραμάτων

```
python3 resource_optimiser.py blueprint.json MNIST_keras.py -cpus
```

147.102.19.131:

```
Epoch 1/3
235/235 [==============================] - 19s 40ms/step - loss: 2.2178 - acc: 0.2847
Epoch 2/3
235/235 [==============================] - 15s 32ms/step - loss: 2.0012 - acc: 0.5315
Epoch 3/3
235/235 [==============================] - 15s 32ms/step - loss: 1.4633 - acc: 0.6981
```

147.102.19.132:

```
Epoch 1/3
235/235 [==============================] - 19s 40ms/step - loss: 2.2178 - acc: 0.2847
Epoch 2/3
235/235 [==============================] - 15s 32ms/step - loss: 2.0012 - acc: 0.5315
Epoch 3/3
235/235 [==============================] - 15s 32ms/step - loss: 1.4633 - acc: 0.6981
```

147.102.19.133:

```
Epoch 1/3
235/235 [==============================] - 19s 40ms/step - loss: 2.2178 - acc: 0.2847
Epoch 2/3
235/235 [==============================] - 15s 32ms/step - loss: 2.0012 - acc: 0.5315
Epoch 3/3
235/235 [==============================] - 15s 32ms/step - loss: 1.4633 - acc: 0.6981
```

Στην πρώτη εκτέλεση βλέπουμε την σύγχρονη εκπαίδευση στα μηχανήματα που περιέχουν αποκλειστικά CPUs. Το batch size είναι προφανώς μειωμένο και μπορούμε επίσης να δούμε τα αποτελέσματα για τη συνάρτηση loss και την ακρίβεια (accuracy). Η εκπαίδευση ολοκληρώνεται σε τρείς εποχές.

```
python3 resource_optimiser.py blueprint.json MNIST_keras.py -gpus
```

147.102.19.4:

```
Epoch 1/3
469/469 [==============================] - 9s 18ms/step - loss: 2.2174 - accuracy: 0.2142
Epoch 2/3
469/469 [==============================] - 1s 3ms/step - loss: 1.9502 - accuracy: 0.5165
Epoch 3/3
469/469 [==============================] - 1s 3ms/step - loss: 1.4742 - accuracy: 0.7417
```

147.102.19.7:

```
Epoch 1/3
469/469 [==============================] - 9s 18ms/step - loss: 2.2174 - accuracy: 0.2142
Epoch 2/3
469/469 [==============================] - 1s 3ms/step - loss: 1.9502 - accuracy: 0.5165
Epoch 3/3
469/469 [==============================] - 1s 3ms/step - loss: 1.4742 - accuracy: 0.7417
```

Στην δεύτερη εκτέλεση, βλέπουμε τη σύγχρονη εκπαίδευση στα δύο μηχανήματα που περιέχουν GPUs. Το batch size είναι μεγαλύτερο από πριν αφού έχουμε δύο μηχανήματα και μπορούμε επιπλέον να δούμε μία σημαντική βελτίωση όσον αφορά το χρόνο εκπαίδευσης (που είναι προφανές αφού χρησιμοποιούμε GPUs) και την ακρίβεια (accuracy).

```
python3 resource_optimiser.py blueprint.json MNIST_keras.py -energy
```

147.102.19.131:

```
Epoch 1/3
    469/Unknown - 20s 43ms/step - loss: 2.2626 - acc: 0.24862019-11-09 02:04:52.928844: W t
ensorflow/core/common_runtime/base_collective_executor.cc:216] BaseCollectiveExecutor::Star
tAbort Out of range: End of sequence
         [[{{node IteratorGetNext}}]]
469/469 [==============================] - 20s 43ms/step - loss: 2.2626 - acc: 0.2486
Epoch 2/3
469/469 [==============================] - 16s 35ms/step - loss: 2.1108 - acc: 0.5181
Epoch 3/3
469/469 [==============================] - 16s 35ms/step - loss: 1.7929 - acc: 0.6607
```

147.102.19.132:

```
Epoch 1/3
    469/Unknown - 20s 43ms/step - loss: 2.2626 - acc: 0.24862019-11-09 02:04:52.928844: W t
ensorflow/core/common_runtime/base_collective_executor.cc:216] BaseCollectiveExecutor::Star
tAbort Out of range: End of sequence
         [[{{node IteratorGetNext}}]]
469/469 [==============================] - 20s 43ms/step - loss: 2.2626 - acc: 0.2486
Epoch 2/3
469/469 [==============================] - 16s 35ms/step - loss: 2.1108 - acc: 0.5181
Epoch 3/3
469/469 [==============================] - 16s 35ms/step - loss: 1.7929 - acc: 0.6607
```

Στην τρίτη περίπτωση βλέπουμε τη σύγχρονη εκπαίδευση στα δύο μηχανήματα με πανομοιότυπες και ενεργειακά αποδοτικές (energy efficient) CPUs. Προφανώς ο χρόνος εκπαίδευσης είναι μεγαλύτερος από τις προηγούμενες περιπτώσεις καθώς αυτό που μας ενδιαφέρει εδώ δεν είναι η απόδοση αλλά η οικονομία.

Για να έχουμε μία συνολική εικόνα για τις προηγούμενες περιπτώσεις εκτελούμε την εκπαίδευση σε ένα μόνο μηχάνημα χωρίς κατανομή σε άλλες συσκευές:

python3 MNIST_keras.py

147.102.19.131:

```
Epoch 1/3
938/938 [==============================] - 17s 18ms/step - loss: 1.9424 - acc: 0.4883
Epoch 2/3
938/938 [==============================] - 17s 18ms/step - loss: 0.9933 - acc: 0.7881
Epoch 3/3
938/938 [==============================] - 16s 17ms/step - loss: 0.5846 - acc: 0.8553
```

Στη συγκεκριμένη περίπτωση βλέπουμε ότι το μοντέλο ολοκληρώνει την εκπαίδευση σχετικά γρήγορα με αρκετά καλούς αριθμούς απόδοσης (accuracy). Αυτό συμβαίνει επειδή το μοντέλο μας είναι μικρό σε μέγεθος και επομένως η εκπαίδευση σε μία μόνο CPU μπορεί να φέρει ικανοποιητικά αποτελέσματα.

7.4 Αξιολόγηση της Διαδικασίας

Ο αλγόριθμος που προτείνουμε παρουσιάζει έναν τρόπο για να αξιοποιηθούν συγκεκριμένοι πόροι από μία δεξαμενή μηχανημάτων στα οποία έχουμε πρόσβαση. Υπάρχουν τέσσερις τρόποι αξιοποίησης πόρων οι οποίοι δίνουν διαφορετικά αποτελέσματα ανάλογα με τα διαθέσιμα μηχανήματα και τους πόρους. Οι χρήστες γνωρίζοντας από πριν τα μοντέλα τους αρκεί να κάνουν ορισμένες αλλαγές στον κώδικά τους, ουσιαστικά για να προσθέσουν έναν τύπο στρατηγικής που υποστηρίζει το TensorFlow (στη συγκεκριμένη περίπτωση τη στρατηγική MultiWorker Mirrored Strategy). Από εκεί και πέρα ο κώδικας μας αναλαμβάνει την κατανομή του μοντέλου στους κατάλληλους πόρους. Προφανώς ο αλγόριθμος μας είναι κατάλληλος σε περιπτώσεις που υπάρχει μεγάλος αριθμός μηχανημάτων και πόρων και κατά συνέπεια η κατανομή του μοντέλου σε αυτούς από τον επιστήμονα δεδομένων είναι κουραστική. Ως παράδειγμα δίνουμε το μοντέλο ResNet το οποίο απαιτεί έως και μία εβδομάδα για την εκπαίδευση του ακόμα και όταν ανατίθεται σε μία πανίσχυρη GPU. Σε περιπτώσεις όπως αυτή όπου απαιτείται κατανομή του μοντέλου σε πόρους για την συντόμευση της εκπαίδευσης ο αλγόριθμος μας μπορεί να δουλέψει καλά.

Ένας παράγοντας που θεωρείται ως μειονέκτημα στον τρόπο μας είναι το γεγονός ότι ο προγραμματιστής θα πρέπει να έχει αξιολογήσει από πριν το μοντέλο του και να έχει επιλέξει ποια από τις τέσσερις επιλογές είναι κατάλληλη για αυτόν. Μία σημαντική πρόταση που ξεφεύγει από τα όρια της παρούσας διπλωματικής, είναι η δημιουργία ενός API που θα αυτοματοποιεί επιπλέον τον δικό μας αλγόριθμο. Συγκεκριμένα, θα λάμβανε υπόψιν του στοιχεία όπως των αριθμό και τον τύπο των παραμέτρων, τα επίπεδα (layers) του μοντέλου, τις υπολογιστικές δυνατότητες των πόρων, το μέγεθος του συνόλου δεδομένων εισόδου (input dataset) ακόμη και τα επίπεδα κίνησης (network traffic) μεταξύ των μηχανημάτων. Με βάση αυτά και κάποιο μοντέλο απόφασης θα μπορούσε ίσως να

κάνει την καταλληλότερη επιλογή όσων αφορά την επιλογή των πόρων αλλά και την επιλογή της στρατηγικής του TensorFlow. Θα μπορούσε επίσης να τροποποιεί τον κώδικα με βάση τη συγκεκριμένη στρατηγική.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Κωνσταντίνος Διαμαντάρας. Τεχνητά Νευρωνικά Δίκτυα, Κλειδάριθμος 2007

Βλαχάβας Ιωάννης, Κεφάλας Πέτρος, Βασιλειάδης Νικόλαος, Κόκκοράς Φώτης, Σακκελαρίου Ηλίας. Τεχνητή Νοημοσύνη, Εκδόσεις Πανεπιστημίου Μακεδονίας, 2011

Σαραντής Μητρόπουλος, Χρήστος Δουληγέρης. Πληροφοριακά Συστήματα στο Διαδίκτυο, Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών 2015

R. ELMASRI – S.B. NAVATHE. Θεμελιώδεις Αρχές Συστημάτων Βάσεων Δεδομένων, Δίαυλος 2016

Haoran Sun, Xiangyi Chen, Qingjiang Shi, Mingyi Hong, Xiao Fu, Nikos D. Sidiropoulos. LEARNING TO OPTIMIZE: TRAINING DEEP NEURAL NETWORKS FOR WIRELESS RESOURCE MANAGEMENT, 2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)

Erci Xu and Shanshan Li. Revisiting Resource Management for Deep Learning Framework, *Electronics* 2019, *8*(3), 327

Vrettos Moulos , George Chatzikyriakos , Vassilis Kassouras, Anastasios Doulamis, Nikolaos Doulamis, Georgios Leventakis, Thodoris Florakis 1, Theodora Varvarigou 1, Evangelos Mitsokapas, Georgios Kioumourtzis, Petros Klirodetis , Alexandros Psychas, Achilleas Marinakis, Thanasis Sfetsos, Alexios Koniaris and Dimitris Liapis and Anna Gatzioura. A Robust Information Life Cycle Management Framework for Securing and Governing Critical Infrastructure Systems, *Inventions* 2018, *3*(4), 71

Mingxi Cheng, Ji Li, and Shahin Nazarian. DRL-Cloud: Deep Reinforcement Learning-Based Resource Provisioning and Task Scheduling for Cloud Service Providers, 978-1-5090-0602-1/18/$31.00 ©2018 IEEE

ΠΑΡΑΡΤΗΜΑ 1

Ακολουθεί ο κώδικας του resource_optimizer.py για την αποδοτική κατανομή των διαθέσιμων πόρων:

```
import sys, os, string, json, argparse

from subprocess import call, Popen, PIPE

parser = argparse.ArgumentParser()
parser.add_argument("-all",      help="use      all      resources",
action="store_true")
parser.add_argument("-cpus",       help="use       only       cpus",
action="store_true")
parser.add_argument("-gpus",       help="use       only       gpus",
action="store_true")
parser.add_argument("-energy",  help="use  only  energy  efficient
machines", action="store_true")

parser.add_argument("input_file", help="input file")
parser.add_argument("run_file", nargs="?", help="run file")
args = parser.parse_args()

f1 = open(args.input_file)

data = json.load(f1)

all_ = data['connections']

cpus = []
gpus = []
energy = []

for p in data['connections']:
     if(p['energy_efficient'] == 'yes'):
          energy.append(p)
     if(p['GPUS'] != 0):
          gpus.append(p)
     else:
          cpus.append(p)

l = []

if args.cpus:
     l = cpus
elif args.gpus:
     l = gpus
elif args.energy:
     l = energy
```

```
else:
     l = all_

cluster0 = []
for p in l:
     cluster0.append(p["IP"])

cluster = [x + ":2223" for x in cluster0]

i = 0
for p in l:
     TF_CONFIG='{\"cluster\": {\"worker\": ' + str(cluster) + '},
\"task\": {\"index\": ' + str(i) + ', \"type\": \"worker\"}}'
     i = i + 1
     print(TF_CONFIG)
     Popen(['gnome-terminal', '--tab', '-e', "ssh -t ntua@" +
p['IP'] + " \'cd my_tensorflow/; source ./venv/bin/activate; export
TF_CONFIG; python3 " + str(args.run_file) + "; bash -l\'"])
```

ΠΑΡΑΡΤΗΜΑ 2

Ακολουθεί ο κώδικας εκπαίδευσης του δημοφιλούς μοντέλου συνελικτικού Νευρωνικού δικτύου MNIST για την ταξινόμηση χειρόγραφων ψηφίων σε ένα μόνο μηχάνημα:

```
from __future__ import absolute_import, division, print_function,
unicode_literals

import tensorflow_datasets as tfds
import tensorflow as tf
tfds.disable_progress_bar()

BUFFER_SIZE = 10000
BATCH_SIZE = 64

def make_datasets_unbatched():
  # Scaling MNIST data from (0, 255] to (0., 1.]
  def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    return image, label

  datasets, info = tfds.load(name='mnist',
                             with_info=True,
                             as_supervised=True)

  return datasets['train'].map(scale).cache().shuffle(BUFFER_SIZE)

train_datasets = make_datasets_unbatched().batch(BATCH_SIZE)

def build_and_compile_cnn_model():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32,        3,        activation='relu',
input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')
  ])
  model.compile(
      loss=tf.keras.losses.sparse_categorical_crossentropy,
      optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
      metrics=['accuracy'])
  return model

single_worker_model = build_and_compile_cnn_model()
single_worker_model.fit(x=train_datasets, epochs=3)
```

ΠΑΡΑΡΤΗΜΑ 3

Ακολουθεί ο κώδικας εκπαίδευσης του δημοφιλούς μοντέλου συνελικτικού Νευρωνικού δικτύου MNIST για την ταξινόμηση χειρόγραφων ψηφίων πολλαπλά μηχανήματα (N workers):

```
from __future__ import absolute_import, division, print_function,
unicode_literals

import tensorflow_datasets as tfds
import tensorflow as tf
tfds.disable_progress_bar()

BUFFER_SIZE = 10000
BATCH_SIZE = 64

def make_datasets_unbatched():
  # Scaling MNIST data from (0, 255] to (0., 1.]
  def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    return image, label

  datasets, info = tfds.load(name='mnist',
                             with_info=True,
                             as_supervised=True)

  return datasets['train'].map(scale).cache().shuffle(BUFFER_SIZE)

train_datasets = make_datasets_unbatched().batch(BATCH_SIZE)

def build_and_compile_cnn_model():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32,       3,       activation='relu',
input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')
  ])
  model.compile(
      loss=tf.keras.losses.sparse_categorical_crossentropy,
      optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
      metrics=['accuracy'])
  return model

strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()

NUM_WORKERS = N
```

```
GLOBAL_BATCH_SIZE = 64 * NUM_WORKERS
with strategy.scope():
  # Creation of dataset, and model building/compiling need to be
within
  # `strategy.scope()`.
  options = tf.data.Options()
  options.experimental_distribute.auto_shard = False
  train_datasets_no_auto_shard                                    =
train_datasets.with_options(options))
  multi_worker_model = build_and_compile_cnn_model()
multi_worker_model.fit(x=train_datasets_no_auto_shard, epochs=3)
```

ΠΑΡΑΡΤΗΜΑ 4

Ακολουθεί ο source code για την στρατηγική που ακολουθήσαμε:
tf.distribute.experimental.MultiWorkerMirroredStrategy

```
"""Class CollectiveAllReduceStrategy implementing DistributionStrategy."""


from __future__ import absolute_import
from __future__ import division
from __future__ import print_function


import copy


from tensorflow.core.protobuf import config_pb2
from tensorflow.core.protobuf import rewriter_config_pb2
from tensorflow.core.protobuf import tensorflow_server_pb2
from tensorflow.python.distribute import cross_device_ops as
cross_device_ops_lib
from tensorflow.python.distribute import cross_device_utils
from tensorflow.python.distribute import device_util
from tensorflow.python.distribute import distribute_lib
from tensorflow.python.distribute import input_lib
from tensorflow.python.distribute import mirrored_strategy
from tensorflow.python.distribute import multi_worker_util
from tensorflow.python.distribute import numpy_dataset
from tensorflow.python.distribute import reduce_util
from tensorflow.python.distribute import values
from tensorflow.python.distribute.cluster_resolver import
SimpleClusterResolver
from tensorflow.python.distribute.cluster_resolver import
TFConfigClusterResolver
from tensorflow.python.eager import context
from tensorflow.python.framework import ops
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import collective_ops
from tensorflow.python.platform import tf_logging as logging
from tensorflow.python.util.tf_export import tf_export



# TODO(yuefengz): support in-graph replication.
```

```
@tf_export("distribute.experimental.MultiWorkerMirroredStrategy", v1=[])
class CollectiveAllReduceStrategy(distribute_lib.Strategy):
  """A distribution strategy for synchronous training on multiple workers.
  This strategy implements synchronous distributed training across multiple
  workers, each with potentially multiple GPUs. Similar to
  `tf.distribute.MirroredStrategy`, it creates copies of all variables in the
  model on each device across all workers.
  It uses CollectiveOps's implementation of multi-worker all-reduce to
  to keep variables in sync. A collective op is a single op in the
  TensorFlow graph which can automatically choose an all-reduce algorithm in
  the TensorFlow runtime according to hardware, network topology and tensor
  sizes.
  By default it uses all local GPUs or CPU for single-worker training.
  When 'TF_CONFIG' environment variable is set, it parses cluster_spec,
  task_type and task_id from 'TF_CONFIG' and turns into a multi-worker
strategy
  which mirrores models on GPUs of all machines in a cluster. In the current
  implementation, it uses all GPUs in a cluster and it assumes all workers
have
  the same number of GPUs.
  It supports both eager mode and graph mode. However, for eager mode, it has
to
  set up the eager context in its constructor and therefore all ops in eager
  mode have to run after the strategy object is created.
  """


  def __init__(
      self,
      communication=cross_device_ops_lib.CollectiveCommunication.AUTO):
    """Creates the strategy.
    Args:
      communication: optional Enum of type
        `distribute.experimental.CollectiveCommunication`.  This provides a
way
        for the user to override the choice of collective op communication.
        Possible values include `AUTO`, `RING`, and `NCCL`.
    """
    super(CollectiveAllReduceStrategy, self).__init__(
        CollectiveAllReduceExtended(
            self,
            communication=communication))


  @classmethod
  def _from_local_devices(cls, devices):
```

```
    """A convenience method to create an obejct with a list of devices."""
    obj = cls()
    obj.extended._initialize_local(TFConfigClusterResolver(), devices=devices)
# pylint: disable=protected-access
    return obj


  def scope(self):  # pylint: disable=useless-super-delegation
    """Returns a context manager selecting this Strategy as current.
    Inside a `with strategy.scope():` code block, this thread
    will use a variable creator set by `strategy`, and will
    enter its "cross-replica context".
    In `MultiWorkerMirroredStrategy`, all variables created inside
    `strategy.scope() will be mirrored on all replicas of each worker.
    Moreover, it also sets a default device scope so that ops without
    specified devices will end up on the correct worker.
    Returns:
      A context manager to use for creating variables with this strategy.
    """
    return super(CollectiveAllReduceStrategy, self).scope()
```

```
@tf_export(v1=["distribute.experimental.MultiWorkerMirroredStrategy"])
class CollectiveAllReduceStrategyV1(distribute_lib.StrategyV1):


  __doc__ = CollectiveAllReduceStrategy.__doc__


  def __init__(
      self,
      communication=cross_device_ops_lib.CollectiveCommunication.AUTO):
    """Initializes the object."""
    super(CollectiveAllReduceStrategyV1, self).__init__(
        CollectiveAllReduceExtended(
            self,
            communication=communication))
```

```
class CollectiveAllReduceExtended(mirrored_strategy.MirroredExtended):
  """Implementation of CollectiveAllReduceStrategy."""
```

```
  def __init__(self,
               container_strategy,
               communication,
               cluster_resolver=TFConfigClusterResolver()):
    distribute_lib.StrategyExtendedV1.__init__(self, container_strategy)
    assert isinstance(
        communication,
        cross_device_ops_lib.CollectiveCommunication)
    self._communication = communication
    self._initialize_strategy(cluster_resolver)
    assert isinstance(self._get_cross_device_ops(),
                      cross_device_ops_lib.CollectiveAllReduce)


  def _initialize_strategy(self, cluster_resolver):
    if cluster_resolver.cluster_spec().as_dict():
      self._initialize_multi_worker(cluster_resolver)
    else:
      self._initialize_local(cluster_resolver)


  def _initialize_local(self, cluster_resolver, devices=None):
    """Initializes the object for local training."""
    self._is_chief = True
    self._num_workers = 1


    if ops.executing_eagerly_outside_functions():
      try:
        context.context().configure_collective_ops(
            scoped_allocator_enabled_ops=("CollectiveReduce",),
            use_nccl_communication=(self._communication ==
cross_device_ops_lib
                                    .CollectiveCommunication.NCCL))
      except RuntimeError:
        logging.warning("Collective ops is not configured at program startup.
"
                        "Some performance features may not be enabled.")
      self._collective_ops_configured = True


    # TODO(b/126786766): TFConfigClusterResolver returns wrong number of GPUs
in
    # some cases.
    if isinstance(cluster_resolver, TFConfigClusterResolver):
```

```
      num_gpus = context.num_gpus()
    else:
      num_gpus = cluster_resolver.num_accelerators().get("GPU", 0)


    if devices:
      local_devices = devices
    else:
      if num_gpus:
        local_devices = tuple("/device:GPU:%d" % i for i in range(num_gpus))
      else:
        local_devices = ("/device:CPU:0",)
    self._worker_device = device_util.canonicalize("/device:CPU:0")
    self._host_input_device = numpy_dataset.SingleDevice(self._worker_device)


    self._collective_keys = cross_device_utils.CollectiveKeys()
    # TODO(yuefengz): remove num_gpus_per_worker from CollectiveAllReduce.
    self._cross_device_ops = cross_device_ops_lib.CollectiveAllReduce(
        num_workers=self._num_workers,
        num_gpus_per_worker=num_gpus,
        collective_keys=self._collective_keys)
    super(CollectiveAllReduceExtended, self)._initialize_local(local_devices)


    self._cluster_spec = None
    self._task_type = None
    self._task_id = None


    # This is a mark to tell whether we are running with standalone client or
    # independent worker. Right now with standalone client, strategy object is
    # created as local strategy and then turn into multi-worker strategy via
    # configure call.
    self._local_or_standalone_client_mode = True


    # Save the num_gpus_per_worker and rpc_layer for configure method.
    self._num_gpus_per_worker = num_gpus
    self._rpc_layer = cluster_resolver.rpc_layer
    self._warn_nccl_no_gpu()


    logging.info("Single-worker CollectiveAllReduceStrategy with local_devices
"
                 "= %r, communication = %s", local_devices,
```

```
self._communication)


  def _initialize_multi_worker(self, cluster_resolver):
    """Initializes the object for multi-worker training."""
    cluster_spec = multi_worker_util.normalize_cluster_spec(
        cluster_resolver.cluster_spec())
    task_type = cluster_resolver.task_type
    task_id = cluster_resolver.task_id
    if task_type is None or task_id is None:
      raise ValueError("When `cluster_spec` is given, you must also specify "
                       "`task_type` and `task_id`.")
    self._cluster_spec = cluster_spec
    self._task_type = task_type
    self._task_id = task_id


    self._num_workers = multi_worker_util.worker_count(cluster_spec,
task_type)
    if not self._num_workers:
      raise ValueError("No `worker`, `chief` or `evaluator` tasks can be found
"
                       "in `cluster_spec`.")


    self._is_chief = multi_worker_util.is_chief(cluster_spec, task_type,
                                                task_id)


    self._worker_device = "/job:%s/task:%d" % (task_type, task_id)
    self._host_input_device = numpy_dataset.SingleDevice(self._worker_device)


    if (ops.executing_eagerly_outside_functions() and
        not getattr(self, "_local_or_standalone_client_mode", False)):
      context.context().configure_collective_ops(
          collective_leader=multi_worker_util.collective_leader(
              cluster_spec, task_type, task_id),
          scoped_allocator_enabled_ops=("CollectiveReduce",),
          use_nccl_communication=(self._communication == cross_device_ops_lib
                                  .CollectiveCommunication.NCCL),
          device_filters=("/job:%s/task:%d" % (task_type, task_id),))
      self._collective_ops_configured = True


    # Starting a std server in eager mode and in independent worker mode.
```

```
    if (context.executing_eagerly() and
        not getattr(self, "_std_server_started", False) and
        not getattr(self, "_local_or_standalone_client_mode", False)):
      # Checking _local_or_standalone_client_mode as well because we should
not
      # create the std server in standalone client mode.
      config_proto = config_pb2.ConfigProto()
      config_proto = self._update_config_proto(config_proto)
      server_def = tensorflow_server_pb2.ServerDef(
          cluster=cluster_spec.as_cluster_def(),
          default_session_config=config_proto,
          job_name=task_type,
          task_index=task_id,
          protocol=cluster_resolver.rpc_layer or "grpc")
      context.context().enable_collective_ops(server_def)
      self._std_server_started = True
      # The `ensure_initialized` is needed before calling
      # `context.context().devices()`.
      context.context().ensure_initialized()
      logging.info(
          "Enabled multi-worker collective ops with available devices: %r",
          context.context().devices())


    # TODO(yuefengz): The `num_gpus` is only for this particular task. It
    # assumes all workers have the same number of GPUs. We should remove this
    # assumption by querying all tasks for their numbers of GPUs.
    # TODO(b/126786766): TFConfigClusterResolver returns wrong number of GPUs
in
    # some cases.
    if isinstance(cluster_resolver, TFConfigClusterResolver):
      num_gpus = context.num_gpus()
    else:
      num_gpus = cluster_resolver.num_accelerators().get("GPU", 0)


    if num_gpus:
      local_devices = tuple("%s/device:GPU:%d" % (self._worker_device, i)
                            for i in range(num_gpus))
    else:
      local_devices = (self._worker_device,)


    self._collective_keys = cross_device_utils.CollectiveKeys()
    self._cross_device_ops = cross_device_ops_lib.CollectiveAllReduce(
        num_workers=self._num_workers,
```

```
        num_gpus_per_worker=num_gpus,
        collective_keys=self._collective_keys)
    super(CollectiveAllReduceExtended, self)._initialize_local(local_devices)
    self._input_workers = input_lib.InputWorkers(
        self._device_map, [(self._worker_device, self.worker_devices)])


    # Add a default device so that ops without specified devices will not end
up
    # on other workers.
    self._default_device = "/job:%s/task:%d" % (task_type, task_id)


    # Save the num_gpus_per_worker and rpc_layer for configure method.
    self._num_gpus_per_worker = num_gpus
    self._rpc_layer = cluster_resolver.rpc_layer
    self._warn_nccl_no_gpu()


    logging.info(
        "Multi-worker CollectiveAllReduceStrategy with cluster_spec = %r, "
        "task_type = %r, task_id = %r, num_workers = %r, local_devices = %r, "
        "communication = %s", cluster_spec.as_dict(), task_type,
        task_id, self._num_workers, local_devices,
        self._communication)


  def _get_variable_creator_initial_value(self,
                                          replica_id,
                                          device,
                                          primary_var,
                                          **kwargs):
    if replica_id == 0:  # First replica on each worker.
      assert device is not None
      assert primary_var is None


      def initial_value_fn():  # pylint: disable=g-missing-docstring
        # Only the first device participates in the broadcast of initial
values.
        group_key = self._collective_keys.get_group_key([device])
        group_size = self._num_workers
        collective_instance_key = (
            self._collective_keys.get_variable_instance_key())
```

```
      with ops.device(device):
        initial_value = kwargs["initial_value"]
        if callable(initial_value):
          initial_value = initial_value()
        assert not callable(initial_value)
        initial_value = ops.convert_to_tensor(
            initial_value, dtype=kwargs.get("dtype", None))


        if self._num_workers > 1:
          if self._is_chief:
            bcast_send = collective_ops.broadcast_send(
                initial_value, initial_value.shape, initial_value.dtype,
                group_size, group_key, collective_instance_key)
            with ops.control_dependencies([bcast_send]):
              return array_ops.identity(initial_value)
          else:
            return collective_ops.broadcast_recv(initial_value.shape,
                                                 initial_value.dtype,
                                                 group_size, group_key,
                                                 collective_instance_key)
        return initial_value


    return initial_value_fn
  else:
    return super(CollectiveAllReduceExtended,
                 self)._get_variable_creator_initial_value(
                     replica_id=replica_id,
                     device=device,
                     primary_var=primary_var,
                     **kwargs)


def _make_input_context(self):
  if self._cluster_spec is None:
    input_pipeline_id = 0
  else:
    input_pipeline_id = multi_worker_util.id_in_cluster(
        self._cluster_spec, self._task_type, self._task_id)
  input_context = distribute_lib.InputContext(
      num_input_pipelines=self._num_workers,
      input_pipeline_id=input_pipeline_id,
      num_replicas_in_sync=self._num_replicas_in_sync)
  return input_context
```

```
def _experimental_distribute_dataset(self, dataset):
  input_context = self._make_input_context()
  return input_lib.get_distributed_dataset(
      dataset,
      self._input_workers,
      self._container_strategy(),
      split_batch_by=self._num_replicas_in_sync,
      input_context=input_context)


def _make_dataset_iterator(self, dataset):
  """Distributes the dataset to each local GPU."""
  input_context = self._make_input_context()
  return input_lib.DatasetIterator(
      dataset,
      self._input_workers,
      self._container_strategy(),
      split_batch_by=self._num_replicas_in_sync,
      input_context=input_context)


def _make_input_fn_iterator(
    self,
    input_fn,
    replication_mode=distribute_lib.InputReplicationMode.PER_WORKER):
  """Distributes the input function to each local GPU."""
  input_context = self._make_input_context()
  return input_lib.InputFunctionIterator(input_fn, self._input_workers,
                                         [input_context],
                                         self._container_strategy())


def _configure(self,
               session_config=None,
               cluster_spec=None,
               task_type=None,
               task_id=None):
  """Configures the object.
  Args:
    session_config: a `tf.compat.v1.ConfigProto`
    cluster_spec: a dict, ClusterDef or ClusterSpec object specifying the
      cluster configurations.
    task_type: the current task type, such as "worker".
    task_id: the current task id.
  Raises:
```

```
      ValueError: if `task_type` is not in the `cluster_spec`.
    """
    if cluster_spec:
      # Use the num_gpus_per_worker recorded in constructor since _configure
      # doesn't take num_gpus.
      cluster_resolver = SimpleClusterResolver(
          cluster_spec=multi_worker_util.normalize_cluster_spec(cluster_spec),
          task_type=task_type,
          task_id=task_id,
          num_accelerators={"GPU": self._num_gpus_per_worker},
          rpc_layer=self._rpc_layer)
      self._initialize_multi_worker(cluster_resolver)
      assert isinstance(self._get_cross_device_ops(),
                        cross_device_ops_lib.CollectiveAllReduce)


    if session_config:
      session_config.CopyFrom(self._update_config_proto(session_config))


  def _update_config_proto(self, config_proto):
    updated_config = copy.deepcopy(config_proto)
    # Enable the scoped allocator optimization for CollectiveOps.  This
    # optimization converts many small all-reduces into fewer larger
    # all-reduces.
    rewrite_options = updated_config.graph_options.rewrite_options
    rewrite_options.scoped_allocator_optimization = (
        rewriter_config_pb2.RewriterConfig.ON)
    # We turn on ScopedAllocator only for CollectiveReduce op, i.e. enable_op
=
    # ["CollectiveReduce"].  Since we can't assign to a repeated proto field,
we
    # clear and then append.
    del rewrite_options.scoped_allocator_opts.enable_op[:]
    rewrite_options.scoped_allocator_opts.enable_op.append("CollectiveReduce")


    if self._communication ==
cross_device_ops_lib.CollectiveCommunication.NCCL:
      updated_config.experimental.collective_nccl = True


    if not self._cluster_spec:
      return updated_config
```

```
    assert self._task_type
    assert self._task_id is not None

    # Collective group leader is needed for collective ops to coordinate
    # workers.
    updated_config.experimental.collective_group_leader = (
        multi_worker_util.collective_leader(self._cluster_spec,
self._task_type,
                                            self._task_id))

    # The device filters prevent communication between workers.
    del updated_config.device_filters[:]
    updated_config.device_filters.append(
        "/job:%s/task:%d" % (self._task_type, self._task_id))

    return updated_config


  def _reduce_to(self, reduce_op, value, destinations):
    if (isinstance(value, values.Mirrored) and
        reduce_op == reduce_util.ReduceOp.MEAN):
      return value
    assert not isinstance(value, values.Mirrored)

    if (isinstance(value, values.DistributedValues) and
        len(self.worker_devices) == 1):
      value = value.values[0]

    # When there are multiple workers, we need to reduce across workers using
    # collective ops.
    if (not isinstance(value, values.DistributedValues) and
        self._num_workers == 1):
      # This function handles reducing values that are not PerReplica or
      # Mirrored values. For example, the same value could be present on all
      # replicas in which case `value` would be a single value or value could
      # be 0.
      return cross_device_ops_lib.reduce_non_distributed_value(
          reduce_op, self._device_map, value, destinations)
    return self._get_cross_device_ops().reduce(
        reduce_op, value, destinations=destinations)
```

```
  def _warn_nccl_no_gpu(self):
    if ((self._communication ==
         cross_device_ops_lib.CollectiveCommunication.NCCL) and
        self._num_gpus_per_worker == 0):
      logging.warning("Enabled NCCL communication but no GPUs detected/"
                      "specified.")


  @property
  def experimental_between_graph(self):
    return True


  @property
  def experimental_should_init(self):
    return True


  @property
  def should_checkpoint(self):
    return self._is_chief


  @property
  def should_save_summary(self):
    return self._is_chief


  @property
  def _num_replicas_in_sync(self):
    return len(self.worker_devices) * self._num_workers


  # TODO(priyag): Delete this once all strategies use global batch size.
  @property
  def _global_batch_size(self):
    """`make_dataset_iterator` and `make_numpy_iterator` use global batch
size.
    `make_input_fn_iterator` assumes per-replica batching.
    Returns:
      Boolean.
    """
    return True
```

Τέλος ακολουθεί ο source code για την κατανομή των εργασιών σε ένα μηχάνημα: tf.distribute.MirroredStrategy

```
"""Class MirroredStrategy implementing tf.distribute.Strategy."""


from __future__ import absolute_import
from __future__ import division
from __future__ import print_function


import contextlib
import copy
import threading


from tensorflow.python import pywrap_tensorflow
from tensorflow.python.distribute import cross_device_ops as
cross_device_ops_lib
from tensorflow.python.distribute import device_util
from tensorflow.python.distribute import distribute_lib
from tensorflow.python.distribute import input_lib
from tensorflow.python.distribute import multi_worker_util
from tensorflow.python.distribute import numpy_dataset
from tensorflow.python.distribute import reduce_util
from tensorflow.python.distribute import shared_variable_creator
from tensorflow.python.distribute import values
from tensorflow.python.distribute.cluster_resolver import
TFConfigClusterResolver
from tensorflow.python.eager import context
from tensorflow.python.eager import tape
from tensorflow.python.framework import constant_op
from tensorflow.python.framework import device as tf_device
from tensorflow.python.framework import dtypes
from tensorflow.python.framework import ops
from tensorflow.python.framework import tensor_util
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import control_flow_ops
from tensorflow.python.ops import variable_scope
from tensorflow.python.platform import tf_logging as logging
from tensorflow.python.training import coordinator
from tensorflow.python.util import nest
from tensorflow.python.util.tf_export import tf_export
```

```
# TODO(josh11b): Replace asserts in this file with if ...: raise ...



@contextlib.contextmanager
def _enter_graph(g, eager, creator_stack=None):
  """Context manager for selecting a graph and maybe eager mode."""
  if eager:
    with g.as_default(), context.eager_mode():
      if creator_stack is not None:
        g._variable_creator_stack = creator_stack  # pylint:
disable=protected-access
      yield
  else:
    with g.as_default():
      if creator_stack is not None:
        g._variable_creator_stack = creator_stack  # pylint:
disable=protected-access
      yield



def _cpu_device(device):
  cpu_device = tf_device.DeviceSpec.from_string(device)
  cpu_device = cpu_device.replace(device_type="CPU", device_index=0)
  return cpu_device.to_string()



class _RequestedStop(Exception):  # pylint: disable=g-bad-exception-name
  pass



# _call_for_each_replica is not a member of MirroredStrategy so that it is
# not allowed to use anything specific to MirroredStrategy and thus
# can be shared with other distribution strategies.
```

```
# TODO(yuefengz): maybe create a common class for those who need to call this
# _call_for_each_replica.
def _call_for_each_replica(distribution, device_map, fn, args, kwargs):
  """Run `fn` in separate threads, once per replica/worker device.
  Args:
    distribution: the DistributionStrategy object.
    device_map: the DeviceMap with the devices to run `fn` on.
    fn: function to run (will be run once per replica, each in its own
thread).
    args: positional arguments for `fn`
    kwargs: keyword arguments for `fn`.
  Returns:
    Merged return value of `fn` across all replicas.
  Raises:
    RuntimeError: If fn() calls get_replica_context().merge_call() a different
        number of times from the available devices.
  """
  # TODO(josh11b): Add this option once we add synchronization to variable
  # creation. Until then, this is pretty unsafe to use.
  run_concurrently = False
  if not context.executing_eagerly():
    # Needed for per-thread device, etc. contexts in graph mode.
    ops.get_default_graph().switch_to_thread_local()


  coord =
coordinator.Coordinator(clean_stop_exception_types=(_RequestedStop,))


  shared_variable_store = {}


  # TODO(isaprykin): Create these threads once instead of during every call.
  threads = []
  for index in range(device_map.num_replicas_in_graph):
    variable_creator_fn = shared_variable_creator.make_fn(
        shared_variable_store, index)
    t = _MirroredReplicaThread(
        distribution, coord, index, device_map, variable_creator_fn, fn,
        values.select_replica(index, args),
        values.select_replica(index, kwargs))
    threads.append(t)


  for t in threads:
```

```
  t.start()


# When `fn` starts `should_run` event is set on _MirroredReplicaThread
# (`MRT`) threads. The execution waits until
# `MRT.has_paused` is set, which indicates that either `fn` is
# complete or a `get_replica_context().merge_call()` is called.  If `fn` is
# complete, then `MRT.done` is set to True.  Otherwise, arguments
# of `get_replica_context().merge_call` from all paused threads are grouped
# and the `merge_fn` is performed.  Results of the
# `get_replica_context().merge_call` are then set to `MRT.merge_result`.
# Each such `get_replica_context().merge_call` call returns the
# `MRT.merge_result` for that thread when `MRT.should_run` event
# is reset again. Execution of `fn` resumes.


try:
  with coord.stop_on_exception():
    all_done = False
    while not all_done and not coord.should_stop():
      done = []
      if run_concurrently:
        for t in threads:
          t.should_run.set()
        for t in threads:
          t.has_paused.wait()
          t.has_paused.clear()
          if coord.should_stop():
            return None
          done.append(t.done)
      else:
        for t in threads:
          t.should_run.set()
          t.has_paused.wait()
          t.has_paused.clear()
          if coord.should_stop():
            return None
          done.append(t.done)
      if coord.should_stop():
        return None
      all_done = all(done)
      if not all_done:
        if any(done):
          raise RuntimeError("Some replicas made a different number of "
                             "replica_context().merge_call() calls.")
        # get_replica_context().merge_call() case
```

```
            merge_args = values.regroup(
                device_map, tuple(t.merge_args for t in threads))
            merge_kwargs = values.regroup(
                device_map, tuple(t.merge_kwargs for t in threads))
            # We capture the name_scope of the MRT when we call merge_fn
            # to ensure that if we have opened a name scope in the MRT,
            # it will be respected when executing the merge function. We only
            # capture the name_scope from the first MRT and assume it is
            # the same for all other MRTs.
            mtt_captured_name_scope = threads[0].captured_name_scope
            mtt_captured_var_scope = threads[0].captured_var_scope
            # Capture and merge the control dependencies from all the threads.
            mtt_captured_control_deps = set()
            for t in threads:
              mtt_captured_control_deps.update(t.captured_control_deps)
            with ops.name_scope(mtt_captured_name_scope),\
                ops.control_dependencies(mtt_captured_control_deps), \
                variable_scope.variable_scope(mtt_captured_var_scope):
              merge_result = threads[0].merge_fn(distribution, *merge_args,
                                                 **merge_kwargs)
            for r, t in enumerate(threads):
              t.merge_result = values.select_replica(r, merge_result)
        finally:
          for t in threads:
            t.should_run.set()
          coord.join(threads)


        return values.regroup(device_map, tuple(t.main_result for t in threads))



    def _is_device_list_local(devices):
      """Checks whether the devices list is for local or multi-worker.
      Args:
        devices: a list of device strings, either local for remote devices.
      Returns:
        a boolean indicating whether these device strings are for local or for
        remote.
      Raises:
        ValueError: if device strings are not consistent.
      """
      all_local = None
      for d in devices:
        d_spec = tf_device.DeviceSpec.from_string(d)
```

```
    is_local = d_spec.job in (None, "localhost")

    if all_local is None:  # Determine all_local from first device.
      all_local = is_local

    if all_local:
      if not is_local:
        raise ValueError("Local device string cannot have job specified other
"
                         "than 'localhost'")
    else:
      if is_local:
        raise ValueError("Remote device string must have job specified.")
      if d_spec.task is None:
        raise ValueError("Remote device string must have task specified.")
  return all_local


def _cluster_spec_to_device_list(cluster_spec, num_gpus_per_worker):
  """Returns a device list given a cluster spec."""
  cluster_spec = multi_worker_util.normalize_cluster_spec(cluster_spec)
  devices = []
  for task_type in ("chief", "worker"):
    for task_id in range(len(cluster_spec.as_dict().get(task_type, []))):
      if num_gpus_per_worker == 0:
        devices.append("/job:%s/task:%d" % (task_type, task_id))
      else:
        devices.extend([
            "/job:%s/task:%d/device:GPU:%i" % (task_type, task_id, gpu_id)
            for gpu_id in range(num_gpus_per_worker)
        ])
  return devices


def _group_device_list(devices):
  """Groups the devices list by task_type and task_id.
  Args:
    devices: a list of device strings for remote devices.
  Returns:
    a dict of list of device strings mapping from task_type to a list of
```

```
devices
    for the task_type in the asceding order of task_id.
  """
  assert not _is_device_list_local(devices)
  device_dict = {}


  for d in devices:
    d_spec = tf_device.DeviceSpec.from_string(d)


    # Create an entry for the task_type.
    if d_spec.job not in device_dict:
      device_dict[d_spec.job] = []


    # Fill the device list for task_type until it covers the task_id.
    while len(device_dict[d_spec.job]) <= d_spec.task:
      device_dict[d_spec.job].append([])


    device_dict[d_spec.job][d_spec.task].append(d)


  return device_dict




def _is_gpu_device(device):
  return tf_device.DeviceSpec.from_string(device).device_type == "GPU"




def _infer_num_gpus_per_worker(devices):
  """Infers the number of GPUs on each worker.
  Currently to make multi-worker cross device ops work, we need all workers to
  have the same number of GPUs.
  Args:
    devices: a list of device strings, can be either local devices or remote
      devices.
  Returns:
    number of GPUs per worker.
  Raises:
    ValueError if workers have different number of GPUs or GPU indices are not
```

```
    consecutive and starting from 0.
  """
  if _is_device_list_local(devices):
    return sum(1 for d in devices if _is_gpu_device(d))
  else:
    device_dict = _group_device_list(devices)
    num_gpus = None
    for _, devices_in_task in device_dict.items():
      for device_in_task in devices_in_task:
        if num_gpus is None:
          num_gpus = sum(1 for d in device_in_task if _is_gpu_device(d))


        # Verify other workers have the same number of GPUs.
        elif num_gpus != sum(1 for d in device_in_task if _is_gpu_device(d)):
          raise ValueError("All workers should have the same number of GPUs.")


        for d in device_in_task:
          d_spec = tf_device.DeviceSpec.from_string(d)
          if (d_spec.device_type == "GPU" and
              d_spec.device_index >= num_gpus):
            raise ValueError("GPU `device_index` on a worker should be "
                             "consecutive and start from 0.")
    return num_gpus




def all_local_devices(num_gpus=None):
  if num_gpus is None:
    num_gpus = context.num_gpus()
  return device_util.local_devices_from_num_gpus(num_gpus)




def all_devices():
  devices = []
  tfconfig = TFConfigClusterResolver()
  if tfconfig.cluster_spec().as_dict():
    devices = _cluster_spec_to_device_list(tfconfig.cluster_spec(),
                                           context.num_gpus())
  return devices if devices else all_local_devices()
```

```
@tf_export("distribute.MirroredStrategy", v1=[])
class MirroredStrategy(distribute_lib.Strategy):
  """Mirrors vars to distribute across multiple devices and machines.
  This strategy uses one replica per device and sync replication for its
  multi-GPU version.
  The multi-worker version will be added in the future.
  Args:
    devices: a list of device strings.  If `None`, all available GPUs are
used.
    If no GPUs are found, CPU is used.
    cross_device_ops: optional, a descedant of `CrossDeviceOps`. If this is
not
      set, nccl will be used by default.
  """


  def __init__(self, devices=None, cross_device_ops=None):
    extended = MirroredExtended(
        self, devices=devices, cross_device_ops=cross_device_ops)
    super(MirroredStrategy, self).__init__(extended)



@tf_export(v1=["distribute.MirroredStrategy"])
class MirroredStrategyV1(distribute_lib.StrategyV1):


  __doc__ = MirroredStrategy.__doc__


  def __init__(self, devices=None, cross_device_ops=None):
    extended = MirroredExtended(
        self, devices=devices, cross_device_ops=cross_device_ops)
    super(MirroredStrategyV1, self).__init__(extended)



# TODO(josh11b): Switch to V2 when we no longer need to support tf.compat.v1.
class MirroredExtended(distribute_lib.StrategyExtendedV1):
  """Implementation of MirroredStrategy."""
```

```
  def __init__(self, container_strategy, devices=None, cross_device_ops=None):
    super(MirroredExtended, self).__init__(container_strategy)
    if context.executing_eagerly():
      if devices and not _is_device_list_local(devices):
        raise RuntimeError("In-graph multi-worker training with "
                           "`MirroredStrategy` is not supported in eager
mode.")
      else:
        if TFConfigClusterResolver().cluster_spec().as_dict():
          # if you are executing in eager mode, only the single machine code
          # path is supported.
          logging.info("Initializing local devices since in-graph multi-worker
"
                       "training with `MirroredStrategy` is not supported in "
                       "eager mode. TF_CONFIG will be ignored when "
                       "when initializing `MirroredStrategy`.")
        devices = devices or all_local_devices()
    else:
      devices = devices or all_devices()

    assert devices, ("Got an empty `devices` list and unable to recognize "
                     "any local devices.")
    self._cross_device_ops = cross_device_ops
    self._initialize_strategy(devices)

    # TODO(b/128995245): Enable last partial batch support in graph mode.
    if ops.executing_eagerly_outside_functions():
      self.experimental_enable_get_next_as_optional = True

  def _initialize_strategy(self, devices):
    # The _initialize_strategy method is intended to be used by distribute
    # coordinator as well.
    assert devices, "Must specify at least one device."
    devices = tuple(device_util.resolve(d) for d in devices)
    assert len(set(devices)) == len(devices), (
        "No duplicates allowed in `devices` argument: %s" % (devices,))
    if _is_device_list_local(devices):
      self._initialize_local(devices)
    else:
      self._initialize_multi_worker(devices)

  def _initialize_local(self, devices):
```

```
    """Initializes the object for local training."""
    self._local_mode = True
    self._device_map = values.ReplicaDeviceMap(devices)
    self._input_workers = input_lib.InputWorkers(self._device_map)
    self._inferred_cross_device_ops = None if self._cross_device_ops else (
        cross_device_ops_lib.choose_the_best(devices))
    self._host_input_device = numpy_dataset.SingleDevice("/cpu:0")


  def _initialize_multi_worker(self, devices):
    """Initializes the object for multi-worker training."""
    self._local_mode = False
    device_dict = _group_device_list(devices)
    workers = []
    worker_devices = []
    for job in ("chief", "worker"):
      for task in range(len(device_dict.get(job, []))):
        worker = "/job:%s/task:%d" % (job, task)
        workers.append(worker)
        worker_devices.append((worker, device_dict[job][task]))


    # Setting `_default_device` will add a device scope in the
    # distribution.scope. We set the default device to the first worker. When
    # users specify device under distribution.scope by
    #   with tf.device("/cpu:0"):
    #     ...
    # their ops will end up on the cpu device of its first worker, e.g.
    # "/job:worker/task:0/device:CPU:0". Note this is not used in replica
mode.
    self._default_device = workers[0]
    self._host_input_device = numpy_dataset.SingleDevice(workers[0])


    self._device_map = values.ReplicaDeviceMap(devices)
    self._input_workers = input_lib.InputWorkers(
        self._device_map, worker_devices)


    if len(workers) > 1:
      if not isinstance(self._cross_device_ops,
                        cross_device_ops_lib.MultiWorkerAllReduce):
        raise ValueError(
            "In-graph multi-worker training with `MirroredStrategy` is not "
            "supported.")
      self._inferred_cross_device_ops = self._cross_device_ops
```

```
  else:
    # TODO(yuefengz): make `choose_the_best` work with device strings
    # containing job names.
    self._inferred_cross_device_ops = cross_device_ops_lib.NcclAllReduce()


def _get_variable_creator_initial_value(self,
                                        replica_id,
                                        device,
                                        primary_var,
                                        **kwargs):
  """Return the initial value for variables on a replica."""
  if replica_id == 0:
    return kwargs["initial_value"]
  else:
    assert primary_var is not None
    assert device is not None
    assert kwargs is not None


    def initial_value_fn():
      if context.executing_eagerly() or ops.inside_function():
        init_value = primary_var.value()
        return array_ops.identity(init_value)
      else:
        with ops.device(device):
          init_value = primary_var.initial_value
          return array_ops.identity(init_value)


    return initial_value_fn


def _create_variable(self, next_creator, *args, **kwargs):
  """Create a mirrored variable. See `DistributionStrategy.scope`."""
  colocate_with = kwargs.pop("colocate_with", None)
  if colocate_with is None:
    device_map = self._device_map
    logical_device = 0  # TODO(josh11b): Get logical device from scope here.
  elif isinstance(colocate_with, numpy_dataset.SingleDevice):
    with ops.device(colocate_with.device):
      return next_creator(*args, **kwargs)
  else:
    device_map = colocate_with.device_map
    logical_device = colocate_with.logical_device
```

```
    def _real_mirrored_creator(devices, *args, **kwargs):  # pylint:
disable=g-missing-docstring
      value_list = []
      for i, d in enumerate(devices):
        with ops.device(d):
          kwargs["initial_value"] = self._get_variable_creator_initial_value(
              replica_id=i,
              device=d,
              primary_var=value_list[0] if value_list else None,
              **kwargs)
          if i > 0:
            # Give replicas meaningful distinct names:
            var0name = value_list[0].name.split(":")[0]
            # We append a / to variable names created on replicas with id > 0
to
            # ensure that we ignore the name scope and instead use the given
            # name as the absolute name of the variable.
            kwargs["name"] = "%s/replica_%d/" % (var0name, i)
          with context.device_policy(context.DEVICE_PLACEMENT_SILENT):
            # Don't record operations (e.g. other variable reads) during
            # variable creation.
            with tape.stop_recording():
              v = next_creator(*args, **kwargs)
          assert not isinstance(v, values.DistributedVariable)
          value_list.append(v)
      return value_list


    return distribute_lib.create_mirrored_variable(
        self._container_strategy(), device_map, logical_device,
        _real_mirrored_creator, values.MirroredVariable,
        values.SyncOnReadVariable, *args, **kwargs)


  def _validate_colocate_with_variable(self, colocate_with_variable):
    values.validate_colocate_distributed_variable(colocate_with_variable,
self)


  def _make_dataset_iterator(self, dataset):
    return input_lib.DatasetIterator(
        dataset,
        self._input_workers,
        self._container_strategy(),
        split_batch_by=self._num_replicas_in_sync)
```

```
def _make_input_fn_iterator(
    self,
    input_fn,
    replication_mode=distribute_lib.InputReplicationMode.PER_WORKER):
  input_contexts = []
  num_workers = self._input_workers.num_workers
  for i in range(num_workers):
    input_contexts.append(distribute_lib.InputContext(
        num_input_pipelines=num_workers,
        input_pipeline_id=i,
        num_replicas_in_sync=self._num_replicas_in_sync))
  return input_lib.InputFunctionIterator(input_fn, self._input_workers,
                                         input_contexts,
                                         self._container_strategy())


def _experimental_distribute_dataset(self, dataset):
  return input_lib.get_distributed_dataset(
      dataset,
      self._input_workers,
      self._container_strategy(),
      split_batch_by=self._num_replicas_in_sync)


def _experimental_make_numpy_dataset(self, numpy_input, session):
  return numpy_dataset.one_host_numpy_dataset(
      numpy_input, self._host_input_device, session)


def _experimental_distribute_datasets_from_function(self, dataset_fn):
  input_contexts = []
  num_workers = self._input_workers.num_workers
  for i in range(num_workers):
    input_contexts.append(distribute_lib.InputContext(
        num_input_pipelines=num_workers,
        input_pipeline_id=i,
        num_replicas_in_sync=self._num_replicas_in_sync))


  return input_lib.get_distributed_datasets_from_function(
      dataset_fn,
      self._input_workers,
      input_contexts,
      self._container_strategy())
```

```
  # TODO(priyag): Deal with OutOfRange errors once b/111349762 is fixed.
  def _experimental_run_steps_on_iterator(self, fn, iterator, iterations,
                                          initial_loop_values=None):
    if initial_loop_values is None:
      initial_loop_values = {}
    initial_loop_values = nest.flatten(initial_loop_values)


    ctx = input_lib.MultiStepContext()
    def body(i, *args):
      """A wrapper around `fn` to create the while loop body."""
      del args
      fn_result = fn(ctx, iterator.get_next())
      for (name, output) in ctx.last_step_outputs.items():
        # Convert all outputs to tensors, potentially from
`DistributedValues`.
        ctx.last_step_outputs[name] = self._local_results(output)
      flat_last_step_outputs = nest.flatten(ctx.last_step_outputs)
      with ops.control_dependencies([fn_result]):
        return [i + 1] + flat_last_step_outputs


    # We capture the control_flow_context at this point, before we run `fn`
    # inside a while_loop. This is useful in cases where we might need to exit
    # these contexts and get back to the outer context to do some things, for
    # e.g. create an op which should be evaluated only once at the end of the
    # loop on the host. One such usage is in creating metrics' value op.
    self._outer_control_flow_context = (
        ops.get_default_graph()._get_control_flow_context())  # pylint:
disable=protected-access


    cond = lambda i, *args: i < iterations
    i = constant_op.constant(0)
    loop_result = control_flow_ops.while_loop(
        cond, body, [i] + initial_loop_values, name="",
        parallel_iterations=1, back_prop=False, swap_memory=False,
        return_same_structure=True)
    del self._outer_control_flow_context


    ctx.run_op = control_flow_ops.group(loop_result)
```

```
    # Convert the last_step_outputs from a list to the original dict structure
    # of last_step_outputs.
    last_step_tensor_outputs = loop_result[1:]
    last_step_tensor_outputs_dict = nest.pack_sequence_as(
        ctx.last_step_outputs, last_step_tensor_outputs)


    for name, reduce_op in ctx._last_step_outputs_reduce_ops.items():  #
pylint: disable=protected-access
      output = last_step_tensor_outputs_dict[name]
      # For outputs that have already been reduced, wrap them in a Mirrored
      # container, else in a PerReplica container.
      if reduce_op is None:
        last_step_tensor_outputs_dict[name] = values.regroup(self._device_map,
                                                             output)
      else:
        assert len(output) == 1
        last_step_tensor_outputs_dict[name] = output[0]


    ctx._set_last_step_outputs(last_step_tensor_outputs_dict)  # pylint:
disable=protected-access
    return ctx


  def _broadcast_to(self, tensor, destinations):
    # This is both a fast path for Python constants, and a way to delay
    # converting Python values to a tensor until we know what type it
    # should be converted to. Otherwise we have trouble with:
    #   global_step.assign_add(1)
    # since the `1` gets broadcast as an int32 but global_step is int64.
    if isinstance(tensor, (float, int)):
      return tensor
    # TODO(josh11b): In eager mode, use one thread per device, or async mode.
    if not destinations:
      # TODO(josh11b): Use current logical device instead of 0 here.
      destinations = values.LogicalDeviceSpec(
          device_map=self._device_map, logical_device=0)
    return self._get_cross_device_ops().broadcast(tensor, destinations)


  def _call_for_each_replica(self, fn, args, kwargs):
    if context.executing_eagerly():
      logging.log_first_n(logging.WARN, "Using %s eagerly has significant "
                          "overhead currently. We will be working on improving
"
```

```
                          "this in the future, but for now please wrap "
                          "`call_for_each_replica` or `experimental_run` or "
                          "`experimental_run_v2` inside a tf.function to get "
                          "the best performance." %
                          self._container_strategy().__class__.__name__, 5)
    return _call_for_each_replica(self._container_strategy(),
self._device_map,
                                  fn, args, kwargs)


  def _configure(self,
                 session_config=None,
                 cluster_spec=None,
                 task_type=None,
                 task_id=None):
    del task_type, task_id


    if session_config:
      session_config.CopyFrom(self._update_config_proto(session_config))


    if cluster_spec:
      # TODO(yuefengz): remove the following code once cluster_resolver is
      # added.
      num_gpus_per_worker = _infer_num_gpus_per_worker(
          self._device_map.all_devices)
      multi_worker_devices = _cluster_spec_to_device_list(
          cluster_spec, num_gpus_per_worker)
      self._initialize_multi_worker(multi_worker_devices)


  def _update_config_proto(self, config_proto):
    updated_config = copy.deepcopy(config_proto)
    updated_config.isolate_session_state = True
    return updated_config


  def _get_cross_device_ops(self):
    return self._cross_device_ops or self._inferred_cross_device_ops


  def _reduce_to(self, reduce_op, value, destinations):
    if (isinstance(value, values.Mirrored) and
        reduce_op == reduce_util.ReduceOp.MEAN):
      return value
```

```
    assert not isinstance(value, values.Mirrored)
    if not isinstance(value, values.DistributedValues):
      # This function handles reducing values that are not PerReplica or
      # Mirrored values. For example, the same value could be present on all
      # replicas in which case `value` would be a single value or value could
      # be 0.
      return cross_device_ops_lib.reduce_non_distributed_value(
          reduce_op, self._device_map, value, destinations)
    return self._get_cross_device_ops().reduce(
        reduce_op, value, destinations=destinations)

  def _batch_reduce_to(self, reduce_op, value_destination_pairs):
    return self._get_cross_device_ops().batch_reduce(reduce_op,
                                                     value_destination_pairs)

  def _update(self, var, fn, args, kwargs, group):
    # TODO(josh11b): In eager mode, use one thread per device.
    assert isinstance(var, values.DistributedVariable)
    updates = []
    for i, (d, v) in enumerate(zip(var.devices, var.values)):
      name = "update_%d" % i
      with ops.device(d), distribute_lib.UpdateContext(d),
ops.name_scope(name):
        # If args and kwargs are not mirrored, the value is returned as is.
        updates.append(fn(v,
                          *values.select_device_mirrored(d, args),
                          **values.select_device_mirrored(d, kwargs)))
    return values.update_regroup(self, self._device_map, updates, group)

  def _update_non_slot(self, colocate_with, fn, args, kwargs, group):
    assert isinstance(colocate_with, tuple)
    # TODO(josh11b): In eager mode, use one thread per device.
    updates = []
    for i, d in enumerate(colocate_with):
      name = "update_%d" % i
      with ops.device(d), distribute_lib.UpdateContext(d),
ops.name_scope(name):
        updates.append(fn(*values.select_device_mirrored(d, args),
                          **values.select_device_mirrored(d, kwargs)))
    return values.update_regroup(self, self._device_map, updates, group)

  def read_var(self, replica_local_var):
```

```
    """Read the aggregate value of a replica-local variable."""
    if isinstance(replica_local_var, values.SyncOnReadVariable):
      return replica_local_var._get_cross_replica()  # pylint:
disable=protected-access
    assert isinstance(replica_local_var, values.Mirrored)
    return array_ops.identity(replica_local_var.get())


  def _local_results(self, val):
    if isinstance(val, values.DistributedValues):
      return val.values
    return (val,)


  def value_container(self, val):
    return values.value_container(val)


  @property
  def _num_replicas_in_sync(self):
    return self._device_map.num_replicas_in_graph


  @property
  def worker_devices(self):
    return self._device_map.all_devices


  @property
  def worker_devices_by_replica(self):
    return self._device_map.devices_by_replica


  @property
  def parameter_devices(self):
    return self._device_map.all_devices


  @property
  def experimental_between_graph(self):
    return False


  @property
  def experimental_should_init(self):
    return True
```

```
  @property
  def should_checkpoint(self):
    return True


  @property
  def should_save_summary(self):
    return True


  def non_slot_devices(self, var_list):
    del var_list
    # TODO(josh11b): Should this be the last logical device instead?
    return self._device_map.logical_to_actual_devices(0)


  # TODO(priyag): Delete this once all strategies use global batch size.
  @property
  def _global_batch_size(self):
    """`make_dataset_iterator` and `make_numpy_iterator` use global batch
size.
    `make_input_fn_iterator` assumes per-replica batching.
    Returns:
      Boolean.
    """
    return True




class _MirroredReplicaThread(threading.Thread):
  """A thread that runs() a function on a device."""


  def __init__(self, dist, coord, replica_id, device_map, variable_creator_fn,
               fn, args, kwargs):
    super(_MirroredReplicaThread, self).__init__()
    self.coord = coord
    self.distribution = dist
    self.device_map = device_map
    self.replica_id = replica_id
    self.variable_creator_fn = variable_creator_fn
    # State needed to run and return the results of `fn`.
    self.main_fn = fn
```

```
self.main_args = args
self.main_kwargs = kwargs
self.main_result = None
self.done = False
# State needed to run the next merge_call() (if any) requested via
# ReplicaContext.
self.merge_fn = None
self.merge_args = None
self.merge_kwargs = None
self.merge_result = None
self.captured_name_scope = None
self.captured_var_scope = None
# We use a thread.Event for the main thread to signal when this
# thread should start running (`should_run`), and another for
# this thread to transfer control back to the main thread
# (`has_paused`, either when it gets to a
# `get_replica_context().merge_call` or when `fn` returns). In
# either case the event starts cleared, is signaled by calling
# set(). The receiving thread waits for the signal by calling
# wait() and then immediately clearing the event using clear().
self.should_run = threading.Event()
self.has_paused = threading.Event()
# These fields have to do with inheriting various contexts from the
# parent thread:
context.ensure_initialized()
ctx = context.context()
self.in_eager = ctx.executing_eagerly()
self.record_thread_local_context_fields()
self.context_device_policy = (
    pywrap_tensorflow.TFE_ContextGetDevicePlacementPolicy(
        ctx._context_handle))
self.graph = ops.get_default_graph()
with ops.init_scope():
  self._init_in_eager = context.executing_eagerly()
  self._init_graph = ops.get_default_graph()


self._variable_creator_stack = self.graph._variable_creator_stack[:]
self._var_scope = variable_scope.get_variable_scope()
# Adding a "/" at end lets us re-enter this scope later.
self._name_scope = self.graph.get_name_scope()
if self._name_scope:
  self._name_scope += "/"
if self.replica_id > 0:
  if not self._name_scope:
    self._name_scope = ""
```

```
    self._name_scope += "replica_%d/" % self.replica_id


def run(self):
  self.should_run.wait()
  self.should_run.clear()
  try:
    if self.coord.should_stop():
      return
    self.restore_thread_local_context_fields()
    # TODO(josh11b): Use current logical device instead of 0 here.
    with self.coord.stop_on_exception(), \
        _enter_graph(self._init_graph, self._init_in_eager), \
        _enter_graph(self.graph, self.in_eager,
                     self._variable_creator_stack), \
        context.device_policy(self.context_device_policy), \
        MirroredReplicaContext(self.distribution, constant_op.constant(
            self.replica_id, dtypes.int32)), \
        ops.device(self.device_map.logical_to_actual_devices(0)[
            self.replica_id]), \
        ops.name_scope(self._name_scope), \
        variable_scope.variable_scope(
            self._var_scope, reuse=self.replica_id > 0), \
        variable_scope.variable_creator_scope(self.variable_creator_fn):
      self.main_result = self.main_fn(*self.main_args, **self.main_kwargs)
      self.done = True
  finally:
    self.has_paused.set()


def record_thread_local_context_fields(self):
  """Record thread local fields of context.context() in self."""
  ctx = context.context()
  self._summary_step = ctx.summary_step
  self._summary_writer = ctx.summary_writer
  self._summary_recording = ctx.summary_recording
  self._summary_recording_distribution_strategy = (
      ctx.summary_recording_distribution_strategy)
  # TODO(b/125892694): record other fields in EagerContext.


def restore_thread_local_context_fields(self):
  """Restore thread local fields of context.context() from self."""
  ctx = context.context()
  ctx.summary_step = self._summary_step
  ctx.summary_writer = self._summary_writer
```

```
    ctx.summary_recording = self._summary_recording
    ctx.summary_recording_distribution_strategy = (
        self._summary_recording_distribution_strategy)
    # TODO(b/125892694): restore other fields in EagerContext.



class MirroredReplicaContext(distribute_lib.ReplicaContext):
  """ReplicaContext used in MirroredStrategy.extended.call_for_each_replica().
  Opened in `_MirroredReplicaThread`, to allow the user to invoke
  `MirroredStrategy`'s specific implementation of `merge_call()`,
  which works by delegating the function and its arguments to
  the main thread (the one that invoked
  `MirroredStrategy.extended.call_for_each_replica()`).
  """


  def _merge_call(self, fn, args, kwargs):
    """Delegate to the main thread to actually perform merge_call()."""
    t = threading.current_thread()  # a _MirroredReplicaThread
    t.merge_fn = fn
    t.merge_args = args
    t.merge_kwargs = kwargs
    t.captured_name_scope = t.graph.get_name_scope()
    # Adding a "/" at end lets us re-enter this scope later.
    if t.captured_name_scope:
      t.captured_name_scope += "/"


    t.captured_var_scope = variable_scope.get_variable_scope()
    t.captured_control_deps = t.graph._current_control_dependencies()  #
pylint: disable=protected-access


    # NOTE(priyag): Throw an error if there is a merge call in the middle of a
    # `fn` passed to call_for_each_replica which changes the graph being used
    # while calling `fn`. This can happen when the `fn` is decorated with
    # `tf.function` and there is a merge_call in `fn`. This breaks because
each
    # thread tries to create a distinct tf.function. Each tf.function creation
    # takes a lock, and so if there is a merge call in the middle, the lock is
    # never released and subsequent replica threads cannot proceed to define
    # their own functions. Checking for the graph being the same is one way
for
    # us to check this didn't happen.
```

```
    if ops.get_default_graph() != t.graph:
      raise RuntimeError(
          "`merge_call` called while defining a new graph or a tf.function. "
          "This can often happen if the function `fn` passed to "
          "`strategy.experimental_run()` is decorated with "
          "`@tf.function` (or contains a nested `@tf.function`), and `fn` "
          "contains a synchronization point, such as aggregating gradients. "
          "This behavior is not yet supported. Instead, please wrap the entire
"
          "call `strategy.experimental_run(fn)` in a `@tf.function`, and avoid
"
          "nested `tf.function`s that may potentially cross a synchronization
"
          "boundary.")


    t.has_paused.set()
    t.should_run.wait()
    t.should_run.clear()
    if t.coord.should_stop():
      raise _RequestedStop()
    return t.merge_result


  @property
  def devices(self):
    distribute_lib.require_replica_context(self)
    replica_id = tensor_util.constant_value(self._replica_id_in_sync_group)
    return [self._strategy.extended.worker_devices_by_replica[replica_id]]
```