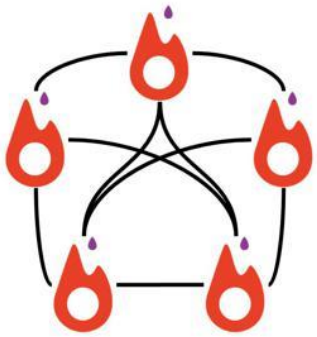
ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Παράλληλη και κατανεμημένη υλοποίηση αλγορίθμων συστάσεων



Του φοιτητή

Βασίλη Μοσχόπουλου

Αρ. Μητρώου: 123890

Επιβλέπων μέλος ΔΕΠ

Δρ. Κωνσταντίνος Ι. Διαμαντάρας
Καθηγητής
Τμήμα Μηχανικών Πληροφορικής ΑΤΕΙΘ

Ιανουάριος 2019

Θεσσαλονίκη

ΠΕΡΙΛΗΨΗ

Τα συστήματα συστάσεων χρησιμοποιούνται σήμερα ευρέως από ένα εύρος πλατφόρμων για την σύσταση προϊόντων και υπηρεσιών σε καταναλωτές, πελάτες και συνδρομητές. Μερικοί από τους στόχους των συστημάτων συστάσεων περιλαμβάνουν την αύξηση των πωλήσεων, την καλύτερη κατανόηση της αγοράς και των καταναλωτών, την βελτίωση εμπειρίας μια πλατφόρμας και την αύξηση της ικανοποίησης των πελατών.

Τα συστήματα συστάσεων έρχονται να παίξουν ένα πολύ μεγάλο ρόλο σε μια εποχή αφθονίας προϊόντων και υπηρεσιών, με στόχο να φιλτράρουν τα προϊόντα και τις υπηρεσίες και να κάνουν καλύτερες προτάσεις στους καταναλωτές, ανάλογα με τα ενδιαφέροντα τους. Μερικά από τα πιο σημαντικά συστήματα συστάσεων μπορεί να ανακαλύψει κανείς σε πλατφόρμες όπως το YouTube της Google, σε μεγάλα ηλεκτρονικά καταστήματα όπως το Amazon και το E-Bay, σε πλατφόρμες κοινωνικής δικτύωσης όπως είναι το Facebook και το Instagram και μουσικές πλατφόρμες όπως για παράδειγμα το Spotify και το Soundcloud.

Ταυτόχρονα, βιβλιοθήκες όπως η numpy, pandas και scikit-learn μπορούν να χρησιμοποιηθούν για την υλοποίηση αλγορίθμων συστάσεων σε λογισμικό. Σε αυτή την εργασία μελετάται εκτενώς η βιβλιοθήκη μηχανικής μάθησης PyTorch. Η PyTorch αποτελεί μια νέα πρόσθεση στο οικοσύστημα της βαθιάς μάθησης.

Σε μια εποχή που τα δεδομένα όλο και αυξάνονται η χρήση cluster υπολογιστών για την παράλληλη κατανεμημένη επεξεργασία των δεδομένων αυτών γίνεται μια όλο και πιο διαδεδομένη πρακτική.

Σε αυτή την εργασία θα παρουσιασθούν αναλυτικά και θα υλοποιηθούν αλγόριθμοι συστημάτων συστάσεων, ειδικότερα του είδους συνεργατικού φιλτραρίσματος, με την χρήση της βιβλιοθήκης PyTorch, και θα μελετηθούν και εκπαιδευθούν μοντέλα των αλγορίθμων αυτών, με απλό τρόπο, παράλληλα, αλλά και παράλληλα κατανεμημένα, με την χρήση διεργασιών. Τέλος, θα αξιολογηθούν, ως προς την ακρίβεια και την ορθότητα τους.

ABSTRACT

Recommendation systems are widely used today by a range of platforms for product and service recommendations to consumers, customers and subscribers. Recommendation systems' goals include increasing sales, understanding the market, as well as the consumers, and increasing the experience quality of platforms and customer satisfaction.

Recommendation systems have come to play an important role in an era of product and service abundance, with the goal of filtering products and services and increasing recommendation quality to consumers, depending on their interests. Notable recommendation systems can be found in platforms such as Google's YouTube, large e-shops such as Amazon and E-Bay, social platforms in the likes of Facebook and Instagram and music platforms, for example, Spotify and Soundcloud.

Meanwhile, libraries like numpy, pandas and scikit-learn can be used to implement recommendation algorithms in software. In this thesis we study the machine-learning library, PyTorch. PyTorch is a new addition to the deep learning framework.

At a time when data become increasingly larger, usage of computer clusters for the parallel distributed processing of data becomes a growing common trend.

In this thesis we analytically present and implement recommendation system algorithms, specifically the collaborative filtering kind, with the use of PyTorch. We study and train models of these algorithms, in a simple manner, as well as in a parallel and parallel distributed way, with the use of processes, and conclude by evaluating these algorithms in terms of accuracy and correctness.

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω πολύ τον επιβλέπων καθηγητή μου, κ. Διαμαντάρα, για την πρόταση ενός επίκαιρου και απαιτητικού θέματος πάνω στα παράλληλα συστήματα και τις συμβουλές του, τον φίλο μου Νικόλαο-Αλέξανδρο Τσακαλωφά, τελειόφοιτο φοιτητή του τμήματος Μαθηματικών του ΑΠΘ, για την βοήθεια του κατά την πρώτη μου επαφή με τους αλγορίθμους συστάσεων, τον κ. Παντελή Καπλάνογλου για τη συνεργασία και την παροχή περιβάλλοντος στους υπολογιστές που χρησιμοποίησα κατά τις δοκιμές μου με την πτυχιακή μου εργασία, τον Γιάννη Σταματούλη, φίλο και πνευματικό αδερφό, φοιτητή του τμήματος Στατιστικής του Οικονομικού Πανεπιστήμιου Αθηνών, για την υποστήριξη του κατά το εξάμηνο αυτό, και τους γονείς και φίλους μου που με υπέμειναν καθ' όλη τη διάρκεια της πτυχιακής εργασίας μου.

Η πτυχιακή αυτή εργασία είναι αφιερωμένη σε όσους πίστεψαν σε εμένα και συνεχίζουν να με στηρίζουν, και κυρίως, στον εαυτό μου.

ΠΕΡΙΕΧΟΜΕΝΑ

Ευρετήριο εικόνων

Ευρετήριο κωδίκων

ΚΕΦΑΛΑΙΟ 1 – Εισαγωγή

1.1 – Εισαγωγή στους αλγορίθμους συστάσεων και την βιβλιοθήκη μηχανικής μάθησης PyTorch

Recommendation Systems και Collaborative filtering

Τα συστήματα συστάσεων είναι εργαλεία λογισμικού που χρησιμοποιούνται για την πρόταση προϊόντων και υπηρεσιών, όπως είναι για παράδειγμα μια ταινία, ένα βιβλίο, ένα ταξίδι διακοπών ή μια συναυλία, σε πελάτες ή συνδρομητές, προϊόντα και υπηρεσίες τα οποία κρίνονται πως τους ενδιαφέρουν.

Στα συστήματα συστάσεων ένα προϊόν ή υπηρεσία ονομάζεται αντικείμενο, ενώ ένας πελάτης ή συνδρομητής ονομάζεται χρήστης. Στην πραγματική ζωή, παραδείγματα συστημάτων συστάσεων περιλαμβάνουν τον αλγόριθμο σύστασης ταινιών του Netflix, το σύστημα σύστασης φίλων του Facebook, το σύστημα σύστασης διαφημιστικών της Google όπως και το σύστημα συστάσεων βίντεο του YouTube.

Ένα από τα είδη συστημάτων συστάσεων, το οποίο και μελετάται σε αυτή την εργασία, είναι τα συστήματα συστάσεων συνεργατικού φιλτραρίσματος ή αλλιώς, Collaborative Filtering. Τα συστήματα συστάσεων συνεργατικού φιλτραρίσματος χρησιμοποιούν ήδη υπάρχουσες βαθμολογήσεις αντικειμένων από χρήστες για την πρόβλεψη νέων βαθμολογήσεων.

Δύο είδη συστημάτων συνεργατικού φιλτραρίσματος εξετάζονται σε αυτή την εργασία, τα συστήματα γειτονιών, ή αλλιώς βασισμένων σε μνήμη (Neighborhood/Memory-based), και τα συστήματα παραγοντοποίησης (Latent Factor) βασισμένων σε μάθηση, ή αλλιώς βασισμένων σε μοντέλα (Learning/Model-based).

PyTorch

Η PyTorch αποτελεί μια βιβλιοθήκη βαθιάς μάθησης για την γλώσσα Python. Η PyTorch χρησιμοποιεί μια δομή παρόμοια με τα arrays της Numpy, τους tensors (τανυστές), αλλά και έτοιμα εργαλεία για την υλοποίηση Datasets, στρωμάτων νευρωνικών δικτύων, αυτοματοποιημένης εκπαίδευσης και πολλών άλλων.

Ένα πλεονέκτημα της σε σχέση με άλλες βιβλιοθήκες, όπως για παράδειγμα η scikit-learn, είναι η δυνατότητα αξιοποίησης καρτών γραφικών γενικού σκοπού της nVidia για υπολογισμούς μεταξύ tensors, πράγμα που μπορεί να επιταχύνει την διαδικασία μάθησης ενός μοντέλου έως και εκατοντάδες φορές, δεδομένου ότι έχουμε στην διάθεση μας μια αξιόλογη κάρτα γραφικών.

Κατανεμημένες εφαρμογές σε PyTorch

Σε αυτή την εργασία θα μελετηθεί και θα αξιοποιηθεί εκτενώς το πακέτο distributed της PyTorch.

Εμπνευσμένο από το MPI (Message Passing Interface), το πακέτο distributed της PyTorch, προβλέποντας πιθανές ανάγκες για χρήση σε παραγωγή, υποστηρίζει την επικοινωνία μεταξύ διαφορετικών διεργασιών, όχι μόνο τοπικά, αλλά και μεταξύ διεργασιών που μπορεί να εδρεύουν σε διαφορετικά μηχανήματα μέσα σε ένα δίκτυο, όπως είναι για παράδειγμα ένα cluster υπολογιστών.

Στόχος είναι λοιπόν η υλοποίηση αλγορίθμων συστάσεων με την χρήση της βιβλιοθήκης PyTorch και η συνεργατική εκπαίδευση μοντέλων αυτών, μέσω διεργασιών που εδρεύουν σε διαφορετικές μηχανήματα.

Θα χρησιμοποιηθούν ένα κοινό μοντέλο και κοινά training και test datasets. Το μοντέλο θα εκπαιδευτεί συνεργατικά από τις διεργασίες και έπειτα θα αξιολογηθεί. Θα υλοποιηθεί ακόμη ένας αλγόριθμος δημιουργίας διεργασιών, τοπικά, με τη χρήση του πακέτου multiprocessing, επέκταση της βιβλιοθήκης multiprocessing της Python.

Τέλος, ο αλγόριθμος θα αξιολογηθεί ως προς την ακρίβεια του και θα βρεθούν κατάλληλες παράμετροι για την ελαχιστοποίηση του σφάλματος αξιολόγησης.

1.2 – Στόχοι και σκοποί της πτυχιακής εργασίας

Στόχος είναι η υλοποίηση δύο ειδών κόμβων, έτσι ώστε να είναι δυνατή η επικοινωνία τους μέσω δικτύου, για την συνεργατική διαδικασία εκπαίδευσης ενός μοντέλου αλγορίθμου συστάσεων, βασισμένο σε μάθηση. Οι κόμβοι μπορούν να εδρεύουν στον ίδιο υπολογιστή (διαφορετικές διεργασίες), σε διαφορετικούς εντός τοπικού δικτύου, αλλά και σε διαφορετικούς εντός διαφορετικών δικτύων, και

μπορούν να είναι οποιουδήποτε αριθμού αλλά και να επικοινωνήσουν ανεξαρτήτως είδους.

Η εκπαίδευση και αποθήκευση των μοντέλων αλγορίθμων μπορεί ακόμη να λαμβάνει χώρα στην κύρια μνήμη του υπολογιστή ή την μνήμη καρτών γραφικών γενικού σκοπού (General Purpose GPU) με την χρήση της τεχνολογίας CUDA.

Σκοποί αυτής της πτυχιακής εργασίας είναι η μελέτη και η κατανόηση της βιβλιοθήκης PyTorch, η μελέτη και η ανάπτυξη αλγορίθμων συνεργατικού φιλτραρίσματος (collaborative filtering) βασισμένων στον χρήστη (user-based) και στο αντικείμενο (item-based), η μελέτη και η χρήση του distributed communication package της PyTorch και η υλοποίηση αλγορίθμων συστάσεων, βασισμένων σε μάθηση, έτσι ώστε να μπορούν να υλοποιηθούν σε μία συστάδα υπολογιστών.

Στους σκοπούς περιλαμβάνεται ακόμη η μελέτη της βιβλιοθήκης multiprocessing της Python, η οποία θα χρησιμοποιηθεί κατά την παράλληλη κατανεμημένη εκπαίδευση μοντέλων.

1.3 – Δομή της πτυχιακής εργασίας

Στο Κεφάλαιο 2 θα παρουσιασθούν και θα αναλυθούν αρχικά αλγόριθμοι συστημάτων συστάσεων συνεργατικού φιλτραρίσματος. Θα παρουσιασθεί ένα παράδειγμα αλγορίθμου σύστασης γειτονιάς σε Python καθώς και άλλες μέθοδοι συνοπτικά.

Έπειτα, στην ενότητα 3.2, θα γίνει εισαγωγή στην βιβλιοθήκη PyTorch και στο πως μπορεί κανείς να την χρησιμοποιήσει για να φτιάξει και να εκπαιδεύσει ένα δικό του μοντέλο και θα δωθούν συμβουλές για την καλύτερη χρήση της PyTorch και την αποσφαλμάτωση εφαρμογών. Θα δοθεί ακόμη παράδειγμα υλοποίησης αλγορίθμου συστάσεων παραγοντοποίησης με την χρήση της βιβλιοθήκης PyTorch.

Στην ενότητα 3.3 θα αναλύσουμε το πακέτο distributed της PyTorch, θα δείξουμε πως μπορεί κανείς να φτιάξει δικές του κατανεμημένες εφαρμογές με τη χρήση της PyTorch και θα δώσουμε μερικές ακόμη συμβουλές για την καλύτερη υλοποίηση και αποσφάλματωση κατανεμημένων εφαρμογών.

Στο Κεφάλαιο 4 θα παρουσιάσουμε δύο κόμβους συνεργατικής εκπαίδευσης ενός μοντέλου με την βοήθεια του πακέτου distributed της PyTorch, θα αποδείξουμε την

ορθότητα της εκπαίδευσης προσπαθώντας να αναπαράξουμε το αποτέλεσμα σε διαφορετικά μηχανήματα, θα αξιολογήσουμε τον αλγόριθμο και θα επιχειρήσουμε να βρούμε κατάλληλες παραμέτρους για την ελαχιστοποίηση του σφάλματος αξιολόγησης.

Τέλος, στο Κεφάλαιο 5, θα παραθέσουμε συμπεράσματα και πιθανές μελλοντικές επεκτάσεις της πτυχιακής αυτής εργασίας.

ΚΕΦΑΛΑΙΟ 2 – Συστήματα συστάσεων συνεργατικού φιλτραρίσματος (Collaborative Filtering)

2.1 – Εισαγωγή

Τα συστήματα συστάσεων είναι εργαλεία λογισμικού και τεχνικές συστάσεων αντικειμένων σε χρήστες, τα οποία κρίνονται πιθανότερο να αφορούν τα ενδιαφέροντα των χρηστών, με βάση τις προτιμήσεις που έχουν δηλώσει, άμεσα ή έμμεσα. [1]

Αντικείμενο ονομάζεται τι ένα σύστημα συστάσεων προτείνει γενικότερα στους χρήστες. Ένα αντικείμενο λοιπόν, θα μπορούσε να είναι ένα βιβλίο, μία ταινία, ένα βίντεο κτλ. Σκοπός είναι η πρόταση ενός αντικειμένου σε ένα χρήστη για λόγους όπως: αύξηση του αριθμού πωλήσεων, πώληση διαφορετικών και ποικίλων αντικειμένων, αύξηση της ικανοποίησης των χρηστών κ.α.

Τα συστήματα συστάσεων μπορούν να χρησιμοποιούν δεδομένα, δηλωμένα κατηγορηματικά, είτε από τους χρήστες, είτε για τα αντικείμενα, ή έμμεσα, όπως είναι πχ. η προτίμηση του χρήστη με βάση ποια αντικείμενα έχει επιλέξει να σχολιάσει, να νοικιάσει, να αγοράσει κοκ.

Παραδείγματα συστημάτων συστάσεων στην καθημερινή μας ζωή αποτελούν ο αλγόριθμος συστάσεων του YouTube, οι αλγόριθμοι συστάσεων διαφημίσεων και φίλων του Facebook, ο αλγόριθμος σύστασης ταινιών του Netflix (αξίζει να σημειωθεί το βραβείο Netflix Prize) κτλ.

Από όλες τις διαφορετικές τεχνικές, σε αυτό το κεφάλαιο (και την πτυχιακή εργασία), θα εξεταστούν οι τεχνικές συνεργατικού φιλτραρίσματος (Collaborative Filtering). Για την περίπτωση τεχνικών βασισμένες σε χρήστες για παράδειγμα, οι τεχνικές συνεργατικού φιλτραρίσματος αναγνωρίζουν τις προτιμήσεις χρηστών και προτείνουν σε έναν χρήστη αντικείμενα, τα οποία χρήστες με παρόμοια ενδιαφέροντα, τα οποία κρίνονται από τις κοινές προτιμήσεις τους βάσει άλλων αντικείμενα, τείνουν να αρέσκονται.

Θα ξεκινήσουμε με την μελέτη και ανάπτυξη των μεθόδων γειτονιών και έπειτα θα μας απασχολήσουν οι μέθοδοι παραγοντοποίησης βασισμένες στην μάθηση, μία

από τις οποίες θα χρησιμοποιήσουμε για την εκπαίδευση και παράλληλη κατανεμημένη εκπαίδευση ενός μοντέλου, με την υλοποίηση επικοινωνούντων κόμβων, αργότερα σε επόμενο κεφάλαιο.

2.2 – Μέθοδοι γειτονιών (Neighborhood based)

Οι μέθοδοι συνεργατικού φιλτραρίσματος μπορούν να διαφοροποιηθούν σε δύο κατηγορίες: μέθοδοι γειτονιών και μέθοδοι βασισμένες σε μάθηση ή αλλιώς, μέθοδοι βασισμένες σε μοντέλα. [1][2]

Οι μέθοδοι γειτονιών, οι οποίες εξετάζονται σε αυτό το κεφάλαιο, βασίζονται στην ιδέα γειτονικών χρηστών η αντικειμένων. Κάθε αντικείμενο ή χρήστης έχει ένα αριθμό γειτόνων, οι οποίοι ορίζονται από την ύπαρξη κοινών μοτίβων βαθμολόγησης ανάμεσα στους δύο γείτονες. Έτσι, η πρόβλεψη πρότασης ενός αντικειμένου μπορεί να γίνει απευθείας, μόνο με την εύρεση παρόμοιων χρηστών ή παρόμοιων αντικειμένων με τα αντικείμενα προτίμησης του χρήστη, και την πρόταση αντικειμένων που αρέσουν σε παρόμοιους χρήστες ή αντικειμένων σχετικών με τα ενδιαφέροντα του χρήστη.

Από την άλλη, σε αντίθεση με τις μεθόδους γειτονιών, οι μέθοδοι βασισμένες σε μάθηση, εκπαιδεύουν ένα μοντέλο πρόβλεψης συστάσεων μέσω μιας διαδικασίας μάθησης και μαθαίνουν να κάνουν καλύτερες συστάσεις σε βάθος χρόνου.

2.2.1 – Πλεονεκτήματα μεθόδων γειτονιών

Παρά το γεγονός ότι νεότερες μέθοδοι βασισμένες σε μάθηση μπορούν να προσφέρουν ακριβέστερα αποτελέσματα, οι μέθοδοι γειτονιών προσφέρουν χρήσιμα χαρακτηριστικά, τα οποία οι μέθοδοι βασισμένες σε γειτονιές δεν προσφέρουν, ενώ τίθεται ακόμη το ζήτημα της έννοιας serendipity (καλή εύνοια της τύχης). Η έννοια serendipity αναφέρεται στο γεγονός ανακάλυψης ενός αντικειμένου που διαφορετικά, ο χρήστης ίσως να μην είχε ανακαλύψει, μέσω ενός αλγορίθμου βασισμένου στη μάθηση, ο οποίος στην περίπτωση αυτή, θα έκανε μια σύσταση «κομμένη και ραμμένη» στις μέχρι τώρα προτιμήσεις του χρήστη.

Στα πλεονεκτήματα των μεθόδων γειτονιών συγκαταλέγονται [1]:

- Απλότητα: Οι μέθοδοι γειτονιών είναι διαισθητικά κατανοητές και εύκολες στην υλοποίηση.
- Αιτιολόγηση: Οι συστάσεις που γίνονται έχουν μια περιεκτική και διαισθητική αιτιολόγηση, η οποία στην περίπτωση μιας σύστασης ενός αντικειμένου σε ένα χρήστη, μπορεί να συμπεριληφθεί στην περιγραφή.
- Αποδοτικότητα: Οι μέθοδοι γειτονιών σε αντίθεση με τις μεθόδους βασισμένων σε μάθηση, δεν περιλαμβάνουν δαπανηρές διαδικασίες μάθησης και εκπαίδευσης, οι οποίες μπορούν να καταλάβουν ένα σημαντικό αριθμό πόρων.
- Σταθερότητα: Δεν επηρεάζονται ιδιαίτερα από την συνεχή πρόσθεση νέων χρηστών και αντικειμένων.

2.2.2 – Μονάδες μέτρησης ακρίβειας

Έστω U το σύνολο των χρηστών, I το σύνολο των αντικειμένων, R το σύνολο των βαθμολογήσεων από τους χρήστες για τα αντικείμενα και S το σύνολο των τιμών που μπορεί να πάρει μια βαθμολόγηση πχ. $S = \{1, ..., 5\}$ ή $S = [0\ (\text{dislike}), 1\ (\text{like})]$. Κάθε χρήστης $u \in U$ μπορεί να βαθμολογήσει ένα αντικείμενο $i \in I$ μόνο μια φορά, έστω r_{ui} η βαθμολόγηση αυτή. Το σύνολο των χρηστών που έχουν βαθμολογήσει ένα αντικείμενο i ορίζεται ως U_i, παρομοίως I_u για το σύνολο αντικείμενων που έχουν βαθμολογηθεί από έναν χρήστη u. Τέλος, τα αντικείμενα που έχουν βαθμολογηθεί από δύο χρήστες, u και v, $I_u \cap I_v$, αναπαρίστανται ως I_{vu}. Παρομοίως, U_{it} οι χρήστες που έχουν βαθμολογήσει δύο αντικείμενα, i και ι. [1][2]

Έστω f η εξίσωση: $U * I -> S$ η οποία προβλέπει την βαθμολογία $f(u,i)$ ενός χρήστη u για ένα αντικείμενο i. Τυπικά, σε τέτοια προβλήματα, όπως και εδώ, οι βαθμολογήσεις R διαχωρίζονται σε ένα σετ εκπαίδευσης (training set) R_{train} και ένα σετ εκτίμησης (test set) R_{test}, με το ένα να χρησιμοποιείται για την μάθηση της εξίσωσης f και το άλλο για την εκτίμηση της επίδοσης της. [1]

Μία από τις μονάδες μέτρηση ακρίβειας και εκτίμησης της επίδοσης της f, η MAE (Mean Absolute Error) ορίζεται ως το μέσο απόλυτο σφάλμα, ανάμεσα στην πρόβλεψη και την πραγματική τιμή μιας σύστασης για το σετ βαθμολογήσεων R_{test}: [1]

$$MAE(f) = 1/|R_{test}| * \Sigma_{u \in U} |f(u,i) - r_{ui}| \quad (2.1)$$

Ενώ η RMSE (Root Mean Squared Error) ορίζεται ως η ρίζα του μέσου τετραγωνικού σφάλματος [1]:

$$RMSE(f) = (1/|R_{test}| * \Sigma_{u \in U} (f(u,i) - r_{ui})^2)^{1/2} \quad (2.2)$$

Έστω ακόμη μια λίστα αντικειμένων, που ταξινομούνται ανάλογα με το ενδιαφέρον, για ένα χρήστη u, $L(u)$. Χτίζοντας το test set, εάν επιλέγεται μονάχα ένα αντικείμενο i_u το οποίο ανήκει στα αντικείμενα που έχει βαθμολογήσει ο χρήστης αυτός ($i_u \in I_u$), μία μονάδα μέτρησης είναι η ARHR (Average Reciprocal Hit-Rate) [1]:

$$ARHR(L) = 1/|U| * \Sigma_{u \in U}(1/rank(i_u, L(u)) \quad (2.3)$$

Στα επόμενα παραδείγματα θα χρησιμοποιήσουμε την MSE (Mean Squared Error), η οποία είναι παρόμοια με την RMSE (Root Mean Squared Error).

Παρακάτω δίνεται ένα πολύ απλό παράδειγμα κώδικα για τον υπολογισμό της MSE, ανάμεσα στην τιμή μίας πρόβλεψης βαθμολόγησης και την πραγματική τιμή αυτής της βαθμολόγησης, με τη χρήση της βιβλιοθήκης scikit-learn σε Python.

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(prediction, actual)
```

Κώδικας 1 – Mean squared error

2.2.3 – Μέθοδοι βασισμένες στον χρήστη (User-based)

Οι μέθοδοι βασισμένες στον χρήστη προβλέπουν την βαθμολόγηση ενός αντικειμένου από έναν χρήστη με βάση τους κοντινότερους (περισσότερο παρόμοιους) γείτονες (χρήστες) αυτού του χρήστη. [1][2]

Έστω τέσσερις χρήστες: Γιάννης, Αγγελική, Χρήστος και Σμαράγδα. Και τέσσερις ταινίες: Fight Club, Lion King, Stardust και Guardians of the Galaxy. Παρακάτω δίνονται οι βαθμολογήσεις των χρηστών αυτών για τις συγκεκριμένες ταινίες:

Πίνακας 1 – Παράδειγμα βαθμολόγησης χρηστών-ταινιών

	Fight Club	Lion King	Stardust	Guardians of the Galaxy
Γιάννης	4	4	2	3
Αγγελική	2	4	5	3
Χρήστος		5	3	2
Σμαράγδα	1	4		3

Έστω πως:

Ο Χρήστος, 22 ετών, δεν έχει παρακολουθήσει την ταινία Fight Club, λόγω του γεγονότος ότι είναι μια παλιά ταινία του 1999, ενώ η Σμαράγδα δεν έχει παρακολουθήσει την ταινία Stardust, καθώς κάθε φορά που την είχε στην τηλεόραση, η Σμαράγδα αποκοιμιόταν στον καναπέ λόγω του γεμάτου και κουραστικού ωραρίου της. Η Σμαράγδα ξέροντας ότι η Αγγελική έχει ίδια γούστα με αυτή και ότι βαθμολόγησε την ταινία Stardust πολύ υψηλά, ίσως να έπαιρνε τον χρόνο να την δει σε ένα από τα ρεπό της. Από την άλλη, ο Χρήστος έχοντας παρόμοια γούστα με το Γιάννη, πιθανόν να καθόταν ένα Σαββατόβραδο αργά με την κοπέλα του, η οποία έχει τα ίδια γούστα με τον Χρήστο, για να παρακολουθήσουν την ταινία Fight Club. Με τον ίδιο τρόπο, οι αλγόριθμοι σύστασης γειτονιών, κάνουν συστάσεις σε χρήστες για αντικείμενα που δεν έχουν βαθμολογήσει.

Ας υποθέσουμε ότι για οποιουσδήποτε δύο χρήστες, u και v, υπάρχει μια τιμή ομοιότητας w_{uv}, που αναπαριστά το βαθμό ομοιότητας ανάμεσα στους δύο χρήστες, και $N_i(u)$ το σύνολο των k κοντινότερων προς τον u, χρηστών v που έχουν βαθμολογήσει ένα κοινό αντικείμενο i που ο u δεν έχει βαθμολογήσει. Την προβλεπόμενη λοιπόν βαθμολογία του αντικειμένου i από τον χρήστη u, r_{ui}, μπορούμε να υπολογίσουμε ως εξής [1]:

$$\overline{r_{ui}} = \Sigma_{v \in Ni(u)} (w_{uv} * r_{ui}) \,/\, \Sigma_{v \in Ni(u)} |w_{uv}| \qquad (2.4)$$

Για την παραπάνω περίπτωση:

Ας θεωρήσουμε ότι θέλουμε να προβλέψουμε την βαθμολόγηση του Χρήστου για την ταινία Fight Club και ότι οι δύο κοντινότεροι γείτονες προς τον Χρήστο σε ένα σύστημα σύστασης είναι η Αγγελική και ο Γιάννης. Δίνονται οι τιμές ομοιότητας ανάμεσα στον Χρήστο και την Αγγελική και ανάμεσα στον Χρήστο και τον Γιάννη, οι οποίες είναι w_{XA} = 0.35 και $w_{XΓ}$ = 0.85 αντίστοιχα. Δεδομένου ότι η βαθμολογία της Αγγελικής για την ταινία Fight Club ήταν 2 και του Γιάννη 4, μπορούμε να προβλέψουμε την βαθμολόγηση του Χρήστου ως εξής:

$$r = (0.35 * 2 + 0.85 * 4) / (0.35 + 0.85) = 3.146$$

Κανονικοποίηση

Παρά το γεγονός ότι η παραπάνω πρόβλεψη μοιάζει να έγινε αντικειμενικά σωστά, πρέπει να ληφθεί ακόμη υπόψη το γεγονός ότι κάθε χρήστης δεν βαθμολογεί με τον ίδιο τρόπο. Για παράδειγμα, ένα χρήστης x μπορεί συχνά να βαθμολογεί ταινίες με μικρότερο ενθουσιασμό, δίνοντας πολλές φορές ούτε ιδιαίτερα αρνητικές βαθμολογήσεις, πχ. 1, αλλά ούτε και εξαιρετικά υψηλές, πχ. 5, ενώ από την άλλη ένας πιο ενθουσιώδης χρήστης y μπορεί να βαθμολογεί όλες τις ταινίες με βαθμολογήσεις πάντοτε μεγαλύτερες του 3.

Η κανονικοποίηση των βαθμολογιών των χρηστών μπορεί για αυτό το λόγο να αποδειχτεί εξαιρετικά σημαντική. Μια κανονικοποιημένη βαθμολόγηση r_{ui} ενός χρήστη u για αντικείμενο i μπορεί να αναπαρασταθεί ως $h(r_{ui})$, όπου h μια συνάρτηση κανονικοποίησης. Ο κανόνας κανονικοποίησης παρουσιάζεται παρακάτω [1]:

$$\overline{r_{ui}} = h^{-1}\left(\Sigma_{v \in Ni(u)} (w_{uv} * h(r_{ui})) / \Sigma_{v \in Ni(u)} |w_{uv}|\right) \quad (2.5)$$

Παρατηρείστε το h^{-1} στην εξίσωση. Ο λόγος ύπαρξης του είναι ότι η βαθμολόγηση πρέπει να μετατραπεί ξανά σε κανονική κλίμακα μετά τον υπολογισμό της.

2.2.4 – Μέθοδοι βασισμένες στο αντικείμενο (Item-based)

Παρόμοια με τις μεθόδους γειτονιών βασισμένες στο χρήστη, οι μέθοδοι βασισμένες στο αντικείμενο κάνουν συστάσεις στους χρήστες, κάνοντας συγκρίσεις ανάμεσα σε αντικείμενα αντί για χρήστες.

Όπως στην εξίσωση 2.3 επιχειρούμε να κάνουμε πρόβλεψη με βάση το σύνολο των k κοντινότερων χρηστών προς τον χρήστη u, έτσι εδώ κάνουμε πρόβλεψη με

βάση το σύνολο των k κοντινότερων αντικειμένων προς το αντικείμενο i, για το οποίο επιθυμούμε να κάνουμε μια πρόβλεψη βαθμολόγησης του χρήστη u. Αντίστοιχα, οι τιμές (βάρη) ομοιότητας για το αντικείμενο i προς βαθμολόγηση και ένα αντικείμενο $\iota \in N_u(\iota)$, όπου $N_u(\iota)$ το σύνολο των κοντινότερων (γειτονικών) αντικειμένων προς το αντικείμενο i, σύνολο αντικειμένων το οποίο ο χρήστης u έχει ήδη βαθμολογήσει, ορίζονται ως $w_{i\iota}$. [1]

Αντικαθιστώντας στην εξίσωση 2.4 για αντικείμενα στην θέση των χρηστών, προκύπτει η εξής εξίσωση πρόβλεψης βαθμολόγησης:

$$\overline{r_{ui}} = h^{-1}\left(\Sigma_{\iota \in Nu(i)}\left(w_{i\iota} * h(r_{ui})\right) / \Sigma_{\iota \in Nu(i)} |w_{i\iota}|\right) \quad (2.6)$$

Φτάνοντας στο τέλος αυτής της ενότητας, θα θέλαμε να δείξουμε ένα παράδειγμα μιας μεθόδου πρόβλεψης γραμμένη σε Python. Στο παρακάτω απόσπασμα δίνονται δύο πίνακες, ratings και similarity, ίσου μεγέθους n*n, που αντικατοπτρίζουν τις βαθμολογήσεις των χρηστών για τα αντικείμενα και τις τιμές ομοιότητας (βάρη). [3] Οι υπολογισμοί μεταξύ των δύο γίνονται όπως ορίστηκαν στην 2.3.

```
import numpy as np

def predict_fast_simple(ratings, similarity,
                        kind = 'user'):
    #user case
    if kind == 'user':
        #dot product of similarities (weights)
        #multiplied by the ratings, divided by
        #the sum of the absolute values of the
        #cosine similarities (weights)
        return similarity.dot(ratings)/\
               np.array([np.abs(similarity)\
                         .sum(axis=1)]).T
    #item case
    elif kind == 'item':
        #the same but transposed
        return ratings.dot(similarity)/\
               np.array([np.abs(similarity)\
                         .sum(axis=1)])
```

Κώδικας 2 – Simple prediction function

2.2.5 – Σύγκριση μεθόδων

Επιλέγοντας ανάμεσα στις δύο παραπάνω μεθόδους πρέπει κανείς να αναλογιστεί τις επιπτώσεις της υλοποίησης του καθενός. Για παράδειγμα, σε ένα σύστημα συστάσεων όπου οι χρήστες υπερβαίνουν τα αντικείμενα πληθυσμιακά, η υλοποίηση ενός αλγορίθμου βασισμένου σε γειτονιές με βάση τους χρήστες μπορεί να προσφέρει υψηλότερη ακρίβεια σε σχέση με έναν αλγόριθμο βασισμένο σε αντικείμενα. Παρ΄ όλα αυτά, σε αυτή την περίπτωση ένας αλγόριθμος βασισμένος στα αντικείμενα μπορεί να είναι πιο αποδοτικός καθώς θα καταναλώσει λιγότερους πόρους. [1]

Θα πρέπει ακόμη κανείς να αναλογιστεί την σταθερότητα του συστήματος. Σε ένα περιβάλλον όπου οι χρήστες είναι γενικώς στατικοί, μια προσέγγιση βασισμένη στους χρήστες μπορεί να αποδειχτεί ασφαλέστερη. Σημαντική είναι ακόμη η αιτιολόγηση της σύστασης. Σε έναν αλγόριθμο βασισμένο σε αντικείμενα είναι απλούστερο να αιτιολογηθεί η πρόταση του αντικείμενου σε έναν χρήστη. Από την άλλη όμως ένας σύστημα βασισμένο στα αντικείμενα μπορεί να κάνει περισσότερες προτάσεις με βάση τα ήδη υπάρχοντα ενδιαφέροντα ενός χρήση. Ως αντίλογος, ένας αλγόριθμος βασισμένος στους χρήστες μπορεί να προτείνει σε έναν χρήστη αντικείμενα που δεν είχε ανακαλύψει ακόμη, τα οποία παρόμοιοι χρήστες αρέσκονται, και τα οποία μπορεί πιθανόν επίσης να του αρέσουν. [1]

Εν τέλει, χτίζοντας ένα σύστημα συστάσεων γειτονιών όλα τα παραπάνω πρέπει να ληφθούν υπόψη.

2.2.6 – Κανονικοποίηση (Normalization)

Η κανονικοποίηση, στην οποία αναφερθήκαμε και πρότινος, αποτελεί μια μέθοδο εξισορρόπησης των διαφορών στους τρόπους βαθμολόγησης για διαφορετικά αντικείμενα και χρήστες. Παρακάτω παρατίθενται οι δύο κυριότερες μέθοδοι κανονικοποίησης [1]:

Μέση τιμή

Η πιο κλασσική και συχνά χρησιμοποιούμενη τακτική κατά την διαδικασία της κανονικοποίησης είναι η χρήση της μέσης τιμής. Έστω, $\overline{r_u}$ η μέση βαθμολόγηση των κοντινότερων (γειτόνων) προς τον χρήστη u χρηστών. Θέτοντας $h(r_{ui}) = r_{ui} - \overline{r_u}$ στην εξίσωση 2.5, προκύπτει:

$$\overline{r_{ui}} = \overline{r_u} + (\Sigma_{v \in Ni(u)} (w_{uv} * (r_{ui} - \overline{r_u})) / \Sigma_{v \in Ni(u)} |w_{uv}|) \quad (2.7)$$

Αντίστοιχα, με μια προσέγγιση ως προς τα αντικείμενα, θέτουμε $h(r_{ui}) = r_{ui} - \overline{r_i}$, όπου $\overline{r_i}$ η μέση βαθμολογία των κοντινότερων (γειτόνων) προς το αντικείμενο i για το οποίο θέλουμε να προβλέψουμε την βαθμολόγηση του χρήση u. Αντικαθιστώντας αντίστοιχα στην 2.6, προκύπτει:

$$\overline{r_{ui}} = \overline{r_u} + (\Sigma_{v \in Nu(i)} (w_{it} * (r_{ui} - \overline{r_i})) / \Sigma_{v \in Nu(i)} |w_{it}|) \quad (2.8)$$

Z-Score

Κατά την τεχνική αυτή, λαμβάνεται υπόψη η διαφορετικότητα των βαθμολογήσεων. Έστω ένας χρήστης x, ο οποίος τείνει να βαθμολογεί τακτικά με 3 και ένας χρήστης y, του οποίου οι βαθμολογήσεις κυμαίνονται ομοιόμορφα ανάμεσα σε όλες τις πιθανές τιμές. Με βάση αυτή την παρατήρηση, μια βαθμολόγηση 3 από τον χρήστη y δεν έχει ιδιαίτερη σημασία, ενώ η ίδια βαθμολόγηση από τον χρήστη y μπορεί να περιέχει περισσότερες πληροφορίες για την ψυχολογία του χρήστη ή την ποιότητα του αντικειμένου.

Στην περίπτωση μεθόδων βασισμένων στον χρήστη λοιπόν, ορίζουμε $h(r_{ui}) = (r_{ui} - \overline{r_i}) / \sigma_u$, όπου σ_u η μέση απόκλιση των βαθμολογήσεων του χρήστη u.

Αντικαθιστώντας στην 2.4, προκύπτει:

$$\overline{r_{ui}} = \overline{r_u} + \sigma_u * (\Sigma_{v \in Ni(u)} (w_{uv} * (r_{ui} - \overline{r_u}) / \sigma_u) / \Sigma_{v \in Ni(u)} |w_{uv}|) \quad (2.9)$$

Αντίστοιχα για μεθόδους βασισμένες στο αντικείμενο:

$$\overline{r_{ui}} = \overline{r_i} + \sigma_i * (\Sigma_{v \in Nu(i)} (w_{it} * (r_{ui} - \overline{r_i}) / \sigma_i) / \Sigma_{v \in Nu(i)} |w_{it}|) \quad (2.10)$$

Παρακάτω παρουσιάζεται ένα παράδειγμα κώδικα μιας μεθόδου πρόβλεψης με τη χρήση της μέσης τιμής. [3] Να σημειωθεί ότι η μεταβλητή bias (κλίση/προτίμηση), η οποία εμφανίζεται τακτικά σε παρόμοια προβλήματα μπορεί να αναπαριστά ένα μεγάλο εύρος πιθανών τιμών. Σε αυτή την περίπτωση ορίζουμε το bias ως τη μέση τιμή.

```
import numpy as np

#prediction with bias
def predict_nobias(ratings, similarity, kind='user'):
    #user case
    if kind == 'user':
        #mean rating is bias
        user_bias = ratings.mean(axis=1)
        #calculate the ratings again based on bias
        ratings = (ratings - user_bias[:, np.newaxis].copy())
        #make the prediction, then add bias
        #(h()*h()^-1 golden rule)
        pred = similarity.dot(ratings) /
np.array([np.abs(similarity).sum(axis=1)]).T
        pred += user_bias[:, np.newaxis]
    #item case
    elif kind == 'item':
        #same as before
        item_bias = ratings.mean(axis=0)
        ratings = (ratings - item_bias[np.newaxis, :]).copy()
        pred = ratings.dot(similarity) /
np.array([np.abs(similarity).sum(axis=1)])
        pred += item_bias[np.newaxis, :]
    return pred
```

Κώδικας 3 – Prediction function with bias

2.2.7 – Μέτρα ομοιότητας

Στις παραπάνω ενότητες είδαμε παραδείγματα για το πως μπορούμε να προβλέψουμε τη βαθμολόγηση ενός χρήστη με βάση την τιμή ομοιότητας ανάμεσα σε αυτόν και κάποιον άλλο χρήστη ή ανάμεσα σε αντικείμενα που έχει βαθμολογήσει ο ίδιος και ένα αντικείμενο προς βαθμολόγηση. Αυτό που δεν δείξαμε όμως ακόμη, είναι το πως μπορούμε να υπολογίσουμε την τιμή της ομοιότητας ανάμεσα στους χρήστες και τα αντικείμενα. Παρακάτω παρατίθενται ορισμένα μέτρα ομοιότητας [1][2]:

Cosine Vector (CV)

Έστω ένα διάνυσμα x_u, το οποίο αναπαριστά τις βαθμολογήσεις ενός χρήστη για όλα τα αντικείμενα. Στην περίπτωση που ο χρήστης έχει βαθμολογήσει ένα

αντικείμενο i, ορίζεται $x_{ui} = r_{ui}$, όπου r_{ui} η βαθμολόγηση του χρήστη u για το αντικείμενο i.

Για να το αναπαραστήσουμε καλύτερα θα δώσουμε ένα πολύ μικρό παράδειγμα σε κώδικα για άλλη μια φορά. Θεωρείστε έναν χρήστη σε ένα σύστημα συστάσεων με μόνο 6 αντικείμενα. Έστω ότι αυτός ο χρήστης έχει δώσει βαθμολογήσεις για τα αντικείμενα 0, 1, 3 και 5, μετρώντας από το 0, με τρόπο που συχνά γίνεται στην επιστήμη των υπολογιστών. Αντίστοιχα, ένας δεύτερος χρήστης έχει βαθμολογήσει τα αντικείμενα 1 και 5 (για να είμαστε ελαφρώς πιο πραγματιστές, αφού σε ένα σύστημα συστάσεων εξαιρετικά σπάνια οι χρήστες βαθμολογούν έστω και το 5 τοις εκατό των αντικειμένων). Ας υποθέσουμε ότι οι χρήστες αυτοί είναι προς το παρόν οι μοναδικοί στο σύστημα συστάσεων που έχουμε στα χέρια μας. Στην περίπτωση αυτή, ο πίνακας βαθμολογήσεων του συστήματος συστάσεων μας, έστω ratings, θα είναι ένας πίνακας 2*6. Παρακάτω φαίνονται τα διανύσματα βαθμολόγησης αυτών των χρηστών καθώς και ο πίνακας ratings.

```
x1 = [1,3,0,2,0,4]
x2 = [0,3,0,0,0,2]

ratings = [x1, x2]
```

Ψευδοκώδικας 4 – Διανύσματα και πίνακας βαθμολογήσεων

Μελετώντας τα παραπάνω παραδείγματα κώδικα, μπορούμε σταδιακά να βγάλουμε περισσότερο νόημα για το τι ακριβώς συμβαίνει.

Ένας απλός τρόπος να υπολογίσουμε τις ομοιότητες μεταξύ των διανυσμάτων βαθμολογήσεων των χρηστών, είναι ο υπολογισμός του διανύσματος του συνημίτονου μεταξύ των διανυσμάτων βαθμολογήσεων. Στην περίπτωση ενός συστήματος συστάσεων βασισμένο σε χρήστες, ο υπολογισμός αυτός μπορεί να γίνει ως εξής:

$$CV(u,v) = cos(x_u,x_v) = (\Sigma_{i \in Iuv} r_{ui} * r_{vi}) / (\Sigma_{i \in Iu} r_{ui}^2 * \Sigma_{i \in Iv} r_{vi}^2) \quad (2.11)$$

Όπου I_{uv} το σύνολο των αντικείμενων που έχουν βαθμολογηθεί από κοινού από τους χρήστες u και v.

Αντίστοιχα για αντικείμενα:

$$CV(i,j) = cos(x_i,x_j) = (\Sigma_{u \in Uij} r_{ui} * r_{uj}) / (\Sigma_{u \in Ui} r_{ui}^2 * \Sigma_{v \in Yj} r_{vi}^2) \quad (2.12)$$

Pearson Correlation (PC)

Μια διαφορετική μετρική, η συσχέτιση Pearson (Pearson Correlation), συγκρίνει βαθμολογήσεις, αφαιρώντας κατά την διαδικασία την επίδραση της μέσης τιμής και της διακύμανσης:

$$PC(u,v) = (\Sigma_{i \in Iuv} (r_{ui} - \overline{r_u}) * (r_{vi} - \overline{r_u})) / (\Sigma_{i \in Iu} (r_{ui} - \overline{r_u})^2 * \Sigma_{i \in Iv} (r_{vi} - \overline{r_u})^2) \quad (2.13)$$

Η παραπάνω εξίσωση υπολογίζει για μια μέθοδο βασισμένη σε χρήστες. Εναλλακτικά, για μία μέθοδο βασισμένη σε αντικείμενα:

$$PC(i,\iota) = (\Sigma_{u \in Ui\iota} (r_{ui} - \overline{r_i}) * (r_{u\iota} - \overline{r_i})) / (\Sigma_{u \in Ui\iota} (r_{ui} - \overline{r_i})^2 * \Sigma_{u \in Ui\iota} (r_{v\iota} - \overline{r_i})^2) \quad (2.14)$$

Adjusted Cosine (AC)

Μια παραλλαγή της παραπάνω εξίσωσης, η οποία υπολογίζει ομοιότητες ανάμεσα σε αντικείμενα με τη χρήση όμως του μέσου όρου της βαθμολόγησης των χρηστών είναι η Adjusted Cosine:

$$AC(i,\iota) = (\Sigma_{u \in Ui\iota} (r_{ui} - \overline{r_u}) * (r_{u\iota} - \overline{r_u})) / (\Sigma_{u \in Ui\iota} (r_{ui} - \overline{r_u})^2 * \Sigma_{u \in Ui\iota} (r_{v\iota} - \overline{r_u})^2) \quad (2.15)$$

Άλλες μετρικές, που δεν θα αναφερθούν περαιτέρω, περιλαμβάνουν τις: Mean Squared Difference (MSD), Spearman Rank Correlation (SPC), Frequency-Weighted Pearson Correlation (FWPC) και Weighted Pearson Correlation (WPC). [1]

Στα παραδείγματα μας θα χρησιμοποιήσουμε μια απλή μέθοδο υπολογισμού του διανύσματος του συνημίτονου ανάμεσα στα διανύσματα των βαθμολογήσεων χρηστών και ανάμεσα στα διανύσματα βαθμολογιών των ταινιών για τους πίνακες ομοιοτήτων. Παρακάτω παρουσιάζονται δύο μέθοδοι υπολογισμού των ομοιοτήτων. Η αργή μέθοδος υλοποιεί αναλυτικά τους υπολογισμούς των εξισώσεων με τον τρόπο που τους παραθέσαμε για τον κάθε ένα χρήστη ή αντικείμενο. Από την άλλη, η γρήγορη μέθοδος κάνει τους υπολογισμούς γρηγορότερα, με πράξεις σε επίπεδο πινάκων, προσθέτοντας ένα πολύ μικρό ποσό για την αποφυγή σφαλμάτων κατά την διαίρεση με μηδενικά κελιά του πίνακα. Για λόγους υπολογιστικής ταχύτητας και καλαισθησίας κώδικα, θα επιλέξουμε την γρήγορη. [3]

```
import numpy as np
```

```
#cosine similarity methods

#slow method
def slow_similarity(ratings, kind='user'):
    #user case
    if kind == 'user':
        axmax = 0
        axmin = 1
    #item case
    elif kind == 'item':
        axmax = 1
        axmin = 0
        #make the similarity matrix
        #(item*item or user*user depending on the case)
    sim = np.zeros((ratings.shape[axmax],
                    ratings.shape[axmax]))
    #for every user or item
    for u in range(ratings.shape[axmax]):
        #for every user or item (again)
        for uprime in range(ratings.shape[axmax]):
            #sum factors
            rui_sqrd = 0.
            ruprimei_sqrd = 0.
            #for each column or row (depending on kind)
            for i in range(ratings.shape[axmin]):
                #numerator
                sim[u, uprime] = ratings[u, i] * ratings[uprime,
i]
                #denominator root sums
                rui_sqrd += ratings[u,i] ** 2
                ruprimei_sqrd += ratings[uprime, i] ** 2
            #fill the similarity matrix for user or item
            sim[u, uprime] /= np.sqrt(rui_sqrd*ruprimei_sqrd)
    return sim

#fast method
#epsilon: very small number for handling divided-by-zero errors
def fast_similarity(ratings, kind='user', epsilon=1e-9):
    #user case
    if kind == 'user':
        sim = ratings.dot(ratings.T) + epsilon
    #item case (transposed case scenario)
```

```
    elif kind == 'item':
        sim = ratings.T.dot(ratings) + epsilon
    #make the norms
    norms = np.array([np.sqrt(np.diagonal(sim))])
    return (sim/norms/norms.T)
```

Κώδικας 5 – Μέθοδοι υπολογισμού πινάκων ομοιότητας

2.2.8 – Προφιλτράρισμα (Prefiltering)

Σε μεγάλα συστήματα συστάσεων, τα αντικείμενα και οι χρήστες κυμαίνονται στο ποσό των εκατομμυρίων. Για το λόγο αυτό, είναι συχνά αδύνατο να υπολογίσουμε την πρόβλεψη ενός χρήστη με βάση όλους τους υπόλοιπους χρήστες του συστήματος ή αντικείμενα που έχει βαθμολογήσει ο χρήστης στο σύστημα. Η λύση είναι απλή και μάλιστα την παρουσιάσαμε ήδη έμμεσα στις παραπάνω ενότητες. Επιλέγουμε έναν αριθμό κοντινότερων γειτόνων. [1][2]

Θα επιλέξουμε να παρουσιάσουμε την απλούστερη τακτική prefiltering, η οποία είναι η επιλογή ενός σταθερού αριθμού κοντινότερων γειτόνων (Top-k), με ένα παράδειγμα, συνδυάζοντας ταυτόχρονα ό,τι μάθαμε από την ενότητα της κανονικοποίησης. Παρακάτω παρατίθεται μια μέθοδος πρόβλεψης βαθμολόγησης, η οποία αρχικά επιλέγει τους k κοντινότερους γείτονες και έπειτα προβλέπει την βαθμολόγηση με τη χρήση της μεθόδου mean centering, με τον ίδιο τρόπο που παρουσιάσαμε στην ενότητα της κανονικοποίησης. [3]

```
import numpy as np

#top-k prediction with bias
def predict_topk_nobias(ratings, similarity, kind='user', k=40):
    #make the prediction matrix
    pred = np.zeros(ratings.shape)
    #user case
    if kind == 'user':
        #mean bias
        user_bias = ratings.mean(axis=1)
        #remove bias
        ratings = (ratings - user_bias[:, np.newaxis]).copy()
        #for each user
        for i in range(ratings.shape[0]):
            #get the top-k users
            top_k_users = [np.argsort(similarity[:,i])[:-k-1:-1]]
            for j in range(ratings.shape[1]):
```

```
                #make the prediction for top-k neighbours
                pred[i, j] = similarity[i,
:][top_k_users].dot(ratings[:, j][top_k_users])
                pred[i, j] /= np.sum(np.abs(similarity[i,
:][top_k_users]))
        #add the bias
        #(h()*h()^-1 golden rule)
        pred += user_bias[:, np.newaxis]
        #item case
    if kind == 'item':
        #same as before
        item_bias = ratings.mean(axis=0)
        ratings = (ratings - item_bias[np.newaxis, :]).copy()
        for j in range(ratings.shape[1]):
            top_k_items = [np.argsort(similarity[:,j])[:-k-1:-1]]
            for i in range(ratings.shape[0]):
                pred[i, j] = similarity[j,
:][top_k_items].dot(ratings[i, :][top_k_items].T)
                pred[i, j] /= np.sum(np.abs(similarity[j,
:][top_k_items]))
        pred += item_bias[np.newaxis, :]
    return pred
```

Κώδικας 6 – Μέθοδος υπολογισμού πρόβλεψης για k κοντινότερους γείτονες με τη χρήση mean centering

2.2.9 – Ένα παράδειγμα αλγορίθμου γειτονιάς σε Python

Παραθέτουμε ένα περιγραφικό παράδειγμα υλοποίησης ενός αλγορίθμου γειτονιάς, αποσυναρμολογώντας την διαδικασία σε κομμάτια, χρησιμοποιώντας ταυτόχρονα ότι μάθαμε παραπάνω. [3]

Part 1: Import libraries and functions

Ξεκινούμε από την εισαγωγή των κατάλληλων βιβλιοθηκών και μεθόδων.

Θα χρειαστούμε τις εξής βιβλιοθήκες:

- numpy για τους υπολογισμούς μεταξύ πινάκων [4]
- pandas για την φόρτωση και διαχωρισμό των δεδομένων (pandas dataframes) [5]
- pathlib για την αναζήτηση του dataset μας μέσα στο σύστημα αρχείων
- time για την χρονομέτρηση της διαδικασίας πρόβλεψης

- sklearn (scikit-learn) για την χρήση της μεθόδου εύρεσης του μέσου τετραγωνικού σφάλματος
- matplotlib και seaborn για τον σχεδιασμό των γραφημάτων

Θα χρησιμοποιήσουμε τις βιβλιοθήκες pandas, pathlib, time, matplotlib και seaborn στο κυρίως πρόγραμμα μας, ενώ όσον αφορά τις βιβλιοθήκες numpy και scikit-learn θα τις χρησιμοποιήσουμε για την δημιουργία των μεθόδων τις οποίες εισάγουμε από ένα script με όνομα helper.py:

- Εισάγουμε μια νέα μέθοδο, την train_test_split, η οποία θα διαχωρίσει το dataset μας σε training και test set.
- Η μέθοδος fast_similarity που δείξαμε παραπάνω θα χρησιμοποιηθεί για τον υπολογισμό του πίνακα ομοιοτήτων.
- Η predict_topk_nobias θα χρησιμοποιηθεί για τον υπολογισμό του πίνακα προβλέψεων.
- Εισάγουμε ακόμη μία νέα μέθοδο, με όνομα get_mse, η οποία θα υπολογίσει την απόκλιση ανάμεσα στον πίνακα πρόβλεψης και το test set μας με τη χρήση του υπολογισμού του μέσου τετραγωνικού σφάλματος.

```
#neighbourhood-based algorithm

#part 1: importing

#libraries
#import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
import time #time
#from sklearn.metrics import mean_squared_error #mse
import matplotlib.pyplot as plt #plotting
import seaborn as sns #data visualization based on matplotlib

#methods
from helper import train_test_split, fast_similarity,
predict_topk_nobias, get_mse
```

Κώδικας 7 – Μέθοδοι γειτονιών, Part 1: εισαγωγή βιβλιοθηκών και μεθόδων

```
#split dataset to training and test set
def train_test_split(ratings):
    test = np.zeros(ratings.shape)
```

```
    train = ratings.copy()
    #for each user
    for user in range(ratings.shape[0]):
        #select 10 random ratings of a user for an item,
        #without replacement
        test_ratings = np.random.choice(ratings[user,
:].nonzero()[0],
                                        size=10,
                                        replace=False)
        #exclude those ratings from the training set
        train[user, test_ratings] = 0.
        #include them in the test set
        test[user, test_ratings] = ratings[user, test_ratings]
    #assert (test) if the training and test sets contain
    #common ratings (they should not)
    assert(np.all((train*test == 0)))
    return train, test #return the test and training set arrays
```

Κώδικας 8 – Μέθοδοι γειτονιών, μέθοδος διαχωρισμού dataset σε training και test set

```
#mean squared error
def get_mse(pred, actual):
    #prepare the prediction table for the given ratings
    pred = pred[actual.nonzero()].flatten()
    #prepare the evaluation table for the given ratings
    actual = actual[actual.nonzero()].flatten()
    #calculate the mean squared error between the two
    return mean_squared_error(pred, actual)
```

Κώδικας 9 – Μέθοδοι γειτονιών, μέθοδος υπολογισμού μέσου τετραγωνικού σφάλματος

Part 2: Prepare the data

Έχοντας φτιάξει τις μεθόδους μας, η διαδικασία προετοιμασίας των δεδομένων μοιάζει εξαιρετικά εκτενής σε σχέση με τα υπόλοιπα κομμάτια της υλοποίησης.

Το πρώτο πράγμα που θα χρειαστούμε είναι ένα dataset. Τα datasets στα συστήματα συστάσεων αποτελούν εξαιρετικά μεγάλα αρχεία, τα οποία αναπαριστούν βαθμολογήσεις χρηστών για αντικείμενα. Ένα dataset μπορεί να περιέχει εκατοντάδες χιλιάδες γραμμές δεδομένων έως και αρκετά εκατομμύρια. Λόγω της φύσης και του μεγέθους τους, ένα ρεαλιστικό dataset, βασισμένο σε πραγματικούς χρήστες και αντικείμενα, μπορεί να χρειαστεί χρόνια για να συλλεχτούν τα απαραίτητα δεδομένα για την δημιουργία του. Για αυτό το λόγο είναι

συχνά δύσκολο να συντάξουμε μόνοι μας ένα dataset. Αντιθέτως, προτιμούμε ένα έτοιμο dataset, από αυτά που είναι συχνά διαθέσιμα στο διαδίκτυο ελεύθερα.

Σε αυτή την περίπτωση, επιλέξαμε το παλιό MovieLens dataset των 100 χιλιάδων βαθμολογήσεων (ml-100k). Τα MovieLens datasets είναι datasets δημιουργημένα από αληθινούς χρήστες και αντικείμενα, αναπαριστούν βαθμολογήσεις χρηστών για ταινίες και είναι ελεύθερα για το ευρύ κοινό, με το μέγεθος τους να φτάνει έως και τα 20 εκατομμύρια ratings. [6]

Από όλα τα αρχεία του dataset, θα χρησιμοποιήσουμε το αρχείο u.data, του οποίου η κάθε γραμμή περιέχει τη βαθμολογία ενός χρήστη για ένα αντικείμενο, μαζί με ένα timestamp που δηλώνει την ημερομηνία σε second. Το κάθε πεδίο διαφοροποιείται από ένα άλλο, με έναν χαρακτήρα Tab.

```
196   242   3     881250949
186   302   3     891717742
22    377   1     878887116
244   51    2     880606923
166   346   1     886397596
...
```

Κώδικας 10 – Η δομή του αρχείου u.data του dataset ml-100k

Έπειτα, θα υπολογίσουμε τον αριθμό των χρηστών και αντικειμένων για την δημιουργία του πίνακα ratings και θα βρούμε το ποσοστό βαθμολόγησης αντικειμένων από χρήστες (sparsity). Στην συνέχεια διαχωρίζουμε το dataset σε training και test sets.

```
#part 2: preparing data

PATH = Path('..\Datasets\ml-100k')

#file does not contain headers
#we append them ourselves
names = ['user_id', 'item_id', 'rating', 'timestamp']
#data seperated by tabs, set headers to the names list
df = pd.read_csv(PATH/'u.data', sep='\t', names=names)
print(df.head()) #get the first 5

#number of unique users, items
n_users = df.user_id.unique().shape[0]
n_items = df.item_id.unique().shape[0]
```

```
print(f'\n{n_users} users')
print(f'{n_items} items')

#make the ratings matrix
ratings = np.zeros((n_users, n_items))
#we map user/item IDs to user/item indices by removing the
#"Python starts at 0" offset between them
for row in df.itertuples():
    ratings[row[1]-1, row[2]-1] = row[3]
print(f"\n {ratings}")

#sparsity
#(get the non-zero ratings)
sparsity = float(len(ratings.nonzero()[0]))
sparsity /= (ratings.shape[0]*ratings.shape[1])
#percentage
sparsity *= 100
print(f'\nRatings sparsity: {sparsity}%')

#make the training and test sets
train, test = train_test_split(ratings)
```

Κώδικας 11 – Μέθοδοι γειτονιών, Part 2: προετοιμασία δεδομένων

Part 3: Make the similarities table

Για την υπολογισμό των ομοιοτήτων θα χρονομετρήσουμε την διάρκεια υπολογισμού με τη χρήση της βιβλιοθήκης time.

```
#part 3: make the similarities table

start = time.time()
user_similarity = fast_similarity(train)
item_similarity = fast_similarity(train, kind = 'item')
end = time.time()
print(f'\nTime elapsed for measuring cosine \
similarity with fast method: {end-start}')

#print the first four in the item similarity table
print (f'\nitem_similarity[:4, :4]: {item_similarity[:4, :4]}')
```

Κώδικας 12 – Μέθοδοι γειτονιών, Part 3: υπολογισμός πίνακα ομοιοτήτων

Part 4: Make the predictions table

Παρομοίως για τον υπολογισμό των προβλέψεων.

```
#part 4: make the predictions table
```

```
start = time.time()
item_prediction = predict_topk_nobias(train, item_similarity,
kind='item')
user_prediction = predict_topk_nobias(train, user_similarity)
end = time.time()
print(f'\nTime elapsed for predicting \
similarity with Top-k method with bias: {end-start}')
```

Κώδικας 13 – Μέθοδοι γειτονιών, Part 4: υπολογισμός πίνακα προβλέψεων

Part 5: Evaluate the predictions

Ο υπολογισμός της συνάρτησης απώλειας μεταξύ training και test set είναι εξαιρετικά απλός.

```
#part 5: testing

print(f'\nUser-based CF MSE: {get_mse(user_prediction, test)}')
print(f'Item-based CF MSE: {get_mse(item_prediction, test)}')
```

Κώδικας 14 – Μέθοδοι γειτονιών, Part 5: αξιολόγηση

Εύρεση βέλτιστου αριθμού γειτόνων

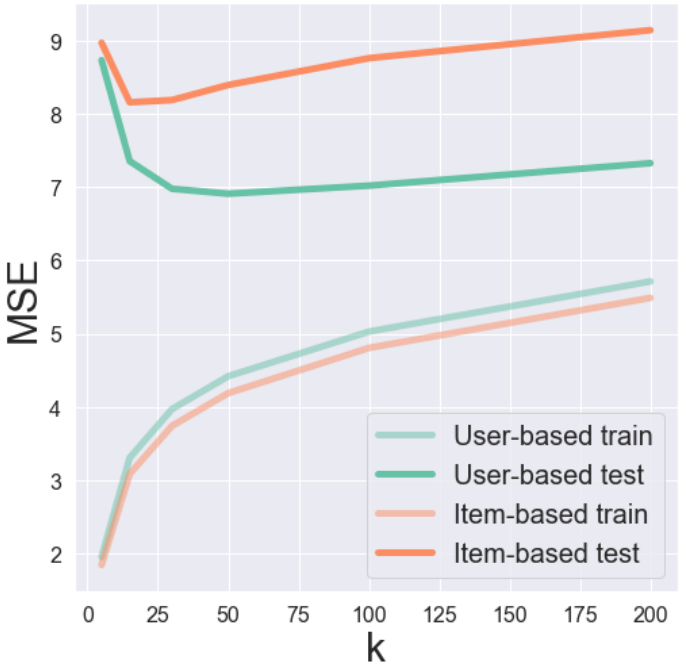
Συνοψίζοντας όσον αφορά τις μεθόδους γειτονιών, θα παρουσιάσουμε κάτι τελευταίο. Μιλήσαμε παραπάνω για το πως οι βιβλιοθήκες matplotlib και seaborn θα χρησιμοποιηθούν για την αναπαράσταση γραφημάτων. [7][8]

Έστω ότι θέλουμε να ανακαλύψουμε ποιος αριθμός χρηστών είναι ο βέλτιστος για το σύστημα συστάσεων μας. Παραθέτουμε ένα παράδειγμα κώδικα για το πως μπορούμε να κάνουμε κάτι τέτοιο, καθώς και τα αποτελέσματα του κώδικα αυτού.

```
#let's try a different amount of top-k neighbors
#test (really slow). 50 seems to be optimal amount of top-k
neighbors

#list of possible top-k neighbors
k_array = [5, 15, 30, 50, 100, 200]
user_train_mse = []
user_test_mse = []
item_test_mse = []
item_train_mse = []

#for each possible amount of neighbors
for k in k_array:
    #top k predictions
```

```
    user_pred = predict_topk_nobias(train, user_similarity,
kind='user', k=k)
    item_pred = predict_topk_nobias(train, item_similarity,
kind='item', k=k)

    #sum the mse for each:
    #user
    user_train_mse += [get_mse(user_pred, train)]
    user_test_mse += [get_mse(user_pred, test)]
    #item
    item_train_mse += [get_mse(item_pred, train)]
    item_test_mse += [get_mse(item_pred, test)]

#make a seaborn set
sns.set()

#pick a color palette to draw from (Set2 palette, 2 colors
#selected)
pal = sns.color_palette("Set2", 2)

plt.figure(figsize=(8,8)) #8x8 figure
#draw the following
#alpha (intensity) = 0.5
plt.plot(k_array, user_train_mse, c=pal[0], label='User-based
train', alpha=0.5, linewidth=5)
plt.plot(k_array, user_test_mse, c=pal[0], label='User-based
test', linewidth=5)
plt.plot(k_array, item_train_mse, c=pal[1], label='Item-based
train', alpha=0.5, linewidth=5)
plt.plot(k_array, item_test_mse, c=pal[1], label='Item-based
test', linewidth=5)
#draw the legend at the best location, fonts: 20px
plt.legend(loc='best', fontsize=20)
#make the ticks as following
plt.xticks(fontsize=16);
plt.yticks(fontsize=16);
#and the labels
plt.xlabel('k', fontsize=30);
plt.ylabel('MSE', fontsize=30);
plt.show()
```

Κώδικας 15 – Μέθοδοι γειτονιών, εύρεση βέλτιστου αριθμού γειτόνων



Εικόνα 2 – Αποτέλεσμα Κώδικα 15

2.3 – Μέθοδοι βασισμένες σε μάθηση (Learning based)

Οι μέθοδοι βασισμένες σε μάθηση εκπαιδεύουν ένα παραμετρικό μοντέλο που περιγράφει την αλληλεπίδραση μεταξύ χρηστών και αντικειμένων σε μια διάρκεια χρόνου, με την χρήση εποχών και διόρθωσης των παραμέτρων του μοντέλου.

2.3.1 – Παράγοντας μέσης πρόβλεψης και πόλωση (bias)

Παρόμοια με τις μεθόδους γειτονιών, θα χρησιμοποιήσουμε και πάλι το γνωστό σύνολο αντικειμένων για τις εξισώσεις μας. υ, u για τους χρήστες, ι και i για τα αντικείμενα, r για της βαθμολογήσεις και βαθμολογίες και θα εισάγουμε ακόμη ένα νέο είδος μεταβλητής, την t, η οποία συμβολίζει την χρονική στιγμή της βαθμολόγησης. Εισάγουμε ακόμη δύο είδη μεταβλητών, την μέση τιμή βαθμολόγησης μ και την πόλωση (bias) b. [1]

Η πόλωση αναπαριστά την απόκλιση από την μέση πρόβλεψη για ένα αντικείμενο ή χρήστη. [1]

Ας πάρουμε το παράδειγμα του Πίνακα 1.

Έστω ο μέσος όρος βαθμολόγησης, μ = 3.07, και η μέση βαθμολόγηση του Χρήστου (5 +3 + 2) /3 = 3.33, όπως προκύπτει από τον Πίνακα 1. Έστω ακόμη η μέση τιμή βαθμολογίας της ταινίας Fight Club (4 + 2 + 1)/3 = 2.33 όπως προκύπτει και πάλι από τον πίνακα. Θέτουμε ως την απόκλιση από την μέση τιμή βαθμολόγησης του Χρήστου b_X = 3.33 – μ = 0.26. Ενώ την απόκλιση από την μέση τιμή βαθμολογίας για την ταινία Fight Club αντίστοιχα $b_{Fight\ Club}$ = 2.33 – μ = - 0.74. Με βάση τα παραπάνω, ο παράγοντας μέσης πρόβλεψης βαθμολόγησης του Χρήστου για την ταινία Fight Club είναι $\mu + b_X + b_{Fight\ Club}$ = 3.07 + 0.26 – 0.74 = 2.59. Παρακάτω φαίνεται η εξίσωση του υπολογισμού της μέσης πρόβλεψης βαθμολόγησης ενός χρήστη για ένα αντικείμενο.

$$b_{ui} = \mu + b_u + b_i \tag{2.16}$$

2.3.2 – Παραγοντοποίηση

SVD

Τα μοντέλα παραγοντοποίησης πινάκων χαρτογραφούν χρήστες και αντικείμενα σε πίνακες n παραγόντων, έτσι ώστε οι αλληλεπιδράσεις μεταξύ των δύο να σχηματίζονται ως το εσωτερικό γινόμενο μεταξύ των γραμμών των πινάκων. Με αυτό τον τρόπο, μπορούμε να αναδημιουργήσουμε τον αρχικό πίνακα των βαθμολογήσεων κάθε χρήστη για κάθε αντικείμενο, πολλαπλασιάζοντας τον πίνακα παραγόντων του χρήστη επί τον πίνακα παραγόντων του αντικειμένου.

Κάθε χρήστης αντιστοιχίζεται με ένα διάνυσμα παραγόντων p_u διάστασης n, ενώ ένα αντικείμενο αντιστοιχίζεται με ένα διάνυσμα q_i διάστασης n αντίστοιχα.[1]

Ας εξετάσουμε και πάλι τον Πίνακα 1 και ας χρησιμοποιήσουμε σαν παράδειγμα την βαθμολόγηση του Γιάννη για την ταινία Stardust, η οποία είναι 2. Έστω πως σε ένα σύστημα συστάσεων παραγοντοποίησης, όπως αυτά που εξετάζουμε σε αυτή την υποενότητα, ο αριθμός των παραγόντων είναι n = 5. Έστω τα παρακάτω διανύσματα παραγόντων, για τον χρήστη Γιάννη, p_giannis, και την ταινία Stardust, q_stardust. Πολλαπλασιάζοντας τους παράγοντες του Γιάννη με τους παράγοντες της ταινίας και αθροίζοντας τα γινόμενα, μπορούμε να πάρουμε την αρχική βαθμολόγηση του Γιάννη, 2.

```
p_giannis = [0.333, 1.2, 0.8, 0.7, 0.8]
q_stardust = [0.9, 0.5, 0.25, 0.71, 0.5]
```

```
r_giannis_stardust = p_giannis.dot(q_stardust).sum()
```

Ψευδοκώδικας 16 – Μέθοδοι παραγοντοποίησης, διανύσματα παραγόντων και υπολογισμός βαθμολόγησης

Καθένας από τους παράγοντες μπορεί να μετρά ένα εμφανές χαρακτηριστικό ή ένα χαρακτηριστικό το οποίο μπορεί να μην είναι εύκολα ανιχνεύσιμο από έναν απλό παρατηρητή. Για την περίπτωση του Γιάννη και της ταινίας Stardust, οι πρώτοι παράγοντες, 0.9 και 0.333, μπορεί να μετρούν κατά πόσο η ταινία ανήκει στο είδος των ρομαντικών ταινιών και κατά πόσο αρέσουν στον Γιάννη τέτοιου είδους ταινίες. Ενώ οι τρίτοι παράγοντες, 0.25 και 0.8, θα μπορούσαν να μετρούν το κατά πόσο η ταινία θυμίζει νοσταλγικές ταινίες των αρχών της δεκαετίας του 2000 και κατά πόσο αντίστοιχα αρέσουν στο Γιάννη τέτοιες ταινίες, χαρακτηριστικό το οποίο ίσως να μην είναι διαισθητικά αντιληπτό από τον μέσο παρατηρητή.

Λαμβάνοντας υπόψη στους υπολογισμούς μας τις πολώσεις για τον χρήστη και το αντικείμενο, καθώς και την μέση τιμή βαθμολόγησης, μπορούμε να υπολογίσουμε την πρόβλεψη βαθμολόγησης $\overline{r_{ui}}$ ενός χρήστη *u* για ένα αντικείμενο i ως εξής [1]:

$$\overline{r_{ui}} = \mu + b_u + b_i + q_i^T p_u \quad (2.17)$$

Στόχος στα συστήματα παραγοντοποίησης είναι η εύρεση κατάλληλων πολώσεων και παραγόντων που ελαχιστοποιούν την απώλεια μεταξύ ενός test set και μιας προβλεπόμενης τιμής. Το σύστημα εκπαιδεύει ένα μοντέλο παραγόντων και πολώσεων μέσω μιας διαδικασίας μάθησης, με βάση ένα training set. Σε κάθε εποχή οι παράγοντες και οι πολώσεις επανυπολογίζονται με τρόπο, έτσι ώστε η πρόβλεψη να προσεγγίζει όσο το δυνατόν καλύτερα το υπάρχον training set. Περισσότερες λεπτομέρειες για την διαδικασία της μάθησης δίνονται κατά το επόμενο κεφάλαιο, όπου και θα αναπτυχθούμε στο πως ένα σύστημα παραγοντοποίησης SVD, όπως παρουσιάζεται εδώ, μπορεί να υλοποιηθεί με την χρήση της βιβλιοθήκης PyTorch.

SVD++

Παρόμοια με το SVD, το σύστημα παραγοντοποίησης SVD++ στοχεύει στην πρόβλεψη βαθμολόγησης ενός χρήστη για ένα αντικείμενο με βάση ένα σύστημα παραγόντων και πολώσεων, εισάγοντας ταυτόχρονα στους υπολογισμούς,

παράγοντες έμμεσης ανατροφοδότησης που προκύπτουν από τις έμμεσα υποδηλούμενες προτιμήσεις των χρηστών. [1][2]

Έστω $R(u)$ το σύνολο των αντικειμένων που ο χρήστης u έχει επιλέξει να νοικιάσει και y_ι ένα διάνυσμα παραγόντων έμμεσης ανατροφοδότησης (ενοικίαση) για ένα αντικείμενο $\iota \in R(u)$. Τότε, η πρόβλεψη βαθμολόγησης ενός χρήστη u για ένα οποιοδήποτε μη βαθμολογημένο αντικείμενο i, προκύπτει ως εξής [1]:

$$\overline{r_{ui}} = \mu + b_u + b_i + q_i^T \left(p_u + |R(u)|^{-1/2} \Sigma_{\iota \in R(u)} y_\iota \right) \quad (2.18)$$

Όπου y_ι διάνυσμα παραγόντων έμμεσης ανατροφοδότησης, κεντραρισμένο γύρω από το 0, και $|R(u)|^{-1/2}$ το σύνολο των βαθμολογημένων από τον χρήστη αντικειμένων εις στην -1/2 για σκοπούς κανονικοποίησης.

Σε ένα σύστημα SVD++ είναι δυνατή ακόμη η ύπαρξη περισσοτέρων από ενός είδους παράγοντες έμμεσης ανατροφοδότησης. Για παράδειγμα, ενώ η εξίσωση 2.18 μπορεί να αναπαριστά την πρόβλεψη ενός χρήστη για μια ταινία, με y_ι το εάν ο χρήστης έχει νοικιάσει μια από τις ταινίες (αντικείμενα) που παρακολούθησε, μια άλλη εξίσωση μπορεί να λαμβάνει υπόψη όχι μόνο την ενοικίαση, αλλά και την αγορά, όπου $R^1(u)$ και $R^2(u)$ τα σύνολα των ταινιών που έχει νοικιάσει και αγοράσει ο χρήστης u αντίστοιχα [1].

$$\overline{r_{ui}} = \mu + b_u + b_i + q_i^T \left(p_u + |R^1(u)|^{-1/2} \Sigma_{\iota \in R1(u)} y_\iota + |R^2(u)|^{-1/2} \Sigma_{\iota \in R2(u)} y_\iota \right) \quad (2.19)$$

2.4 Άλλες μέθοδοι

Παρά το γεγονός ότι δεν πρόκειται να αναπτυχθούμε περαιτέρω με τον ίδιο τρόπο, όπως στις παραπάνω ενότητες, για λοιπές μεθόδους, αξίζει να αναφέρουμε μερικές από αυτές και να εξηγήσουμε περιληπτικά τον τρόπο λειτουργίας τους.

ALS (Alternating Least Squares)

Μια παραλλαγή του SVD, ο ALS, προσεγγίζει την διαδικασία μάθησης με έναν ελαφρώς διαφορετικό τρόπο. Σε αντίθεση με τον αλγόριθμο SVD, κατά τον ALS, χωρίζουμε την διαδικασία της μάθησης σε δύο ξεχωριστά στάδια. [1][2]

- Κατά το πρώτο στάδιο θέτουμε σταθερό το διάνυσμα παραγόντων των αντικειμένων (q_i) και λύνουμε ως προς το διάνυσμα παραγόντων των χρηστών (p_u).

- Κατά το δεύτερο θέτουμε σταθερό το διάνυσμα παραγόντων των χρηστών (p_u) και λύνουμε ως προς το διάνυσμα παραγόντων των αντικειμένων (q_i) αντίστοιχα.

Χρόνο-συνειδησιακό μοντέλο (Time-aware factor model)

Μια άλλη προσέγγιση παραγοντοποίησης, προσθέτει στον SVD++ την έννοια των παροδικών αλλαγών στους τρόπους βαθμολογίας των αντικειμένων και βαθμολόγησης από τους χρήστες. Η ιδέα βασίζεται σε μια παραλλαγή των πολώσεων, όπου οι πολώσεις υπολογίζονται και πάλι μέσω παρελθοντικών βαθμολογήσεων των χρηστών και βαθμολογιών των ταινιών, αλλά με την εισαγωγή ενός παράγοντος παλαιότητας, όπου παλαιότερες βαθμολογήσεις και βαθμολογίες παίζουν μικρότερο ρόλο στον υπολογισμό των πολώσεων. [1]

Η λογική πίσω από το χρόνο-συνειδησιακό μοντέλο είναι πως αναλόγως των εποχών, των χρονικών τάσεων και των αλλαγών στις προσωπικότητες των χρηστών, μια ταινία μπορεί να βαθμολογηθεί διαφορετικά από έναν χρήστη, για λόγους αλλαγής ψυχολογίας αυτού του χρήστη αλλά και των διαφορετικών προτύπων και αντιλήψεων της εκάστοτε εποχής.

Για να χρησιμοποιήσουμε και πάλι το παράδειγμα του Πίνακα 1, ο Γιάννης το 2017 βαθμολογεί την ταινία Stardust με βαθμολογία 2. Ο Γιάννης το 2019 όμως, έχει αλλάξει ως άνθρωπος και έχει ωριμάσει. Περισσότερο συναισθηματικός και πρόθυμος να αποδεχτεί καινούρια πράγματα, αποφασίζει να δει και πάλι την ταινία Stardust, αλλάζοντας αυτή τη φορά την βαθμολόγηση του σε 4. Ταυτόχρονα, το 2019, σε σχέση με το 2017, η ταινία Stardust θεωρείται πλέον μια κλασσική ρομαντική ταινία, πράγμα που μπορεί επιπλέον να επηρεάσει την άποψη του Γιάννη θετικά για αυτή.

Μέθοδοι γειτονιάς βασισμένες σε μάθηση

Υπάρχουν αρκετοί τρόποι που μία μέθοδος γειτονιάς μπορεί να υλοποιηθεί με την χρήση μάθησης.

Απλές

Ο πρώτος και απλούστερος τρόπος είναι ο υπολογισμός των πολώσεων χρηστών και αντικειμένων και του πίνακα ομοιοτήτων μέσω μιας διαδικασίας μάθησης, αντί της χρήσης μέτρων ομοιότητας, όπως είναι για παράδειγμα το διάνυσμα συνημίτονου, και τεχνικών, όπως είναι το mean-centering ή το z-score.

Μια τέτοια προσέγγιση μπορεί όμως να αποβεί δαπανηρή, καθώς ο υπολογισμός ενός μεγάλου πίνακα, όπως ο πίνακας ομοιοτήτων των μεθόδων γειτονιών, με την χρήση μάθησης, αποτελεί μια εξαιρετικά χρονοβόρα διαδικασία. [1]

Παραγοντοποίηση πίνακα ομοιοτήτων

Μία τεχνική που μπορεί να βοηθήσει στην επίλυση του παραπάνω προβλήματος, είναι η παραγοντοποίηση του πίνακα ομοιοτήτων. Αξιοποιώντας τα πλεονεκτήματα της παραγοντοποίησης, ένας πίνακας λίγων εκατομμυρίων στοιχείων, μπορεί εύκολα να αναπαρασταθεί από δύο μικρότερους πίνακες μερικών εκατοντάδων χιλιάδων στοιχείων, γλυτώνοντας πόρους αλλά και χρόνο κατά τους υπολογισμούς. [1]

Μεικτές

Μια διαφορετική τεχνική, η οποία στοχεύει στην μεγιστοποίηση της ακρίβειας, είναι η υλοποίηση ενός μεικτού συστήματος που συγχωνεύει μεθόδους παραγοντοποίησης και γειτονιών. Σε αυτή την περίπτωση η πρόβλεψη μιας βαθμολόγησης ενός χρήστη για ένα αντικείμενο γίνεται μεικτά, λαμβάνοντας μέρος στον υπολογισμό μια μέθοδος γειτονιάς, με έναν πίνακα ομοιοτήτων, και μια μέθοδος παραγοντοποίησης όπως είναι για παράδειγμα ο αλγόριθμος SVD. [2][9]

Φυσικά, σε όλες αυτές τις μεθόδους, μπορούν να ληφθούν ακόμη υπόψη παράγοντες, όπως είναι οι παρωδικές μεταβολές στην βαθμολόγηση και την βαθμολογία (χρόνο-συνειδησιακά μοντέλα) ή ακόμη και την τοποθεσία, μέθοδοι οι οποίες δεν εξετάζονται σε αυτή την εργασία.

SLIM (Sparse Linear Method)

Τέλος, δεν θα μπορούσαμε βέβαια να κλείσουμε αυτή την ενότητα χωρίς να μιλήσουμε για τον αλγόριθμο SLIM. Κατά τον αλγόριθμο αυτό, μια προβλεπόμενη βαθμολόγηση ενός χρήστη u υπολογίζεται ως ο πολλαπλασιασμός μεταξύ δύο αραιών διανυσμάτων πινάκων, r_u και w_i, όπου το πρώτο αντιστοιχεί σε ένα διάνυσμα αντικειμένων τα οποία έχει βαθμολογήσει ο χρήστης u ενώ το δεύτερο αντιστοιχεί σε ένα διάνυσμα ομοιοτήτων μεταξύ αντικειμένων που έχει βαθμολογήσει ο χρήστης και του αντικειμένου i προς πρόβλεψη. [10]

2.5 – Επίλογος

Σε αυτό το κεφάλαιο παρουσιάστηκαν οι μέθοδοι συνεργατικού φιλτραρίσματος (Collaborative Filtering) και αναπτύχθηκαν οι μέθοδοι γειτονιών (Neighborhood-based) και οι μέθοδοι βασισμένες σε μάθηση (Learning-based).

Κάθε είδος με τα πλεονεκτήματα του, οι μέθοδοι γειτονιών αποτελούν έναν απλό και εύκολο τρόπο υλοποίησης ενός συστήματος συστάσεων και παρέχουν μια ανθεκτική, επεξηγηματική και διαισθητικά αντιληπτή υλοποίηση. Από την άλλη οι μέθοδοι βασισμένες σε μάθηση, όπως οι μέθοδοι παραγοντοποίησης, προϋποθέτουν την γνώση του αριθμού των αντικειμένων και των χρηστών εξ' αρχής, αλλά παρέχουν μια ακριβέστερη και καλύτερα ταιριαστή στις προτιμήσεις των χρηστών λύση.

Από την άλλη, έννοιες όπως η καλή εύνοια της τύχης (serendipity), που μπορεί να παρουσιαστούν σε μεθόδους γειτονιών, σημαίνει ότι μπορεί να προταθούν στους χρήστες αντικείμενα, τα οποία μπορεί ποτέ να μην είχαν ανακαλύψει, με μια πιο στενή προσέγγιση προς τις προτιμήσεις τους.

Ταυτόχρονα, πρέπει κανείς να λάβει υπόψη τις ανάγκες ενός συστήματος. Για συστήματα γειτονιών, μικρότερος αριθμός αντικειμένων σημαίνει πως ένα σύστημα συστάσεων βασισμένο στους χρήστες μπορεί να προσφέρει μεγαλύτερη ακρίβεια, αλλά από την άλλη, ένα σύστημα συστάσεων βασισμένων στους χρήστες μπορεί να καταναλώνει λιγότερους πόρους. [1]

Είναι απαραίτητο ακόμη να αναλογιστεί κανείς τη φύση της πλειονότητας των υποδηλώσεων προτίμησης. Σε ένα σύστημα όπου ελάχιστοι δηλώνουν ρητά την προτίμηση τους με μια βαθμολόγηση, ένα σύστημα όπου θα λαμβάνονται υπόψη οι προτιμήσεις των χρηστών, δηλωμένες έμμεσα, μπορεί να αποβεί χρησιμότερο έναντι ενός απλού που λαμβάνει υπόψη του μόνο τις βαθμολογήσεις. [1]

Σε αυτό το κεφάλαιο αναπτύξαμε μεθόδους γειτονιών και δείξαμε πως μπορούν να υλοποιηθούν με τη χρήση της γλώσσας Python. Στο επόμενο κεφάλαιο θα αναπτυχθούμε στην υλοποίηση μεθόδων μάθησης, με τη χρήση της βιβλιοθήκης PyTorch, και θα ανακαλύψουμε το πως ένα μοντέλο αλγορίθμου βασισμένου σε μάθηση εκπαιδεύεται, με την παρουσίαση παραδειγμάτων κώδικα.

ΚΕΦΑΛΑΙΟ 3 – Η βιβλιοθήκη PyTorch

3.1 – Εισαγωγή

Η βιβλιοθήκη μηχανικής μάθησης και ανάλυσης δεδομένων PyTorch αποτελεί μια καινούρια πρόσθεση στο οικοσύστημα της βαθιάς μάθησης (Deep Learning).

Απόγονος της Torch, γραμμένη για την γλώσσα Lua, η PyTorch είναι ένα front end για τη μηχανή Torch, γραμμένη για την γλώσσα Python, με καλή υποστήριξη υπολογισμών σε επίπεδο GPU, με τη χρήση της τεχνολογίας CUDA. [12]

Ανεπτυγμένη από την ομάδα έρευνας τεχνητής νοημοσύνης του Facebook, η PyTorch διαφέρει από άλλες βιβλιοθήκες όπως η TensorFlow, με την διαφορά να έγκειται στον δυναμικό τρόπο προγραμματισμού της (define-by-run έναντι define-compile-run). Για τον λόγο αυτό η PyTorch αναφέρεται συχνά ως καταλληλότερη για πειραματισμό και ερευνητικούς σκοπούς, όπου βιβλιοθήκες σαν την TensorFlow προσφέρουν έτοιμες κατασκευές υπολογιστικών γράφων. Αντίθετα, στην PyTorch, ο χρήστης μπορεί να ορίσει χειροκίνητα οποιαδήποτε μαθηματική έκφραση και να καλέσει άμεσα τον υπολογισμό της. [12]

Σε αυτό το κεφάλαιο θα παρουσιάσουμε αρχικά βασικά συστατικά του προγραμματισμού σε PyTorch, τα οποία θα χρειαστούμε για την κωδικοποίηση ενός συστήματος συστάσεων βασισμένου σε μάθηση, και έπειτα θα επεκταθούμε στο πακέτο παράλληλης κατανεμημένης επεξεργασίας της PyTorch, torch.distributed, και το πακέτο torch.multiprocessing, επέκταση της βιβλιοθήκης multiprocessing της Python.

Κλείνοντας, στο επόμενο κεφάλαιο, θα παρουσιάσουμε δύο είδη κόμβων, ικανών να εκπαιδεύσουν συνεργατικά ένα μοντέλο συστήματος συστάσεων βασισμένο σε μάθηση, παραγοντοποίησης πινάκων.

3.2 – Γνωρίζοντας την PyTorch

3.2.1 – Tensors

Θα ξεκινήσουμε αρχικά με μία εισαγωγή στο βασικότερο αντικείμενο της PyTorch, τον τανυστή (Tensor). Οι τανυστές στην PyTorch είναι αντικείμενα παρόμοια με τα

διανύσματα NumPy, με βασικές διαφορές να αποτελούν ότι ένας τανυστής είναι σταθερών διαστάσεων και πως πράξεις μεταξύ των τανυστών μπορούν να υλοποιηθούν όχι μόνο στην κύρια μνήμη, με τη βοήθεια του επεξεργαστή, αλλά και σε κάρτες γραφικών γενικού σκοπού, όπως οι κάρτες GeForce της nVidia, δεδομένου ότι υποστηρίζουν την τεχνολογία CUDA και ότι είναι υπολογιστικής ικανότητας 3.0 και άνω, όπως αυτή ορίζεται από την nVidia (Ιανουάριος 2019, PyTorch 1.0). Αξίζει να σημειωθεί ακόμη πως ένα διάνυσμα NumPy μπορεί εύκολα να μεταφραστεί σε torch tensor (όπως θα αναφερόμαστε από δω και στο εξής στους τανυστές) και αντιστρόφως, ένας tensor να μεταφραστεί σε διάνυσμα numpy. [12][13]

Ας ξεκινήσουμε από τα βασικά και ας φτιάξουμε μερικά διανύσματα numpy και έναν tensor. Θα χρειαστούμε τις βιβλιοθήκες όπως ορίζονται, numpy, για την NumPy, και torch για την PyTorch. Θα παρατηρήσετε πως για τις περιπτώσεις των διανυσμάτων, αρχικά φτιάχνουμε ένα διάνυσμα 3*4 και έπειτα ένα διάνυσμα το οποίο περιέχει τα στοιχεία 3.2 και 4.8. Ακόμη, φτιάχνοντας έναν tensor με τον προκαθορισμένο τρόπο, torch.Tensor, παίρνουμε πίσω έναν double floating point tensor. Ο tensor θα είναι μονοδιάστατος και θα περιέχει τα στοιχεία 2 και 3. [12]

```
import torch #PyTorch
import numpy as np #numpy

array1 = np.ndarray([3,4]) #3*4 array
array2 = np.array([3.2,4.8]) #array containing elements 3 and 4

torch.Tensor([2,3]) #double floating point tensor containing
elements 2 and 3
```

Κώδικας 17 – NumPy arrays και PyTorch tensor

Έπειτα, φτιάχνουμε 2 tensors από αυτά τα δύο array, έναν double floating point tensor και έναν long integer 64 bit tensor. Στην δεύτερη περίπτωση μπορούμε να επιτύχουμε κάτι τέτοιο γράφοντας torch.LongTensor. Θα παρατηρήσετε ακόμη πως στην δεύτερη περίπτωση, παρόλο που δίνουμε σαν input ένα floating point array, το αποτέλεσμα είναι σε μορφή integer. Στην προκειμένη περίπτωση λοιπόν ο tensor2 θα περιέχει τα στοιχεία 3 και 4, παρομοίως και ο tensor1. Επιπλέον, για να δείξουμε την μετάβαση από tensor σε numpy array, φτιάχνουμε από τον tensor2 το array3. [12]

```
tensor1 = torch.Tensor(array1)
tensor2 = torch.LongTensor(array2)

array3 = tensor2.numpy()
```

Κώδικας 18 – Tensors, 1

Ας φτιάξουμε μερικούς ακόμη tensors. Έστω πως θέλουμε να δημιουργήσουμε έναν floating point tensor και 2 double floating point, 2 διαστάσεων. Μπορούμε να ορίσουμε εναλλακτικά, όπως και παραπάνω, ρητά το είδος των tensors που επιθυμούμε.

Ο ένας τρόπος είναι πληκτρολογώντας torch.FloatTensor, για τον floating point tensor, και torch.DoubleTensor, για τους double floating point tensors. Το ίδιο μπορούμε να κάνουμε για οποιοδήποτε είδος tensor, γράφοντας torch.ShortTensor (signed), torch.ByteTensor (unsigned), torch.CharTensor (8bit signed), torch.HalfTensor (16bit unsigned) κτλ. [12]

Ο δεύτερος τρόπος είναι γράφοντας torch.Tensor και δηλώνοντας data type (dtype). Για παράδειγμα, για έναν float tensor το dtype θα ήταν torch.float32 ή torch.float, για έναν double torch.float64 ή torch.double κοκ. [12]

Όπως και στα numpy arrays, ένας tensor πρέπει να έχει σταθερές διαστάσεις. Δηλώνοντας για παράδειγμα έναν integer tensor, όπου στην πρώτη γραμμή υπάρχουν δύο στοιχεία και στην δεύτερη ένα, η απάντηση που θα πάρουμε είναι ένα ValueError.

Παρατηρείστε ακόμη τι συμβαίνει στην περίπτωση που δηλώσουμε έναν αριθμό μεγαλύτερο ή μικρότερο του επιτρεπόμενου ορίου, όριο το οποίο εξαρτάται από το είδος του tensor, για ένα στοιχείο του tensor. Τα αποτελέσματα μπορεί να αποβούν απροσδόκητα, αναλόγως του μεγέθους του αριθμού, με το στοιχείο να αλλάζει πρόσημο όπως και τιμή. Υπερβαίνοντας ελαφρώς για παράδειγμα το όριο 32767, το οποίο είναι και η μέγιστη τιμή δυνατή τιμή για έναν ακέραιο 16 bit με πρόσημο, έστω 33000, παίρνουμε πίσω μια αρνητική τιμή, μετρώντας ανάποδα από το χαμηλότερο όριο, -32768, για το ποσό κατά το οποίο υπερβήκαμε το υψηλό όριο. Εάν το όριο 32767 συμβολίζεται ως 1111 1111 1111 1111 και ο αριθμός 33000 συμβολίζεται ως 1 0000 0000 1110 1000 στο δυαδικό σύστημα, τότε με μια αποκοπή στο πρώτο στοιχείο για 16 bit, παίρνουμε τον αριθμό 0000 0000 1110 1000 = 232. Για την περίπτωση ενός ακεραίου με πρόσημο και μετρώντας από το

χαμηλό όριο, -32768, παίρνουμε πίσω την τιμή -32536, η οποία θα είναι και η τιμή του στοιχείου του tensor.

```
float_tensor1 = torch.FloatTensor([[5,7],[1,8]])
double_tensor1 = torch.DoubleTensor([2.8,7.1,3])
double_tensor2 = torch.Tensor([[3,12],[5,1],[2,9]],
dtype=torch.float65)

int_tensor1 = torch.IntTensor([[3,12],[7]]) #error

short_tensor1 = torch.ShortTensor([33000,528]) #[-32536,  528]
short_tensor1 = torch.ShortTensor([8,7])
```

Κώδικας 19 – Tensors, 2

Όπως αναφέραμε, μπορούμε ακόμη να κάνουμε υπολογισμούς ανάμεσα σε tensors, με τη χρήση επιτάχυνσης της τεχνολογίας CUDA. Καλώντας απλώς την μέθοδο .cuda() σε οποιονδήποτε tensor, ο tensor μας αποθηκεύεται στην μνήμη της κάρτας γραφικών, από όπου και μπορούμε να κάνουμε υπολογισμούς μεταξύ άλλων tensors. Είναι σημαντικό παρ' όλα αυτά να είμαστε σίγουροι πριν την εκτέλεση των πράξεων πως οι tensors βρίσκονται στην ίδια τοποθεσία, δηλαδή, είτε να βρίσκονται και οι δυο στην κύρια μνήμη, είτε και οι δυο στην μνήμη της κάρτας γραφικών. Όπως είναι λοιπόν αναμενόμενο, μια πράξη ανάμεσα σε έναν cuda tensor και έναν cpu tensor, θα αποφέρει ένα μήνυμα error. Ακόμη, πρέπει να είμαστε επιπλέον προσεκτικοί κατά τους υπολογισμούς μας, αφού δύο tensors πρέπει να είναι συμβατοί ο ένας με τον άλλο, ήτοι παρόμοιοι, πχ. και οι δύο τύπου integer, float κτλ. [12][13]

```
cuda_tensor1 = tensor1.cuda()
print(cuda_tensor1*tensor2) #error

cuda_short_tensor1 = short_tensor1.cuda()
new_cuda_tensor = cuda_short_tensor1*cuda_tensor1 #error
```

Κώδικας 20 – Tensors, 3

Σε περίπτωση που διαθέτουμε δύο ή περισσότερες κάρτες γραφικών, μπορούμε να ορίσουμε επιπλέον σε ποια θα θέλαμε να τοποθετήσουμε έναν tensor. Αυτό μπορεί να γίνει με τη χρήση της μεθόδου device(). Εναλλακτικά, αν δεν διαθέτουμε δύο κάρτες γραφικών ή εάν θέλουμε να επανατοποθετήσουμε έναν tensor στην

κύρια μνήμη, μπορούμε απλώς να χρησιμοποιήσουμε τις μεθόδους cuda() και cpu() αντίστοιχα. [12][13]

```
cpu_short_tensor1 = cuda_short_tensor1.device("cpu") #.cpu()
cuda0_short_tensor1 = cpu_short_tensor1.device(0) #gpu device 0
cuda1_short_tensor1 = cpu_short_tensor1.device(1) #gpu device 1
```

Κώδικας 21 – Tensors, 4

Μια άλλη τακτική για να επιτύχουμε το αποτέλεσμα του Κώδικα 21 είναι η χρήση της μεθόδου to(), με την οποία μπορούμε να αλλάξουμε την τοποθεσία ενός tensor αλλά και τον τύπο του. [12][13]

```
cuda_short_tensor1 = cuda_short_tensor1.to(1) #copy to gpu device
1
cuda_int_tensor1 = cuda_short_tensor1.to(torch.int32) #cast to
integer
print(cuda_short_tensor1.to(torch.long)) #cast to long integer
```

Κώδικας 22 – Tensors, 5

3.2.2 – Autograd

Σε αντίθεση με ένα numpy array, ένας tensor αποθηκεύει εσωτερικά του ακόμη έναν tensor, τον gradient (κλίση). Σε έναν υπολογιστικό γράφο, ο gradient αναπαριστά ένα ποσό, σύμφωνα με το οποίο τα στοιχεία του tensor θα αλλάξουν κατά την διαδικασία εκπαίδευσης σε μια εποχή. Ο tensor σε αυτή την περίπτωση αναπαριστά έναν κόμβο στον υπολογιστικό γράφο. [14][15]

Δηλαδή, εάν για έναν tensor x ισχύει x.required_grad = True, ιδιότητα που υποδηλώνει την ύπαρξη gradient tensor για τον tensor x, τότε υπάρχει gradient tensor x.grad που περιέχει την τιμή της κλίσης.

Έστω ένας tensor x, όπως παρουσιάστηκε παραπάνω, και ένας tensor y, για τον οποίο ισχύει y.requires_grad = False. Έστω ακόμη μια συνάρτηση loss, mean squared error (mse_loss), τύπου torch.nn.functional, η οποία είναι συνάρτηση του tensor y και μιας άλλης συνάρτησης α, έτσι ώστε loss(y;α(x)). Εφόσον για τον tensor x δίνεται x.requires_grad = True, υπολογίζοντας την τιμή της loss ανάμεσα στον tensor y και την συνάρτηση α, κατά τη διαδικασία μιας εκπαίδευσης για παράδειγμα, και καλώντας loss.backward(), υπολογίζεται ο gradient tensor, x.grad, για τον tensor x με βάση την συνάρτηση loss.

Η μέθοδος backward ενός tensor δεν κάνει τίποτε άλλο στην τελική παρά να υπολογίσει τις μερικές παραγώγους, για όσους tensors συμμετείχαν στον υπολογισμό του tensor αυτού. Για να αξιολογήσουμε ποιοι tensor και πως συμμετείχαν στην διαδικασία, δημιουργείται για κάθε tensor ένα αντικείμενο grad_fn (gradient function), το οποίο αναπαριστά την συνάρτηση που δημιούργησε τον tensor και καταγράφει όσους tensors συμμετείχαν και περιείχαν gradients. Καλώντας δηλαδή την μέθοδο backward, υπολογίζουμε απλώς τους gradient tensors (μερικές παραγώγους) για όσους tensors συμμετείχαν στον υπολογισμό και περιείχαν gradients.

Στην συνέχεια, με την βοήθεια ενός optimizer, πχ. ενός απλού SGD (Stochastic Gradient Descent), ο οποίος θα δεχθεί σαν παράμετρο τον tensor x, θα εκτελέσει ένα βήμα με βάση ένα προκαθορισμένο learning rate, κατά το οποίο υπολογίζονται τα νέα στοιχεία του κόμβου tensor x, ανάλογα με το είδος του optimizer, το learning rate και τον gradient tensor του tensor x, x.grad. [15][16]

Παρακάτω παρουσιάζουμε για πρώτη φορά ένα απλό παράδειγμα υλοποίησης εκπαίδευσης, εμπνευσμένο από τα tutorials της PyTorch. Ένας tensor x αποτελεί το input, ενώ ένας tensor y αποτελεί τους στόχους της εκπαίδευσης, και οι δύο με παράμετρο requires_grad = False. Θα εκπαιδεύσουμε ένα μικρό δίκτυο δύο επιπέδων, με ενδιάμεση συνάρτηση μια συνάρτηση clamp, με ελάχιστο όριο 0. Στόχος είναι η εύρεση βαρών w1 και w2 ώστε να προσεγγίσουμε, μέσω του δικτύου, όσο καλύτερα μπορούμε τον tensor y. [15][17]

```
#importing

import torch #PyTorch
import torch.nn.functional as F #functions
import torch.optim as optim #optimizer

#we demonstrate a two-layer network. a clamp function is used to
#perform calculations for the hidden layer output. the output
#layer is a simple linear function

#number of inputs, input dimension, hidden layer dimension, output
#dimension
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
#make the input and output tensors
x = torch.randn(N, D_in, dtype=torch.float)
y = torch.randn(N, D_out, dtype=torch.float)

#make the weights. the weights are the ones updated, therefore we
#set requires_grad as true
w1 = torch.randn(D_in, H, dtype=torch.float, requires_grad=True)
w2 = torch.randn(H, D_out, dtype=torch.float, requires_grad=True)

#set the learning rate and the number of epochs
learning_rate = 1e-6
epochs = 500

#make the optimizer. the parameters should be an iterable
#structure, in this case a list. the optimizer updates the weights
#w1 and w2 based on their respectable gradients upon calling the
#optimizer step function
optimizer = optim.SGD([w1,w2], lr=learning_rate)

#train for epochs
for epoch in range(epochs)
     #make the prediction by:
     #multiplying x by w1, performing a clamp function, then
     #multiplying with w2
     y_pred = x.mm(w1).clamp(min=0).mm(w2)

     #calculate the loss between the prediction tensor and the y
     #(output) tensor
     loss = F.mse_loss(y_pred, y)
     #always zero the gradients before calling the backward
     #function
     optimizer.zero_grad()

     #what is the loss in each epoch?
     print(f“Epoch {epoch}, training loss: {loss.item()}”)
     loss.backward() #calculate the gradients
     optimizer.step() #perform a step (calculate the new weights)
     #w1 and w2 have been updated
```

Κώδικας 23 – Εκπαίδευση

Μπορούμε ακόμη, αν επιθυμούμε, να γράψουμε δικές μας συναρτήσεις, επεκτείνοντας περαιτέρω το αντικείμενο torch.autograd.Function, με τον τρόπο που

φαίνεται στο επόμενο παράδειγμα. Το αντικείμενο ctx (context) είναι ένα αντικείμενο περιεχομένου στο οποίο μπορούμε να αποθηκεύσουμε πληροφορίες για την κλήση της backward μεθόδου, με την χρήση της μεθόδου ctx.save_for_backward, κατά την μέθοδο forward. Αντίστοιχα, για την backward, για να λάβουμε και πάλι το input που χρειαζόμαστε κατά τους υπολογισμούς, απλώς καλούμε την μέθοδο ctx.saved_tensors για το αντικείμενο ctx, στο οποίο έχουμε αποθηκεύσει το input. [18]

```
class myFunction(torch.autograd.Function)

     @staticmethod
     def forward(ctx, input)
          ctx.save_for_backward(input)
          #...
          #..
          #.
          #pass
          return 0

     @staticmethod
     def backward(ctx, grad_output)
          input, = ctx.saved_tensors
          #...
          #..
          #.
          #pass
          return 0
```

Κώδικας 24 – Επέκταση του αντικειμένου torch.autograd.Function

3.2.3 – Το nn (Neural network) module

Το torch.nn.Module αποτελεί τη βάση κάθε μοντέλου στην PyTorch και τη βάση όλων των στρωμάτων που παρέχει η PyTorch. Όλα τα στρώματα, από γραμμικά έως συστολικά, βασίζονται στο nn module.

Έστω για παράδειγμα, το δίκτυο δύο στρωμάτων, όπως παρουσιάσθηκε στον Κώδικα 23. Μπορούμε να αναπαραστήσουμε το δίκτυο αυτό, δημιουργώντας ένα δικό μας μοντέλο, έστω TwoLayerNet. Αξίζει να παρατηρηθεί πως ένα nn module μπορεί εσωτερικά του να περιέχει άλλα modules/layers, βάσει των οποίων μπορεί να γίνει μια πρόβλεψη. [18][19][20]

Για το μοντέλο μας θα χρησιμοποιήσουμε δύο linear modules για να αναπαραστήσουμε τα δύο στρώματα. Η συνάρτηση forward θα υπολογίσει την έξοδο όταν εμείς θα δώσουμε έναν αριθμό χαρακτηριστικών. Αξίζει όμως να δοθεί έμφαση στην εισαγωγή του σωστού αριθμού χαρακτηριστικών.

```
import torch.nn as nn #neural network module

#two-layer net
class TwoLayerNet(nn.Module):

    #constructor
    def __init__(self, D_in=1000, H=100, D_out=10):
        super(TwoLayerNet, self).__init__() #call base constructor
        self.linear1 = nn.Linear(D_in, H) #linear layer 1
        self.linear2 = nn.Linear(H, D_out) #linear layer 2

    #forward function (prediction)
    def forward(self, idx):
        #hidden layer output
        hidden_out=self.linear1(idx).clamp(min=0)
        #output layer output
        output_out = self.linear2(hidden_out)
        return output_out


#create a new model based on the TwoLayerNet module

#we use the default dimension values(1000, 100, 100)
#no explicit declaration
two_layer = TwoLayerNet()

#number of inputs:64
#dimension value: 1000 (same as the input dimension)
input_tensor = torch.randn(64,1000)

#output dimension is 64*10: (number of inputs)*(output dimension)
#output is determined by the forward function
print(two_layer(input_tensor))
```

Κώδικας 25 – Two-layer network module

```
import torch #PyTorch
import torch.nn as nn #neural network module
```

```
import torch.nn.functional as F #functions

from models import TwoLayerNet

#number of inputs, input dimension, hidden layer dimension, output
#dimension
N, D_in, H, D_out = 64, 1000, 100, 10

#make the input and output tensors
x = torch.randn(N, D_in, dtype=torch.float)
y = torch.randn(N, D_out, dtype=torch.float)

#make the model
two_layer = TwoLayerNet()

#set the learning rate and the number of epochs
learning_rate = 1e-6
epochs = 500

#make sure the model (weights) is at a definite location before
#making the optimizer
optimizer = optim.SGD(two_layer.parameters(), lr=learning_rate)

#train for epochs
for epoch in range(epochs)
     #make the prediction
     y_pred = two_layer(x)

     #calculate the loss between the prediction tensor and the y
     #(output) tensor
     loss = F.mse_loss(y_pred, y)
     optimizer.zero_grad() #zero the gradients

     #show loss
     print(f“Epoch {epoch}, training loss: {loss.item()}”)
     loss.backward() #calculate the gradients
     optimizer.step() #perform a step (calculate the new weights)
     #the model layer parameters have been updated
```

Κώδικας 26 – Two-layer network module training

3.2.4 – Datasets, Dataloaders και batches

Σε προηγούμενο κεφάλαιο είδαμε πως μπορούμε να αξιοποιήσουμε pandas dataframes για την δημιουργία train και test sets και είδαμε ακόμη πως μπορούμε να εκπαιδεύσουμε ένα μοντέλο με τη χρήση των nn module και autograd.

Σε αυτή την ενότητα θα παρουσιάσουμε πως μπορούμε να αξιοποιήσουμε επίσης τις κλάσεις Dataset, DataLoader, Sampler και θα δείξουμε πως μπορούμε να διαλύσουμε την φάση της εκπαίδευσης σε μικρότερα πακέτα (batches).

Ας ξαναγυρίσουμε πίσω στην ενότητα 2.2.9, στο παράδειγμα που παρουσιάσαμε για την δημιουργία των test και train sets. Έστω πως επιθυμούμε, για οποιονδήποτε λόγο, να φτιάξουμε και εμείς τα δικά μας Dataset από το test και το train set της ενότητας 2.2.9. Ένα torch dataset, αποτελείται από τρεις βασικές μεθόδους που θα χρειαστεί να κάνουμε override (υλοποιήσουμε): έναν δομητή, μια μέθοδο επιστροφής ενός αντικειμένου και μια μέθοδο μέτρησης μήκους. [20][21][22]

Σημαντικό είναι ακόμη όσες αρχικοποιήσεις μπορούν να γίνουν να γίνονται μέσα στον δομητή. Έτσι, μπορούμε να εξοικονομήσουμε υπολογισμούς που διαφορετικά θα γίνονταν κατά την επιστροφή ενός αντικειμένου.

Παρακάτω δίνουμε δύο παραδείγματα κώδικα Dataset για τα train και test set, με τον τρόπο που υλοποιήθηκαν στην 2.2.9.

```
#datasets.py

import torch
from torch.utils.data import Dataset

#collaborative filtering dataset based on the MovieLens datasets
class CF_Dataset(Dataset):

    #constructor
    #data initiliaziation is best done in the init method.
    #optimize code to reduce training time
    def __init__(self, data):
        self.data = data
        self.length = len(data)
        self.users = torch.LongTensor(data['userId'].values)
        self.items = torch.LongTensor(data['movieId'].values)
        self.ratings = torch.FloatTensor(data['rating'].values)
```

```
        self.timestamp =
torch.LongTensor(data['timestamp'].values)

    #get item by index number
    def __getitem__(self, idx):
        return {'userId':self.users[idx],
'movieId':self.items[idx], 'rating':self.ratings[idx],
'timestamp':self.timestamp[idx]}

    #dataset length
    def __len__(self):
        return self.length
```

Κώδικας 27 – Collaborative filtering dataset

```
#test.py

from datasets import CF_Dataset

#making datasets
train_dataset = CF_Dataset(train_set)
test_dataset = CF_Dataset(test_set)
```

Κώδικας 28 – Από pandas dataframe σε torch dataset

Έχοντας τα dataset που φτιάξαμε στα παραπάνω παραδείγματα, ας θεωρήσουμε ότι επιθυμούμε με κάποιο τρόπο να τα χρησιμοποιήσουμε στην εκπαίδευση μας και να δοκιμάσουμε ακόμη να χρησιμοποιήσουμε τα αντικείμενα Sampler και DataLoader. Ένας Sampler είναι ένα δειγματολόγιο ενός Dataset. Στο παρακάτω παράδειγμα χρησιμοποιούμε έναν RandomSampler, για όλο το μήκος του Dataset (default: dataset length), αλλάζοντας πρακτικά μονάχα την σειρά των δειγμάτων. Εντέλει, ένας sampler δεν είναι παρά μια κατασκευή που αναπαριστά έναν κατάλογο επιλεγόμενων δειγμάτων. [21]

Ένας DataLoader λαμβάνει το dataset και τον sampler, διαχωρίζει το dataset σε πακέτα ορισμένου μεγέθους, χρησιμοποιώντας, ή μη, workers (διεργασίες) για το ταχύτερο φόρτωμα των δεδομένων. Απαριθμώντας τον DataLoader μπορούμε να πάρουμε πίσω, ανά εποχή, καθένα από τα batches και να εκπαιδεύσουμε το μοντέλο μας με βάση αυτά. [21]

```
#training.py

Import torch.nn.functional as F #functions
```

```
import torch.optim as optim #optimizer
import torch.utils.data as data #utils data

sampler = data.RandomSampler(train_dataset)

dataloader = data.DataLoader(train_dataset, batch_size=100,
num_workers=2, sampler=sampler)

#Adam optimizer, with small weight decay
optimizer = optim.Adam(model.parameters(), lr=learning_rate,
wd=0.01)

#set the model in training mode (most of the time not obligatory)
model.train()

for epoch in range(epochs):
     for batch_idx, batch_sample in enumerate(dataloader):

          y_hat = model(batch_sample) #prediction
          loss = F.mse_loss(y_hat, output) #loss
          optimizer.zero_grad() #zero gradients
          loss.backward() #calculate gradients
          #perform a step/recalculate model weights
          optimizer.step()
```

Κώδικας 29 – Εκπαίδευση με την χρήση Datasets, Sampler και DataLoader

3.2.5 – Αποθήκευση και φόρτωση

Ένα state dictionary (state_dict) στην PyTorch αναπαριστά για κάθε αντικείμενο όλες τις παραμέτρους που μπορούν να εκπαιδευτούν. Για παράδειγμα, για το TwoLayerNet του Κώδικα 25, το state_dict θα περιέχει τα παραμετροποιήσιμα βάρη και πολώσεις των δύο γραμμικών στρωμάτων, linear1 και linear2. [23]

Ένας τρόπος να αποθηκεύσουμε ένα μοντέλο, είναι η αποθήκευση αυτού του dictionary. Αποθηκεύοντας το dictionary κατάστασης, μπορούμε ανά πάσα στιγμή να το φορτώσουμε και πάλι σε ένα μοντέλο πανομοιότυπο σε διαστάσεις του πρώτου, με τη χρήση των μεθόδων torch.save(), torch.load() και torch.load_state_dict(). [23]

```
#save state dictionary
torch.save(model.state_dict(), PATH)
```

```
#load state dictionary
model = Net(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
```

Κώδικας 30 – Αποθήκευση κατάστασης μοντέλου και φόρτωση

Έστω όμως ακόμη, πως δεν γνωρίζουμε το είδος του μοντέλου αλλά ούτε και τις διαστάσεις του. Μια εναλλακτική λύση είναι η αποθήκευση ολόκληρου του μοντέλου, και η ανάκτηση του.

```
#save model
torch.save(model, PATH)

#load model
model = torch.load(PATH)
```

Κώδικας 31 – Αποθήκευση μοντέλου και φόρτωση

Πέρα όμως από μοντέλα, έχουμε την επιλογή να αποθηκεύσουμε ακόμη αντικείμενα όπως ένα dataset, τρέχουσα εποχή ή αριθμό εποχών, απώλεια σφάλματος κοκ. Αυτό μπορεί να γίνει σαν μια αναφορά checkpoint κατά τη διάρκεια εκτέλεσης του πρόγραμμα. Μπορούμε ακόμη να αποθηκεύσουμε καταστάσεις πολλαπλών μοντέλων σε ένα μόνο αρχείο ή να αποθηκεύσουμε μερικές παραμέτρους ενός εκπαιδευμένου μοντέλου A για ένα νέο μοντέλο B, για σκοπούς προεκπαίδευσης. Πρακτικά, με τις λειτουργίες torch.save() και torch.load() μπορούμε να αποθηκεύσουμε και να φορτώσουμε σχεδόν οτιδήποτε. [23]

```
#save dataset
torch.save(train_dataset, PATH)

#load dataset
train_dataset = torch.load(PATH)

#save checkpoint
torch.save({
‘current_epoch’: epoch,
‘epochs’: epochs
‘model state’: model.state_dict(),
‘optimizer state’: optimizer.state_dict(),
‘loss’: loss, ...
}, PATH)

#load checkpoint
```

```
model = Net(*args, **kwargs)
optimizer = some_optimizer(*args, **kwargs)

checkpoint = torch.load(PATH)
optimizer.load_state_dict(checkpoint['optimizer state'])
model.load_state_dict(checkpoint['model state'])
epochs = checkpoint['epochs']
epoch = checkpoint['current epoch']
loss = checkpoint['loss']

#save multiple
torch.save({'modelA state': modelA.state_dict(),
'modelB state': modelB.state_dict(),
'optimizerA state': optimA.state_dict(),
'optimizerB state': optimB.state_dict(), ...
}, PATH)

#load multiple
modelA = NetA(*args, **kwargs)
modelB = NetB(*args, **kwargs)
optimA = some_optimizer(*args, **kwargs)
optimB = some_other_optimizer(*args, **kwargs)

checkpoint = torch.load(PATH)
modelA.load_state_dict(checkpoint['modelA state'])
modelB.load_state_dict(checkpoint['modelB state'])
optimA.load_state_dict(checkpoint['optimizerA state'])
optimB.load_state_dict(checkpoint['optimizerB state'])
```

Κώδικας 32 – Αποθήκευση και φόρτωση dataset, checkpoint και παραμέτρων πολλαπλών μοντέλων

Μιλήσαμε νωρίτερα για το πως ένας tensor μπορεί να τοποθετηθεί, είτε στην κύρια μνήμη, είτε στην μνήμη κάρτας γραφικών γενικού σκοπού. Παρόμοιες ιδιότητες παρουσιάζουν τα nn modules. Μπορούμε δηλαδή για παράδειγμα, απλώς καλώντας την μέθοδο cuda() του μοντέλου, να μπορέσουμε να επωφεληθούμε από τις δυνατότητες της τεχνολογίας CUDA για υπολογισμούς ανάμεσα σε tensors του μοντέλου.

Έστω, πως έχουμε στα χέρια μας ένα μοντέλο στην μνήμη της κάρτας γραφικών μας. Αποθηκεύοντας το μοντέλο και επαναφορτώνοντας το, μια ενδιαφέρουσα

ιδιότητα που θα παρατηρήσουμε, είναι το γεγονός ότι το μοντέλο θα φορτωθεί στην ίδια τοποθεσία που βρισκόταν, ήτοι στην κάρτα γραφικών. [23]

Μπορούμε όμως πολύ εύκολα να το μετακινήσουμε, είτε με τις μεθόδους, cuda(), cpu(), device() και to(), με τον τρόπο που παρουσιάστηκαν στην ενότητα 3.2.1, είτε απλώς ορίζοντας κατά την φόρτωση την τοποθεσία με την παράμετρο map_location. [23]

```
#save model
torch.save(cuda_model, PATH)

#load model to another location
model = torch.load(PATH, map_location = '0')
#or
model = torch.load(PATH, map_location = 'cuda:1')
#or
model = torch.load(PATH, map_location = torch.device('0'))
#or
model = torch.load(PATH, map_location = 'cpu')
#or etc...
```

Κώδικας 33 – Αποθήκευση και φόρτωση μοντέλου σε διαφορετική τοποθεσία

3.2.6 – CUDA

Έχοντας δείξει τα κυριότερα χαρακτηριστικά της PyTorch, μπορούμε να πούμε με σιγουριά, πως οποιοσδήποτε έχει καταλάβει ως εδώ το κεφάλαιο 3, είναι ικανός πλέον να υλοποιήσει σχεδόν οτιδήποτε σε ένα καλό επίπεδο με την βιβλιοθήκη PyTorch.

Θα θέλαμε όμως να επεκταθούμε σε μερικές ακόμη χρήσιμες συμβουλές και tips. Έστω, πως ένας αγανακτισμένος προγραμματιστής, γυρνώντας από μια δύσκολη ημέρα, αποφασίζει ένα βράδυ να καταπιαστεί περισσότερο με την PyTorch, αναζητώντας να αναπτύξει τα ταλέντα του στην επιστήμη των δεδομένων (Data Science). Περνώντας ώρες και μέρες πάνω από το γραφείο του, προσπαθώντας να κατανοήσει την λογική της PyTorch, ανακαλύπτει, όλως τυχαίως, πως ο φορητός του υπολογιστής έχει μια κάρτα γραφικών GeForce, ικανή να τον βοηθήσει περαιτέρω στις δοκιμές του.

Ίσως το πρώτο πράγμα που θα έπρεπε να μάθει αυτός ο προγραμματιστής είναι πως να μην υπερφορτώνει την κάρτα γραφικών του. Άλλωστε ένας φορητός υπολογιστής δεν έχει και τις καλύτερες δυνατότητες, πράγμα που καταλαβαίνει

αφότου την έχει «πατήσει» 4 φορές ήδη προσπαθώντας να καταλάβει τι κάνει λάθος, όταν στην πορεία εκτέλεσης ενός προγράμματος, ο υπολογιστής του μοιάζει να μην ανταποκρίνεται.

Ένα καλό εργαλείο για αυτές τις περιστάσεις είναι η εντολή τερματικού `nvidia-smi`, ή ακόμη καλύτερα `nvidia-smi -l 1`, η οποία εμφανίζει την κατάσταση της κάρτας γραφικών ανά την μικρότερη δυνατή μονάδα χρόνου, 1 second. Διαφορετικά, checkpoints μπορούν να κωδικοποιηθούν κατά την διάρκεια εκτέλεσης για λόγους debugging, με print της χρησιμοποιούμενης μνήμης, μέσω των μεθόδων torch.cuda.memory_allocated(), torch.cuda.max_memory_allocated(), torch.cuda.memory_cached() και torch.cuda.max_memory_cached(). [24][25]

Εάν ο αγανακτισμένος προγραμματιστής μας επιθυμεί ακόμη να μάθει περισσότερα για την κάρτα γραφικών του μέσω της PyTorch, οι μέθοδοι torch.cuda.is_available(), torch.cuda.device_count(), torch.cuda.current_device(), torch.cuda.get_device_capability() και torch.cuda.get_device_name(), μπορούν να αποδειχθούν εξαιρετικά χρήσιμες. [24][25]

Είναι σημαντικό ακόμη να θυμόμαστε πως ο κάθε υπολογιστής δεν είναι ίδιος. Και για αυτό γράφοντας ένα πρόγραμμα, πρέπει να θυμόμαστε να ελέγχουμε εάν ο υπολογιστής μπορεί να υποστηρίξει επιτάχυνση με την τεχνολογία CUDA, διαφορετικά, μια υλοποίηση CPU μπορεί να αποβεί απαραίτητη. [25]

Θα θέλαμε να δείξουμε επίσης τι μπορεί να επιχειρήσει κανείς με ένα CUDA stream. Ένα CUDA stream αφορά μια σειριακή εκτέλεση ορισμένων υπολογισμών. Στο παρακάτω παράδειγμα, ένα stream s εκτελεί την μέθοδο sum(), η οποία μπορεί να τρέξει πριν καν η μέθοδος normal() τελειώσει. [24][25]

```
cuda = torch.device('cuda')
s = torch.cuda.Stream()
A = torch.empty(100,100), device=cuda).normal(0.0, 1.0)
#stream might start before normal finishes
with torch.cuda.stream(s):
     B = torch.sum(A)
```

Κώδικας 34 – CUDA stream

3.2.7 – Συμβουλές και χρήσιμες πληροφορίες

Μην κατακρατάς ιστορικό

Μια χρήσιμη συμβουλή είναι να αποφεύγει κανείς την κατακράτηση ιστορικού. Ένα χρήσιμο παράδειγμα:

Η συσσώρευση απώλειας σφάλματος σε μια μεταβλητή total_loss. Λόγω της φύσης της Python (call-by-object), κάθε φορά που κανείς αθροίζει, με έναν τελεστή +=, την απώλεια στην συνολική απώλεια, η μεταβλητή total_loss δεν αποτελεί πλέον έναν αριθμό, αλλά ένα εσωτερικό άθροισμα όλων των προηγούμενων απωλειών, οι οποίες κρατούνται στην μνήμη. [26]

Ένα διαφορετικό παράδειγμα:

Έστω πως ένας tensor x, τοποθετημένος στην μνήμη της κάρτας γραφικών, αποτελεί ένα στοιχείο μιας λίστας. Έπειτα, διαγράφουμε τον tensor, αλλά η λίστα παραμένει. Όσο η λίστα υπάρχει στην μνήμη, ο x θα καταλαμβάνει χρήσιμο χώρο στην μνήμη της κάρτας γραφικών.

```
total_loss = 0
for i in range(10000):
     optimizer.zero_grad()
     output = model(input)
     loss = criterion(output)
     loss.backward()
     optimizer.step()
     total_loss += loss #do not do that
```

Κώδικας 35 – Παράδειγμα προς αποφυγή, 1

```
x = torch.Tensor([5,2]).cuda()

list_1 = [1,5,x]
del x

#the tensor continues to occupy space in the memory until the list
#goes out of scope
```

Κώδικας 36 – Παράδειγμα προς αποφυγή, 2

Μην κατακρατάς tensors

Ένα για οποιονδήποτε λόγο υπάρχει ένας tensor ο οποίος δεν έχει πλέον άλλη λειτουργία για το πρόγραμμα μας, είναι μια πολύ καλή πρακτική να διαγράψουμε αυτόν τον tensor. [26]

```
x = torch.LongTensor([[7,2],[5,1]])

for i in range(5):
     multiplied = x*i
     total = total*multiplied

print(total)

#multiplied has only been useful inside the loop
#it would be a good practice to delete it
```

Κώδικας 37 – Παράδειγμα προς αποφυγή, 3

3.2.8 – Αναπαραξιμότητα

Ως τελευταία συμβουλή δίνουμε την εξής. Εάν θέλετε να είστε σίγουροι πως τα αποτελέσματα σας είναι σωστά και πως διαφορετικές συνθήκες δεν επηρεάζουν το πρόγραμμα σας, μπορείτε να θέσετε σταθερά seeds για τις βιβλιοθήκες numpy και torch.

Έχετε όμως στο νου, πως κάτι τέτοιο, με την χρήση workers για ένα DataLoader, μπορεί να επιφέρει ίδια αποτελέσματα για όλους τους workers, εάν οι workers χρησιμοποιούν την βιβλιοθήκη numpy. Ο αριθμός torch seed του κάθε worker είναι seed+worker id. Ο αριθμός seed όμως δεν αλλάζει για άλλες βιβλιοθήκες όπως πχ. η numpy. [27]

```
#reproducibility
torch.manual_seed(0)
np.random.seed(0)
```

Κώδικας 38 – Αναπαραξιμότητα

3.2.9 – Επίλογος: Ένα παράδειγμα αλγορίθμου παραγοντοποίησης σε PyTorch

Με βάση ό,τι μάθαμε μέχρι τώρα, πάμε να φτιάξουμε ένα απλό σύστημα συστάσεων παραγοντοποίησης SVD. Όπως και στην ενότητα 2.2.9, χωρίζουμε την διαδικασία σε πέντε μέρη [28]:

- Εισαγωγή βιβλιοθηκών και μεθόδων
- Προετοιμασία δεδομένων
- Μοντελοποίηση
- Εκπαίδευση
- Αξιολόγηση

Part 1: Import the libraries and methods

Ξεκινάμε εισάγοντας τις κατάλληλες μεθόδους. Θα χρειαστούμε:

- Μια μέθοδο επιλογής διαχωρισμού των δεδομένων σε training και test set (input_select)
- Μια μέθοδο κωδικοποίησης των δεδομένων (encode)
- Μια μέθοδο διαχωρισμού των δεδομένων (split)
- Την μέθοδο εκπαίδευσης (train_model)
- Την μέθοδο αξιολόγησης (test_loss)
- Και ακόμη μια μέθοδο, την set_model_untrained_weights. Ο λόγος ύπαρξης μιας τέτοιας μεθόδου είναι απλός αλλά και εξαιρετικά έξυπνος. Συχνά, κατά τον διαχωρισμό των δεδομένων, τυγχάνει αντικείμενα ή χρήστες να παρουσιάζονται στο training set αλλά όχι στο test set. Η μέθοδος set_model_untrained_weights υπολογίζει τον μέσο όρο των παραγόντων των χρηστών και αντικειμένων και ορίζει για χρήστες και αντικείμενα που δεν εκπαιδεύτηκαν ποτέ, λόγω απώλειας τους από το train set, τα βάρη τους στο μέσο αυτό όρο. Σκεφτείτε ένα καθημερινό σύστημα σύστασης, πχ. τον αλγόριθμο του YouTube. Εάν ένας νέος χρήστης εμφανιστεί που το σύστημα δεν αναγνωρίζει, ο αλγόριθμος το καλύτερο που μπορεί να κάνει είναι να του παρουσιάσει το τι ο μέσος χρήστης συχνά αρέσκεται στο να παρακολουθεί. Μια τέτοιου είδους λύση αναφέρονται τακτικά ως λύση του προβλήματος της ψυχρής αρχής (cold start problem solution), πρόβλημα το οποίο τακτικά συναντάται σε συστήματα συστάσεων, και σηματοδοτεί την ύπαρξη νέων χρηστών ή αντικειμένων. [1][2]

Θα χρειαστούμε ακόμη τις κλάσεις:

- Μοντελοποίησης. Στην περίπτωση αυτή θα χρησιμοποιήσουμε ένα δικό μας nn module, με όνομα SVD, όπως ο αλγόριθμος, δημιουργημένο με τρόπο παρόμοιο με αυτόν που έχουμε δείξει ήδη στον Κώδικα 25.
- Dataset δεδομένων. Το CF_Dataset που έχουμε δημιουργήσει ήδη είναι επαρκές για τις ανάγκες μας.

Και τις εξής βιβλιοθήκες:

- numpy, για τους υπολογισμούς μας ανάμεσα σε arrays και την πιθανή υλοποίηση αναπαραξιμότητας (np.random.seed(0))
- pandas, για την δημιουργία των training και test set
- pathlib, για την εύρεση των datasets στο σύστημα αρχείων
- time, για τον χρονομέτρηση της διαδικασίας της μάθησης
- os, για τον πιθανό ορισμό OpenMP threads, που η PyTorch μπορεί να χρησιμοποιήσει
- sys, για την πιθανή διακοπή του προγράμματος (sys.exit())
- torch, για την PyTorch
- torch.nn, για την δημιουργία του SVD μοντέλου
- torch.nn.functional, για την συνάρτηση απώλειας
- torch.utils.data για την δημιουργία του Dataset, την χρήση Dataloader και Sampler
- sklearn, για μερικές υλοποιήσεις ανακατέματος δεδομένων
- threading και queue, για την αποστολή των μηνυμάτων εκπαίδευσης σε ένα ξεχωριστό νήμα, για σκοπούς μείωσης διακοπών input/output [30][31]

```
#part 1: importing

#coded methods
from split_selection import input_select
from data_encode import encode, split
from models import SVD
from datasets import CF_Dataset
from training import train_model, test_loss,
set_model_untrained_weights

#libraries
import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
import time #time
#import os #operating system
#import sys #system
import torch #PyTorch
#import torch.nn as nn #Neural network module
#import torch.nn.functional as F #functions
#import torch.utils.data as data #utils data
```

```
#from sklearn import shuffle #scikit-learn
#import threading #threads
#import queue #queue
```

Κώδικας 39 – Μέθοδοι παραγοντοποίησης σε PyTorch, Part 1: importing

```
#sets the number of OpenMP threads to be used by PyTorch to the
maximum.
#the default is half of that amount
#torch.set_num_threads(os.cpu_count())
#torch.set_num_threads(1)
```

Κώδικας 40 – Μέθοδοι παραγοντοποίησης σε PyTorch, OpenMP threads

```
#reproducibility
torch.manual_seed(0)
np.random.seed(0)
```

Κώδικας 41 – Μέθοδοι παραγοντοποίησης σε PyTorch, αναπαραξιμότητα

Part 2: Prepare the data

Επόμενο βήμα είναι η προετοιμασία των δεδομένων μας. Θα χρησιμοποιήσουμε το MovieLens dataset των 100 χιλιάδων βαθμολογήσεων για τον σκοπό υλοποίησης test και train set (ml-latest-small). [6] Κωδικοποιούμε το dataset σε συνεχείς και μοναδικές τιμές, βρίσκουμε το ποσοστό βαθμολόγησης (sparsity), καθώς και τη μέση βαθμολόγηση, η οποία θα μας χρειαστεί περαιτέρω στην διαδικασία της μοντελοποίησης (mean bias). Έπειτα, σπάμε το dataset σε train και test set και στην συνέχεια δημιουργούμε torch datasets από αυτά.

```
#part 2: preparing data

#import data
print(f'\nLoading the data...')
PATH = Path("../Datasets/ml-latest-small")
data = pd.read_csv(PATH/"ratings.csv")

#encode the dataset into unique and contiguous values
data = encode(data)
print(f'Dataset has been encoded')

num_ratings = len(data.rating)
num_users = len(data.userId.unique())
num_items = len(data.movieId.unique())
sparsity = num_ratings/(num_users*num_items)
mean_rating = data.rating.mean()
```

```
print(f'\nNumber of users: {num_users}')
print(f'Number of items: {num_items}')
print(f'Number of ratings: {num_ratings}')
print(f'Dataset sparsity: {sparsity}')
print(f'\nAverage rating: {mean_rating}\n')

#splitting variables input
method, value = input_select()

print(f'\nSplitting:\nMethod: {method}\nValue: {value}')

#dataset splitting
print('\nSplitting data into training and testing set\n')
train_set, test_set = split(data, method, value)

print(f'\nTraining set: {train_set}\n')
print(f'\nTesting set: {test_set}\n')

print(f'\nMaking the torch datasets\n')
train_dataset = CF_Dataset(train_set)
test_dataset = CF_Dataset(test_set)
```

Κώδικας 42 – Μέθοδοι παραγοντοποίησης σε PyTorch, Part 2: preparing the data

```
import numpy as np
import sys
from sklearn.utils import shuffle

#encodes columns of dataframe data into unique values
#and drops all negative records.
#the dataframe data is returned
def encode(data, columns=['userId', 'movieId']):
    for col_name in columns:
        _,col,_ = proc_col(data[col_name])
        data[col_name] = col
        data = data[data[col_name] >= 0]
    return data

#processes a column and returns a dictionary of its indices
#and values, an array of these values and the length of the
#column. the processing of the column involves transforming it
#into unique values before returning any of the aforementioned
#values
def proc_col(col):
```

```
    uniq = col.unique()
    name2idx = {o:i for i,o in enumerate(uniq)}
    return name2idx, np.array([name2idx.get(x,1) for x in col]),\
    len(uniq)
```

Κώδικας 43 – Μέθοδοι παραγοντοποίησης σε PyTorch, μέθοδοι κωδικοποίησης dataset

```
#returns valid splitting method selection and proper values
#for splitting
def input_select():
    print('Choose splitting method:')
    print('1. Testing size')
    print('2. Percentage split')
    print('3. Use default (Percentage split of 0.1)')
    method = input('')
    if method == '1':
        value = input_split()
    elif method == '2':
        value = input_mask()
    elif method == '3':
        method = 2
        value = float(0.1)
    else:
        print('Again...')
        method, value = input_select()
    return int(method), value
```

Κώδικας 44 – Μέθοδοι παραγοντοποίησης σε PyTorch, μέθοδος επιλογής τρόπου διαχωρισμού των δεδομένων σε train και test set

```
#split dataframe into train and test sets
def split(data, method=2, value=None):
    try:
        #determined by testing set size
        if method == 1:
            if value == None:
                test = data.sample(n=1000)
            else:
                test = data.sample(n=value)
        #determined by split percentage
        elif method == 2:
            if value == None:
                test = data.sample(frac=0.1)
            else:
                test = data.sample(frac=value)
```

```
        train = data.drop(test.index)
        return train, test
    except Exception:
        print('Invalid method, incorrect dataset input, \
invalid testing set size or slicing \
percentage. Program will exit to avoid \
further errors.')
        sys.exit()
```

Κώδικας 45 – Μέθοδοι παραγοντοποίησης σε PyTorch, μέθοδος διαχωρισμού των δεδομένων σε train και test set

Part 3: Modelling

Το επόμενο βήμα είναι η δημιουργία του μοντέλου μας. Θα χρησιμοποιήσουμε το SVD module που φτιάξαμε εμείς, το οποίο αποτελείται από τέσσερα embedding layers: δύο embeddings παραγόντων για τα αντικείμενα και τους χρήστες και δύο embeddings πολώσεων για τα αντικείμενα και τους χρήστες αντίστοιχα. [28]

Ένα embedding layer δεν είναι παρά μόνο ένας πίνακας αναζήτησης βαρών. Δίνοντας ένα ευρετήριο tensor σαν είσοδο, παίρνουμε πίσω βάρη για τις θέσεις του ευρετηρίου. Εάν δηλαδή δώσουμε για παράδειγμα indexes χρηστών σαν είσοδο, στο embedding layer που περιέχει τους παράγοντες των χρηστών, θα λάβουμε πίσω τους παράγοντες για τους χρήστες αυτούς. [19]

Με μια τελευταία πινελιά τον μέσο όρο βαθμολόγησης σαν παράμετρο, που δεν εκπαιδεύεται (requires_grad=False), και μια απλή μέθοδο πρόβλεψης (forward), το μοντέλο μας είναι ολοκληρωμένο.

```
model = SVD(num_users, num_items, mean_rating)
#set the model to cuda for gpu training
#model = model.cuda()
print(f'Model initialized: \n{model}')
import torch.nn as nn #neural network module
```

Κώδικας 46 – Μέθοδοι παραγοντοποίησης σε PyTorch, Part 3: modelling

```
#SVD (singular value decomposition) model with bias and mean
#rating
class SVD(nn.Module):
    def __init__(self, num_users, num_items, mean, emb_size=100):
        super(SVD, self).__init__()
        self.user_emb = nn.Embedding(num_users, emb_size)
        self.user_emb_bias = nn.Embedding(num_users, 1)
        self.item_emb = nn.Embedding(num_items, emb_size)
```

```
        self.item_emb_bias = nn.Embedding(num_items, 1)
        self.user_emb.weight.data.uniform_(0, 0.005)
        self.user_emb_bias.weight.data.uniform_(-0.01, 0.01)
        self.item_emb.weight.data.uniform_(0, 0.005)
        self.item_emb_bias.weight.data.uniform_(-0.01, 0.01)
        self.mean = nn.Parameter(torch.FloatTensor([mean]), False)


    def forward(self, u, v):
        U = self.user_emb(u)
        b_u = self.user_emb_bias(u).squeeze()
        I = self.item_emb(v)
        b_i = self.item_emb_bias(v).squeeze()
        return (I*U).sum(1) + b_u + b_i + self.mean
```

Κώδικας 47 – Μέθοδοι παραγοντοποίησης σε PyTorch, μοντέλο αλγορίθμου SVD σε PyTorch

Part 4: Training

Η εκπαίδευση θα γίνει παρόμοια με τον Κώδικα 29 που δείξαμε σε προηγούμενη ενότητα. Θα υλοποιήσουμε έναν RandomSampler σε συνδυασμό με έναν DataLoader με βάση το train_dataset που φτιάξαμε προηγούμενως.

Η εκπαίδευση θα χρονομετρηθεί και στο τέλος της θα υπολογίσουμε τον μέσο όρο των παραγόντων των αντικείμενων και των χρηστών, με βάση τους οποίους θα ενημερώσουμε τυχόν χρήστες ή αντικείμενα που έλειψαν από την διαδικασία της εκπαίδευσης, ήτοι το training set.

```
#part 4: training

start = time.time()
train_model(model, train_dataset, cuda=True)
end = time.time()
print(f'\nTime elapsed for training: {end-start} sec')

#set untrained users/items weights to the average
set_model_untrained_weights(model, train_dataset, test_dataset,
cuda=True)
```

Κώδικας 48 – Μέθοδοι παραγοντοποίησης σε PyTorch, Part 4: training

Για την εκπαίδευση θα χρησιμοποιήσουμε μια ουρά (queue) και ένα μονάχα παράλληλο thread, για την εκτύπωση όλων των μηνυμάτων εκπαίδευσης. Σκοπός αυτής της υλοποίησης είναι η εξάλειψη των διακοπών input/output (I/O interrupts)

που διαφορετικά θα καθυστερούσαν την εξέλιξη του προγράμματος, διαχειρίζοντας τες με ένα διαφορετικό νήμα επεξεργασίας. [30][31]

Υλοποιούμε ακόμη μια επιλογή εκπαίδευσης σε GPU, με μια μεταβλητή boolean, την cuda, η οποία σηματοδοτεί την ανάγκη μεταφοράς των δεδομένων ενός batch στην κάρτα γραφικών μας, για τον ταχύ υπολογισμό πράξεων, με την υποστήριξη της τεχνολογίας CUDA. Προϋπόθεση για την εκπαίδευση τέτοιου είδους είναι η αποθήκευση του μοντέλου μας στην κάρτα γραφικών, κατά την διάρκεια της διαδικασίας της μοντελοποίησης, με την μέθοδο cuda().

Προσοχή πρέπει να δοθεί στον αριθμό των διεργασιών εργατών φόρτωσης του DataLoader (workers), καθώς ακόμη και ένας μικρός αριθμός εργατών, πχ. 4, είναι ικανός να υπερφορτώσει ένα σύστημα με μια μικρή κύρια μνήμη των 4GB, υπό συγκεκριμένες προϋποθέσεις.

```
import torch
import torch.nn.functional as F
from torch.utils.data import RandomSampler, DataLoader
import threading
import queue

from misc import queue_iter_print

#factorization module training. optimizer is Adam
#by default. options for cuda training and ratings dimension
#increase (unsqueeze)
#prints training error per epoch and batch
def train_model(model, train_dataset, epochs=10, lr=0.01,
                wd=0.0, unsqueeze=False, cuda=False):

    #printing is assigned to a new thread to minimize training
    #time by removing I/O interrupts
    #a queue will handle the passing of the print string messages
    #to the new thread
    q = queue.Queue()
    print_thread = threading.Thread(target=queue_iter_print,
args=(q,))
    print_thread.start()

    #make the sampler
    q.put("Creating sampler")
```

```
    sampler = RandomSampler(train_dataset)

    #make the dataloader
    q.put("Making Dataloader")
    dataloader = DataLoader(train_dataset, batch_size=100,
num_workers=4, sampler=sampler)

    #optimizer created after the model has been set to a definite
location
    optimizer = torch.optim.Adam(model.parameters(),lr=lr,
                                 weight_decay=wd)
    q.put('Created optimizer')

    #set the model to training mode (most of the time not
obligatory)
    model.train()
    q.put('Model is in training mode\n')

    #for epochs
    for i in range(epochs):

        #for every batch
        for batch_idx, batch_sample in enumerate(dataloader):

            #training print preamble message
            train_prt = f"Epoch {i}, batch {batch_idx}: "

            #get user and item indices and matching ratings
            #cast users and items to long, ratings to float
            users = batch_sample["userId"].long()
            items = batch_sample["movieId"].long()
            ratings = batch_sample["rating"].float()

            #increase ratings dimension if needed
            if unsqueeze:
                ratings = ratings.unsqueeze(1)

            #set users, items and ratings to cuda if the model is
            #as well for gpu training
            if cuda:
                users = users.cuda()
                items = items.cuda()
```

```
                ratings = ratings.cuda()

            y_hat = model(users, items) #prediction
            loss = F.mse_loss(y_hat, ratings) #loss
            optimizer.zero_grad() #zero gradients
            loss.backward() #update gradients
            optimizer.step() #step
            q.put(f'{train_prt}Training loss: {loss.item()} ')

    q.put(".end") #signal the print function to return
    print_thread.join() #and close the printing thread
```

Κώδικας 49 – Μέθοδοι παραγοντοποίησης σε PyTorch, εκπαίδευση αλγορίθμου SVD με τη χρήση του MovieLens dataset

```
#iteratively prints elements of a queue. end keyword is ".end"
def queue_iter_print(q):

    while True:
        prt = q.get()
        if prt == ".end": break
        print(prt)
```

Κώδικας 50 – Μέθοδοι παραγοντοποίησης σε PyTorch, εκτύπωση μηνυμάτων εκπαίδευσης σε παράλληλο νήμα με τη χρήση multithreading και queue

```
import torch #PyTorch

from data_encode import get_untrained

#sets the model weights for items and users not included
#in the training set (untrained and unchanged weights)
#to the mean item or user weight.
#a classic cold start problem solution
def set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=False):

    #get the untrained user and item indices
    untrained_users, untrained_items = get_untrained(test_dataset,
train_dataset)

    #find the trained users and items indices
    user_trained_indices =
torch.LongTensor(train_dataset.data['userId'].unique())
    item_trained_indices =
torch.LongTensor(train_dataset.data['movieId'].unique())
```

```
    #if model is in cuda set trained user and item indices tensors
    #to cuda as well
    if cuda == True:
        user_trained_indices = user_trained_indices.cuda()
        item_trained_indices = item_trained_indices.cuda()

    #get the trained user and item weights
    user_trained_weights =
torch.index_select(model.user_emb.weight, dim=0, index =
user_trained_indices)
    item_trained_weights =
torch.index_select(model.item_emb.weight, dim=0, index =
item_trained_indices)

    #get the trained users and items mean weights
    user_trained_average = user_trained_weights.mean(0)
    item_trained_average = item_trained_weights.mean(0)

    #for every untrained user and item set the weights to the mean
    for i in untrained_users:
        model.user_emb.weight[i] = user_trained_average

    for i in untrained_items:
        model.item_emb.weight[i] = item_trained_average
```

Κώδικας 51 – Μέθοδοι παραγοντοποίησης σε PyTorch, υπολογισμός μη εκπαιδευμένων βαρών και χρηστών

```
#returns a tuple of indices of extinct users and items of a
#training set when compared to a testing set
def get_untrained(test_dataset, train_dataset):

    #create user and item arrays
    users = np.array([], dtype=int)
    items = np.array([], dtype=int)

    #for each user
    for i in test_dataset.data['userId'].unique():
        #if not belonging to the train set
        if i not in train_dataset.data['userId'].values:
            #append to the user array
            users = np.append(users, i)
```

```
    #for each item
    for i in test_dataset.data['movieId'].unique():
        #if not belonging to the train set
        if i not in train_dataset.data['movieId'].values:
            #append to the item array
            items = np.append(items, i)

    return users, items
```

Κώδικας 52 – Μέθοδοι παραγοντοποίησης σε PyTorch, μέθοδος εύρεσης μη εκπαιδευμένων χρηστών και αντικειμένων

Part 5: Testing

Τέλος, ελέγχουμε την ακρίβεια του μοντέλου μας.

```
#part 5: testing

test_loss(model, test_dataset, cuda=True)
```

Κώδικας 53 – Μέθοδοι παραγοντοποίησης σε PyTorch, Part 5: testing

```
#calculates loss error between the prediction and the testing
#set. default method of calculation is mean squared error
def test_loss(model, test_dataset, unsqueeze=False,
              cuda=False):
    #set model to evaluation mode (most of the time not
obligatory)
    model.eval()

    #get user and item indices and matching ratings
    users = torch.LongTensor(test_dataset.data.userId.values)
    items = torch.LongTensor(test_dataset.data.movieId.values)
    ratings = torch.FloatTensor(test_dataset.data.rating.values)

    #if model is in cuda set test user, item and rating indices
    #tensors to cuda as well
    if cuda == True:
        users = users.cuda()
        items = items.cuda()
        ratings = ratings.cuda()

    #increase ratings dimension if needed
    if unsqueeze:
        ratings = ratings.unsqueeze(1)
    y_hat = model(users, items) #prediction
    loss = F.mse_loss(y_hat, ratings) #test loss
```

```
    print(f'\nTest loss: {loss.item()}')
```

Κώδικας 54 – Μέθοδοι παραγοντοποίησης σε PyTorch, μέθοδος υπολογισμού σφάλματος

3.3 – Κατανεμημένες εφαρμογές σε PyTorch και το distributed communication package

Το torch.distributed είναι ένα πακέτο εμπνευσμένο κατά κύριο λόγο από το MPI, επιτρέποντας την επικοινωνία ανάμεσα σε διαφορετικές διεργασίες. Σε αντίθεση με το πακέτο torch.multiprocessing, το torch.distributed επιτρέπει την επικοινωνία ανάμεσα σε διεργασίες που εδρεύουν σε διαφορετικά μηχανήματα, δεδομένου ότι είναι δυνατή η επικοινωνία των διεργασιών μέσα από δίκτυο. Ακόμη, οι διεργασίες αυτές μπορούν να χρησιμοποιήσουν μια πληθώρα πρωτοκόλλων οπίσθιας επικοινωνίας (backends) σε σχέση με το πακέτο multiprocessing.

Σε αυτή την ενότητα θα παρουσιάσουμε το πακέτο distributed και θα δείξουμε πως μπορεί κανείς να αναπτύξει μια κατανεμημένη εφαρμογή μάθησης με την βοήθεια της βιβλιοθήκης PyTorch.

3.3.1 – Backends

Ξεκινάμε παρουσιάζοντας τα διαθέσιμα πρωτόκολλα οπίσθιας επικοινωνίας (backends) και το τι μπορεί να προσφέρει το καθένα από αυτά. [32][33]

Gloo

Το Gloo αποτελεί το βασικότερο από τα backends και είναι άμεσα διαθέσιμο με την εγκατάσταση της PyTorch, για εκπαίδευση με τη χρήση της κεντρικής επεξεργαστικής μονάδας και της κύριας μνήμης (CPU training), αλλά και με τη χρήση καρτών γραφικών γενικού σκοπού που υποστηρίζουν την τεχνολογία CUDA (GPU training). Το Gloo ενδείκνυται, σαν πρώτη λύση, για χρήση σε CPU training από την PyTorch και σαν δεύτερη λύση, για χρήση σε GPU training, σαν κανόνας.

NCCL

Η NCCL (NVidia Collective Communications Library) είναι μια βιβλιοθήκη κατανεμημένης επεξεργασίας, ανεπτυγμένη από την nVidia, για τις κάρτες γραφικών της και εκτενώς εμπνευσμένη από το πρωτόκολλο MPI. Η NCCL προσφέρεται σαν backend από το πακέτο distributed, για εκπαίδευση μόνο με την χρήση καρτών γραφικών γενικού σκοπού nVidia. Η NCCL ενδείκνυται για χρήση σε GPU training από την PyTorch, σαν κανόνας.

MPI

Το πρωτόκολλο MPI είναι ένα τυποποιημένο πρωτόκολλο για κατανεμημένη επικοινωνία σε ένα μεγάλο εύρος παράλληλων υπολογιστικών αρχιτεκτονικών. Πολλές εκδοχές του MPI υπάρχουν, αρκετές από τις οποίες είναι ανοιχτού λογισμικού.

Εάν θέλουμε να χρησιμοποιήσουμε το πρωτόκολλο MPI στις κατανεμημένες εφαρμογές μας, θα πρέπει να εγκαταστήσουμε την PyTorch from source. Το MPI είναι το μόνο πρωτόκολλο που δεν προσφέρεται έτοιμο με την εγκατάσταση της PyTorch στο σύστημα μας, ενώ για την εκπαίδευση CPU προτιμάται το Gloo, εκτός και αν συγκεκριμένοι λόγοι υπάρχουν για την χρήση MPI. Αναλόγως της υλοποίησης, το MPI backend μπορεί ακόμη να υποστηρίξει εκπαίδευση GPU. [33]

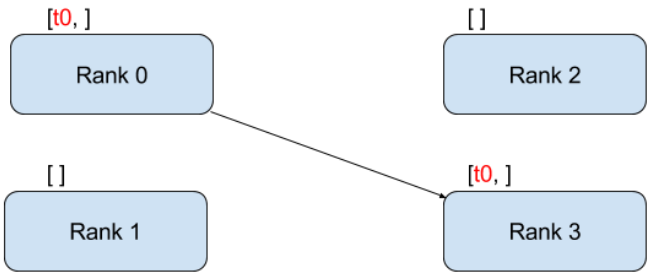
TCP (deprecated)

Ένα ξεπερασμένο πλέον backend, το TCP, αποτελεί μια εναλλακτική λύση κατανεμημένης επικοινωνίας. Το TCP είναι το βασικό πρωτόκολλο επιπέδου μεταφοράς στη δικτύωση υπολογιστών.

3.3.2 – Τύποι λειτουργιών κατανεμημένης επικοινωνίας

Point-to-point

Υπάρχουν διάφοροι τρόποι για την επικοινωνία μεταξύ ενός αριθμού διεργασιών. Ο απλούστερος από αυτούς αφορά την επικοινωνία μεταξύ δύο διεργασιών και είναι η απλή αποστολή (send) και λήψη (receive) δεδομένων, ή αλλιώς, επικοινωνία σημείο προς σημείο (point-to-point).



Εικόνα 3 – PyTorch distributed package, Point-to-point communication

Η διαδικασία αποστολής και λήψης μπορεί να είναι σύγχρονη (blocking), οπού η κάθε διεργασία περιμένει να παραληφθεί ή να παραλάβει ένα μήνυμα, ή ασύγχρονη (non-blocking), κατά την οποία ένα μήνυμα μπορεί να σταλεί χωρίς να χρειαστεί οι διεργασίες να περιμένουν για την παραλαβή. Για την δεύτερη περίπτωση, ένα

γεγονός αναμονής στην κατάλληλη θέση είναι αρκετό για τον έλεγχο της λήψης του μηνύματος. [32][33]

Οι blocking λειτουργίες send και recv και non-blocking isend και ircv είναι υλοποιήσιμες με την χρήση των Gloo και MPI backends για εκπαίδευση σε CPU.

```
import torch #PyTorch
import torch.distributed as dist #distributed

#Send/Receive
#send/recv not supported by GPU!
def send_recv(rank, size):
    print(f'Hello from task no. {rank}!')
    tensor = torch.zeros(1) #tensor([0.])
    #master process, rank = 0
    if rank == 0:
        print('I increase the tensor value by 1\
 and send it to process 1!')
        tensor += 1 #tensor([1.])
        #send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    #slave process, rank = 1
    else:
        print('And I receive it!')
        #receive the tensor sent by process 0
        dist.recv(tensor=tensor, src=0)
    print(f'Rank {rank}: My tensor has this data: {tensor[0]}')

#isend/irecv (asynchronous)
#send/recv not supported by GPU!
def isend_irecv(rank, size):
    print(f'Hello from task no. {rank}!')
    tensor = torch.zeros(1) ) #tensor([0.])
    req = None #waitable object
    #master process, rank = 0
    if rank == 0:
        print('I increase the tensor value by 1\
 and send it to process 1!')
        tensor += 1 #tensor([1.])
        #send the tensor to process 1
        req = dist.isend(tensor=tensor, dst=1)
        print('Task 0 started sending')
```

```
    #slave process, rank = 1
    else:
        print('And I receive it!')
        #receive the tensor sent by process 0
        req = dist.irecv(tensor=tensor, src=0)
        print('Task 1 started receiving')
    #wait to receive/send data
    req.wait()
    print(f'Tasks synchonized!')
    print(f'Rank {rank}: My tensor has this data: {tensor[0]}')
```

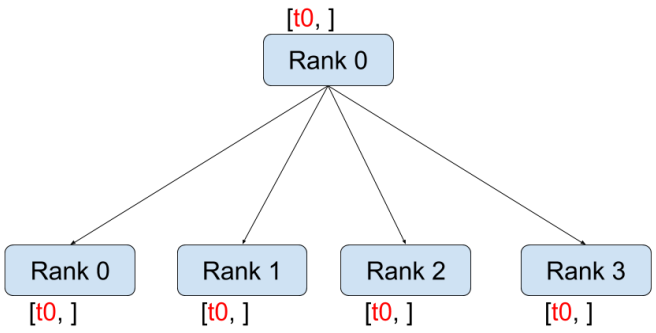
Κώδικας 55 – PyTorch distributed package, υλοποίηση κατανεμημένων λειτουργιών send/recv

Multipoint

Πέρα από την επικοινωνία δύο μόνο διεργασιών, ταυτόχρονη επικοινωνία μπορεί να υπάρξει και μεταξύ περισσοτέρων από δύο διεργασίες. Ονομάζουμε αυτού του είδους την επικοινωνία multipoint, καθώς αφορά τον συνεργατικό συγχρονισμό πολλαπλών σημείων. [32][33]

Broadcast

Η απλούστερη από τις multipoint λειτουργίες κατανεμημένης επικοινωνίας, εκπέμπει έναν tensor από μια διεργασία σε όλες τις υπόλοιπες, μέσα σε ένα group που περιλαμβάνει όλες τις διεργασίες που υπάρχουν στον κόσμο (world) των διεργασιών που λαμβάνουν μέρος στην κατανεμημένη επικοινωνία. Όλες οι διεργασίες, πλην της αποστέλουσας διεργασίας, λαμβάνουν τον tensor που η διεργασία εξέπεμψε. [32][33]



Εικόνα 4 – PyTorch distributed package, Multipoint communication, Broadcast

Να τονισθεί πως το group πρέπει να περιλαμβάνει όλες τις διεργασίες που υπάρχουν μέσα στον εκάστοτε κόσμο διεργασιών και ότι η δημιουργία του group που περιέχει τις διεργασίες πρέπει να γίνεται με την ίδια σειρά σε κάθε διεργασία.

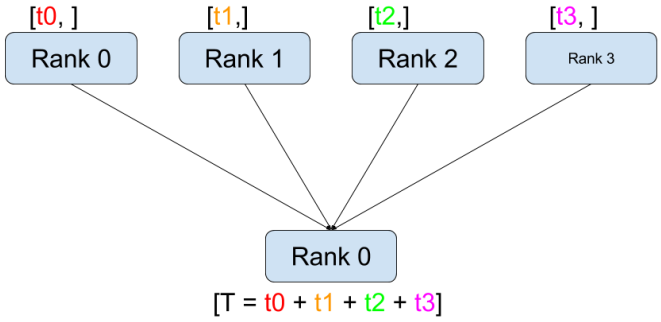
Η λειτουργία broadcast υποστηρίζεται από όλα τα backends κατανεμημένης επικοινωνίας. [32][33]

```
#supported by every backend
def broadcast(rank, size):
    print('Hello from task ', rank, '!')
    #create a new group with the following
    #processes
    #Notice: all processes belonging in the
    #distributed job MUST be entered, even if
    #they are not to be used.
    #Groups should also be created in the same
    #order in all processes
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #everybody in the group stores the received tensor inside the
    #tensor
    #Notice: the tensor should be of the SAME DIMENSIONS for each
    #process
    dist.broadcast(tensor=tensor, src=0, group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])
```

Κώδικας 56 – PyTorch distributed package, υλοποίηση κατανεμημένης λειτουργίας broadcast

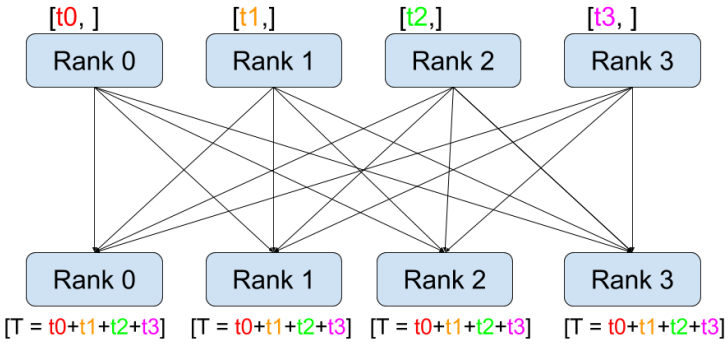
Reduce και all-reduce

Μια αντιστρόφως ανάλογη λειτουργία της broadcast, η reduce, συνοψίζει όλους τους tensors των διεργασιών σε ένα group, σε μια διεργασία. Η σύνοψη μπορεί να γίνει αθροίζοντας τους tensors (sum - συχνότερο), βρίσκοντας το εσωτερικό γινόμενό τους (dot product), βρίσκοντας τον μέγιστο tensor (max) ή βρίσκοντας τον ελάχιστο (min). [32][33]



Εικόνα 5 – PyTorch distributed package, Multipoint communication, Reduce with sum

Μια παραλλαγή της reduce, η all-reduce, υλοποιεί αυτή την λειτουργία για κάθε διεργασία που συμμετέχει στο group.



Εικόνα 6 – PyTorch distributed package, Multipoint communication, All-reduce with sum

Η reduce υποστηρίζεται από κάθε backend, πλην του Gloo για εκπαίδευση GPU, ενώ η all-reduce υποστηρίζεται από κάθε backend.

Να σημειωθεί πως το NCCL backend μπορεί να χρησιμοποιηθεί μόνο για επικοινωνία σε εκπαίδευση GPU, ήτοι οι tensors πρέπει να είναι αποθηκευμένοι στην μνήμη της κάρτας γραφικών (tensor.cuda()).

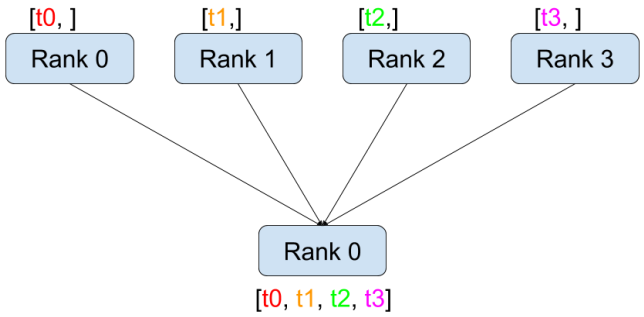
```
#supported by every backend, minus GPU Gloo
def reduce(rank, size):
    print('Hello from task ', rank, '!')
    #create a new group with the following
    #processes
    #Notice: all processes belonging in the
    #distributed job MUST be entered, even if
    #they are not to be used.
    #Groups should also be created in the same
    #order in all processes
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #rank 0 does a reduce operation,
    #summing up the tensors received
    #by everyone else
    dist.reduce(tensor=tensor, dst=0, op=dist.reduce_op.SUM,
group=group)
```

```
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])

#supported by every backend
def all_reduce(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #everybody in the group does an all reduce
    #operation summing up the tensors received
    #by everyone else
    dist.all_reduce(tensor=tensor, op=dist.reduce_op.SUM,
group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])
```

Κώδικας 57 – PyTorch distributed package, υλοποίηση κατανεμημένων λειτουργιών reduce και all-reduce

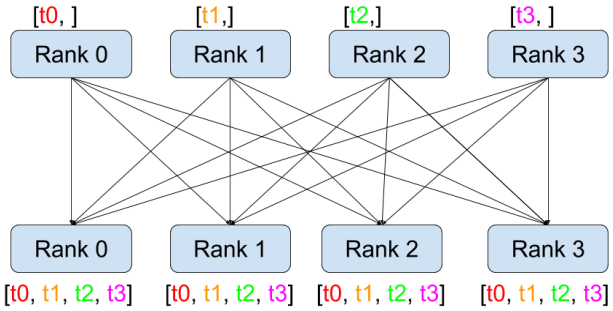
Gather, all-gather

Μια ακόμη αντιστρόφως ανάλογη λειτουργία της broadcast, η gather, συγκεντρώνει όλους τους tensors, από όλες τις διεργασίες, στο group, σε μια διεργασία. [32][33]



Εικόνα 7 – PyTorch distributed package, Multipoint communication, Gather

Μια παραλλαγή της gather, η all-gather, υλοποιεί αυτή την λειτουργία για κάθε διεργασία που συμμετέχει στο group.



Εικόνα 8 – PyTorch distributed package, Multipoint communication, All-gather

Η gather υποστηρίζεται από τα Gloo και MPI backends, για εκπαίδευση CPU, ενώ η all-gather υποστηρίζεται από το Gloo, για εκπαίδευση CPU, το MPI, για εκπαίδευση CPU ή και GPU, και το NCCL, για εκπαίδευση GPU.

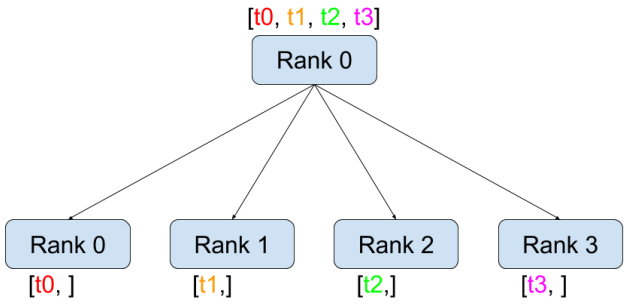
```
#CPU only
def gather(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #rank 0 does a gather operation,
    #storing the tensors received
    #by everyone else inside the tensor list
    if rank == 0:
        #the tensor list should contain correctly sized
        #tensors, matching in dimensions every tensor
        #that is to be received by the processes
        tensor_list = [tensor, tensor]
        dist.gather(tensor=tensor, gather_list=tensor_list,
group=group)
    else:
        dist.gather(tensor=tensor, dst=0, group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])

#supported by every backend, minus GPU Gloo
def all_gather(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
```

```
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #the tensor list should contain correctly sized
    #tensors, matching in dimensions every tensor
    #that is to be received by the processes
    tensor_list = [tensor, tensor]
    #everybody in the group does a gather operation,
    #storing the tensors received
    #by everyone else inside the tensor list
    dist.all_gather(tensor_list=tensor_list, tensor=tensor,
group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])
```

Κώδικας 58 – PyTorch distributed package, υλοποίηση κατανεμημένων λειτουργιών gather και all-gather

Scatter

Η λειτουργία scatter διαμοιράζει μια λίστα από tensors σε όλες τις διεργασίες σε ένα group. [32][33]



Εικόνα 9 – PyTorch distributed package, Multipoint communication, Scatter

Η scatter υποστηρίζεται μόνο για εκπαίδευση CPU, ήτοι από τα Gloo και MPI backends.

```
#CPU only
def scatter(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #rank 0 does a scatter operation,
    #distributing the tensors inside the list
    #to every process inside the group
```

```
    if rank == 0:
        tensor_list = [tensor, tensor]
        dist.scatter(tensor=tensor, scatter_list=tensor_list,
group=group)
    else:
        dist.scatter(tensor=tensor, src=0, group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])
```

Κώδικας 59 – PyTorch distributed package, υλοποίηση κατανεμημένης λειτουργίας scatter

Barrier

Τέλος η λειτουργία barrier συγχρονίζει όλες τις διεργασίες, αναγκάζοντας τες τις να περιμένουν τις υπόλοιπες να φτάσουν στην γραμμή που ορίζεται η λειτουργία barrier. [32][33]

Η λειτουργία barrier υποστηρίζεται από κάθε backend, πλην του Gloo, για εκπαίδευση σε GPU.

```
#supported by every backend, minus GPU Gloo
def barrier(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    tensor += 1
    #wait all processes to reach the following line
    dist.barrier(group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])
```

Κώδικας 60 – PyTorch distributed package, υλοποίηση κατανεμημένης λειτουργίας barrier

Αξίζει να σημειωθεί πως κάθε μια από τις παραπάνω multipoint λειτουργίες κατανεμημένης επικοινωνίας μπορεί να είναι ασύγχρονη. Μια ακόμη παράμετρος, η async_op μπορεί να τεθεί ως True για αυτό το λόγο. Σε αυτή την περίπτωση, η κάθε λειτουργία επιστρέφει ένα αντικείμενο, πχ. req, το οποίο θα πρέπει να περιμένουμε με την μέθοδο req.wait(), με τον τρόπο που παρουσιάστηκε στον Κώδικα 55 για την ασύγχρονη μέθοδο isend_irecv. [32][33]

3.3.3 – Διαδικασία δημιουργίας ομάδων επικοινωνίας

Παραπάνω αναλύσαμε τις λειτουργίες κατανεμημένης επικοινωνίας, φτιάχνοντας για κάθε μία μια μέθοδο υλοποίησης. Πως όμως οι μέθοδοι αυτοί μπορούν να κληθούν και να υλοποιηθούν από τις διεργασίες; Και πως οι διεργασίες αυτές δημιουργούνται;

Σε αυτή την υποενότητα θα παρουσιάσουμε το πως μπορεί κανείς να εκκινήσει μια διεργασία.

Κάθε διεργασία εδρεύει σε μια περιοχή. Η κάθε διεργασία είναι προσβάσιμη από τις υπόλοιπες μέσω μια διεύθυνσης IP και μιας θύρας (port). Αυτό βγάζει νόημα, καθώς σε μια κατανεμημένη εφαρμογή PyTorch, μια διεργασία μπορεί να βρίσκεται οπουδήποτε μέσα σε ένα δίκτυο, με τις διεργασίες να επικοινωνούν πιθανότατα μέσω δικτύου. Κάθε διεργασία έχει ακόμη ένα αναγνωριστικό αριθμό κατάταξης (rank) και συμμετέχει σε ένα κόσμο διεργασιών ενός τάδε αριθμού (world size).

Στις κατανεμημένες εφαρμογές PyTorch, ο συγχρονισμός όλων των υπόλοιπων διεργασιών γίνεται μέσω της διεργασία master, η οποία έχει αναγνωριστικό rank = 0. Για αυτό το λόγο, γνωρίζοντας μόνο τον τρόπο πρόσβασης της τοποθεσίας της διεργασίας master, οι διεργασίες μπορούν να συγχρονιστούν μέσω αυτής. [33]

Σε ένα κόσμο διεργασιών όπου όλες οι διεργασίες εδρεύουν στην ίδια μηχανή, η διεύθυνση IP της διεργασίας master είναι αυτή του τοπικού μηχανήματος, 127.0.0.1 (localhost). Ο αριθμός port προτιμάται να είναι 29500. Η κάθε διεργασία χρειάζεται ακόμη να γνωρίζει τον αναγνωριστικό αριθμό rank της καθώς και το μέγεθος του κόσμου.

Οι τέσσερις αυτές μεταβλητές αποθηκεύονται στο περιβάλλον του λειτουργικού συστήματος, για την κάθε διεργασία.

Οι μεταβλητές αυτές μπορούν να αποθηκευτούν, για την κάθε διεργασία, καλώντας την εντολή `export <variable_name> = '<variable_string_value>'` πριν την κλήση του προγράμματος, ή με την χρήση της βιβλιοθήκης os της python.

Γνωρίζοντας αυτές τις παραμέτρους, το επόμενο βήμα είναι η ένταξη της διεργασίας μέσα στον κόσμο των διεργασιών με τη χρήση της μεθόδου init_process_group. Αφού γίνει αυτό, η διεργασία μας μπορεί να εκτελέσει

λειτουργίες κατανεμημένης επικοινωνίας για την επικοινωνία με οποιαδήποτε άλλη διεργασία στον κόσμο.

```
import torch #PyTorch
import torch.distributed as dist #distributed
import os #operating system

#process initialization method
#process rank, world size, function to run, additional arguments,
#communication backend to use (default is 'gloo') and additional
#keyworded arguments
def init_processes(rank, size, fn, *args, backend='gloo',
**kargs):
    #master rank address (localhost)
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    #master rank port
    os.environ['MASTER_PORT'] = '29500'
    #world size
    os.environ['WORLD_SIZE'] = str(size)
    #process rank identifier
    os.environ['RANK'] = str(rank)

    #initialize the process
    dist.init_process_group(backend, world_size=size, rank=rank)
    #run the preferred function
    fn(*args, rank=rank, size=size, **kargs)

#process rank 0
#rank, world size, function to run, backend
init_processes(0, 2, all_reduce, backend='nccl')

#process rank 1
#rank, world size, function to run, backend
init_processes(1, 2, all_reduce, backend='nccl')
```

Κώδικας 61 – PyTorch distributed package, δημιουργία ομάδων επικοινωνίας

3.3.4 – Η βιβλιοθήκη multiprocessing της Python

Process (διεργασία)

Οι διεργασίες μας μπορεί συχνά να προέρχονται από διαφορετικά python scripts. Τι γίνεται όμως εάν επιθυμούμε να δημιουργήσουμε πολλαπλές διεργασίες με τη χρήση ενός μόνο προγράμματος; Η απάντηση βρίσκεται στο πακέτο

multiprocessing της βιβλιοθήκης PyTorch, το οποίο είναι επέκταση της βιβλιοθήκης multiprocessing της python.

Με τη χρήση του πακέτου multiprocessing μπορούμε να δημιουργήσουμε ένα σετ διεργασιών με μια μέθοδο στόχο που θα εκτελεστεί από την διεργασία, με παραμέτρους που θα ορίσουμε. Κάθε διεργασία εκκινείται με την μέθοδο start() και συγχωνεύεται στο τέλος της, πίσω στην διεργασία που την κάλεσε, με την μέθοδο join(). Μια καλή πρακτική είναι η επισύναψη της κάθε διεργασίας σε μια λίστα διεργασιών, οι οποίες με την σειρά θα συγχωνευτούν πίσω στην κύρια διεργασία. [34][35]

Start methods (μέθοδοι εκκίνησης) και προστασία μνήμης

Καλό είναι ακόμη, για όσα δεδομένα δεν επιθυμούμε να δημιουργηθούν και πάλι στην μνήμη από τις νέες διεργασίες, να τα προστατέψουμε, εντάσσοντας τα σε μια ενότητα if. Ο λόγος είναι πως κατά την δημιουργία μιας νέας διεργασίας, υπάρχει η περίπτωση το πρόγραμμα μας να τρέξει εκ νέου. Σε αυτή την περίπτωση το πρόγραμμα της κύριας διεργασίας θα έχει όνομα (__name__) '__main__', ενώ το πρόγραμμα της νέας θα έχει όνομα ίδιο με το όνομα του python module, πχ. 'do'. [34][35]

Αυτό συμβαίνει όταν η μέθοδος εκκίνησης διεργασίας δεν είναι η προεπιλεγμένη μέθοδος εκκίνησης fork.

Για την δημιουργία μιας νέας διεργασίας δίνονται τρεις τρόποι εκκίνησης [35]:

- fork
- forkserver
- spawn

Fork

Στην περίπτωση της fork, το πρόγραμμα θα φτιάξει μια ‘διχάλα’ της κεντρικής διεργασίας, όπου η διεργασία παιδί θα είναι ένα αντίγραφο της ήδη υπάρχουσας διεργασίας γονέα. Αξίζει να σημειωθεί πως η fork είναι ο προεπιλεγμένος τρόπος δημιουργίας διεργασιών σε συστήματα τύπου Unix.

Στην περίπτωση που ο τρόπος εκκίνησης διεργασίας δεν είναι fork, αλλά forkserver ή spawn (προεπιλεγμένος τρόπος συστημάτων Windows), μια νέα διεργασία θα εκκινηθεί από την αρχή, με βάση το python module.

Forkserver

Στην περίπτωση της forkserver, μια διεργασία server θα δημιουργηθεί και κάθε φορά που μια νέα διεργασία απαιτείται από τον γονέα, η διεργασία server θα φτιάχνει μια νέα διεργασία με τη μέθοδο fork, η οποία θα είναι ενός νήματος (single-threaded).

Spawn

Στην περίπτωση της spawn, ένας νέος διερμηνέας (interpreter) python καλείται και το module τρέχει ξανά από την αρχή, κληρονομώντας μόνο όσους πόρους χρειάζονται για την εκτέλεση της μεθόδου στόχου.

Αξίζει ακόμη να σημειωθεί πως καλώντας την μέθοδο spawn ή forkserver μέσα από μια μέθοδο, μέσα στην οποία μέθοδο καλούμε άλλες μεθόδους που δεν βρίσκονται μέσα σε αυτή, η κλήση των άλλων μεθόδων θα αποτύχει, αφού ο γονέας (αρχική μέθοδος) δεν τις περιλαμβάνει!

Shared memory

Εάν θελήσουμε να μοιραστούμε με τις νέες διεργασίες κάποιους tensors ή το μοντέλο μας, η μέθοδος fork μπορεί να αποδειχτεί προβληματική.

Έστω πως έχουμε ένα αντικείμενο tensor ή μοντέλου αποθηκευμένο στην κύρια μνήμη. Θέλοντας οι νέες διεργασίες που δημιουργήσαμε με την μέθοδο fork να έχουν πρόσβαση σε αυτό, θα πρέπει για το μοντέλο ή τον tensor να καλέσουμε την μέθοδο share_memory(). Η μέθοδος αυτή θα τοποθετήσει το αντικείμενο σε μια κοινόχρηστη μνήμη, όπου όλες οι διεργασίες θα έχουν πρόσβαση. [34]

Να σημειωθεί πως πάρα το γεγονός ότι η μνήμη της κάρτας γραφικών θεωρείται κοινόχρηστη μνήμη (.cuda()), δεν είναι αναγνώσιμη από τις διεργασίες που εκκινήθηκαν με την μέθοδο fork. Για αυτό το λόγο όταν θέλουμε να μοιραστούμε ένα αντικείμενο που βρίσκεται στην μνήμη της κάρτας γραφικών μας, θα πρέπει να χρησιμοποιήσουμε τις μεθόδους spawn ή forkserver. Καλό είναι λοιπόν, να γράφουμε πάντοτε ό,τι δεν θέλουμε να υλοποιηθεί στις νέες διεργασίες, κάτω από μια ενότητα if, ώστε να καλύψουμε πιθανές περιπτώσεις που θα πρέπει να υλοποιηθούν με τις μεθόδους forkserver ή spawn.

Queues

Εάν επιθυμούμε να μεταφέρουμε δεδομένα ανάμεσα σε διεργασίες, πχ. έναν tensor που περιέχει gradients, κατά την εκπαίδευση ενός μοντέλου, μια λύση είναι η χρήση της δομής queue του πακέτου multiprocessing.

Σημαντικό είναι να θυμόμαστε πως τοποθετώντας ένα αντικείμενο tensor ή model σε ένα queue, συνεπάγεται μεταφορά του αντικειμένου σε κοινή μνήμη πρώτα, εάν το αντικείμενο δεν είναι ήδη, πράγμα που μπορεί να προσθέσει επιπλέον overhead. Ας υπενθυμίσουμε ακόμη πως η μνήμη της κάρτας γραφικών θεωρείται κοινόχρηστη μνήμη. [36]

Κατά την μεταφορά, ένα SimpleQueue εγγυάται επιτυχία κάθε φορά, αφού χρησιμοποιεί μόνο ένα νήμα επεξεργασίας, ενώ ένα Queue είναι λιγότερο ασφαλές, αφού πρόκειται για μια πιο σύνθετη κλάση που χρησιμοποιεί πολλαπλά νήματα. [36]

```
import torch.multiprocessing as mp #multiprocessing

#spawn and forkserver methods don't work inside functions...
#def do():
#it is a best practice to use the if __name__=='main' section.
#start methods like spawn and forkserver will produce a copy
#of the process's parent resulting in continuous process
#creation and a runtime error. using the if section ensures
#that no process creation will occur any further, since the
#__name__ variable's value will not be 'main' in the created
#process but a script name instead
if __name__ == "__main__":
    #__name__ will refer to the function and not the script
    #if set within a function.
    #the functions called below will not belong to the
    #main function therefore initialization will fail with
    #spawn and forkserver methods.
    #also, despite expected outcome, fork method (default)
    #will work with cuda tensors using distributed operations!!
    #if there is no specific worry about sharing tensors
    #otherwise with multiprocessing, it seems obvious that
    #any start method will produce similar results with the
    #distributed package operations.
    #furthermore, it is a best practice to set the start
    #method within the if __name__=='main' section.
```

```
    #the start method can also be set only *once* inside
    #a **main** script!!!
    #mp.set_start_method('fork') #default
    #or
    #mp.set_start_method('spawn')
    #or
    #mp.set_start_method('forkserver')
    #otherwise, you can create a context object using a
    #specified start method
    #ctx = mp.get_context('spawn')
    size = 2 #world size
    processes = []
    for rank in range(size):
        print('Initializing task ', rank)
        #p = ctx.Process(target=init_processes, args=(rank, size,
all_reduce))
        p = mp.Process(target=init_processes, args=(rank, size,
all_reduce))
        print('Run process of task ', rank)
        p.start()
        processes.append(p)

    print('\nActive processes before joining: ',
mp.active_children())
    print('Joining ', len(mp.active_children()) ,' processes')
    for p in processes:
        p.join()

    print('Active processes after joining: ',
mp.active_children()) #should be 0
```

Κώδικας 62 – PyTorch multiprocessing package, δημιουργία διεργασιών

3.3.5 – Παράλληλη και παράλληλη κατανεμημένη εκπαίδευση

Σε αυτή την υποενότητα θα δείξουμε πως ένα μοντέλο μπορεί να εκπαιδευτεί παράλληλα αλλά και με τη χρήση κατανεμημένων διεργασιών. [19]

DataParallel

Το πρώτο από τα modules που θα δούμε είναι το torch.nn.DataParallel. Έστω πως έχουμε στα χέρια μας ένα μοντέλο σε ένα μηχάνημα με τρεις κάρτες γραφικών γενικού σκοπού NVidia και πως θέλουμε να αξιοποιήσουμε και τις τρεις για την εκπαίδευση του μοντέλου μας. Ένας απλός τρόπος να τις αξιοποιήσουμε κατά την εκπαίδευση είναι η ενθυλάκωση του μοντέλου μας σε ένα DataParallel module.

Ενθυλακώνοντας το μοντέλο μας, το module θα δημιουργήσει αντίγραφα του μοντέλου σε όλες τις κάρτες γραφικών. Σε κάθε πρόβλεψη που επιθυμούμε να κάνουμε με βάση το μοντέλο μας, ήτοι σε κάθε forward pass, το input θα διαχωρίζεται σε κάθε αντίγραφο της κάθε κάρτας γραφικών, με το κάθε αντίγραφο να δέχεται ένα μέρος του input. Κατά τον υπολογισμό των gradients, ήτοι σε κάθε backward pass, οι gradients θα αθροίζονται πίσω στο αρχικό μοντέλο. [19]

Αξίζει να σημειωθεί πως έχοντας στα χέρια μας αρκετές κάρτες γραφικών στο ίδιο μηχάνημα, μπορούμε να ορίσουμε επίσης ποιες από αυτές θέλουμε να αξιοποιήσουμε (παράμετρος device_ids) και σε ποια θα θέλαμε να γίνεται το άθροισμα κατά την φάση backward (παράμετρος output_device). Επίσης, αν χρησιμοποιούμε DataLoader, το βέλτιστο θα είναι το μέγεθος του batch να είναι πολλαπλάσιο των καρτών γραφικών, για την ομοιόμορφη κατανομή του input ανά συσκευή. [19]

DistributedDataParallel

Το module torch.nn.parallel.DistributedDataParallel είναι το κύριο module που θα χρησιμοποιήσουμε κατά την παράλληλη κατανεμημένη εκπαίδευση. Παρόμοια με το DataParallel, το DistributedDataParallel διαχωρίζει το input στις τοπικές συσκευές που επιθυμούμε.

Ένα προσόν του όμως σε σχέση με το DataParallel, είναι η ικανότητα συγχρονισμού μεταξύ αντιγράφων ενός μοντέλου που ανήκουν σε διαφορετικές διεργασίες για την συνεργατική εκπαίδευση του κοινού αυτού μοντέλου.

Έστω πως έχουμε έναν αριθμό διεργασιών, με κάθε μια από τις διεργασίες να έχει ένα πανομοιότυπο αντίγραφο του μοντέλου και ένα πανομοιότυπο αντίγραφο του dataset, και πως επιθυμούμε να εκπαιδεύσουμε το μοντέλο αυτό με κατανεμημένο τρόπο. Ενθυλακώνοντας το μοντέλο σε ένα DistributedDataParallel module, εξασφαλίζουμε πως σε κάθε backward pass, οι gradients του DistributedDataParallel module ενός μοντέλου, σε κάθε διεργασία, υπολογίζονται στον μέσο όρο μεταξύ των διεργασιών. [19]

Πρακτικά:

- Ένα μοντέλο, ίσος αριθμός αντιγράφων με τον αριθμό των διεργασιών ,για κάθε διεργασία

- Το μοντέλο ενθυλακώνεται σε ένα DistributedDataParallel module και δημιουργούνται αντίγραφα στις συσκευές (κάρτες γραφικών) που ορίζουμε σε κάθε διεργασία (παράμετρος devide_ids)
- Forward pass: κάθε αντίγραφο, σε κάθε συσκευή μιας διεργασίας δέχεται ένα κλάσμα του input
- Backward pass:
    - Για κάθε διεργασία: sum gradients στο αρχικό module (module του output_device)
    - Υπολογισμός του μέσου όρου των gradients μεταξύ διαφορετικών αρχικών module / διεργασιών

Υπάρχουν δύο εξειδικευμένοι τρόποι χρήσης του DistributedDataParallel:

- Single-process multi-GPU: Τρόπος παρόμοιος με το DataParallel. Μια διεργασία ανά μηχάνημα με πολλαπλές GPU. Η παράμετρος device_ids δέχεται μια λίστα από ID συσκευών (0,1,2,…) που θέλουμε να χρησιμοποιήσουμε.
- Multi-process single-GPU: Αριθμός διεργασιών ανά μηχάνημα = αριθμός συσκευών (καρτών γραφικών) του μηχανήματος. Με αυτό τον τρόπο κάθε διεργασία χρησιμοποιεί μόνο μια από τις διαθέσιμες κάρτες γραφικών. Στην βιβλιογραφία αναφέρεται ως η γρηγορότερη προσέγγιση εκπαίδευσης, ακόμη και σε σχέση με το DataParallel module.

Να σημειωθεί πως για την χρήση του DistributedDataParallel module απαιτείται πρώτα η δημιουργία ομάδων επικοινωνίας, με τον τρόπο που παρουσιάστηκε στον Κώδικα 61. Όπως και για το DataParallel, αν χρησιμοποιούμε DataLoader, το βέλτιστο θα είναι το μέγεθος του batch να είναι πολλαπλάσιο των καρτών γραφικών που χρησιμοποιείται σε κάθε διεργασία, για την ομοιόμορφη κατανομή του input ανά συσκευή.

Το DistributedDataParallel module δουλεύει μόνο με τα NCCL και Gloo backends, για εκπαίδευση σε GPU.

Παραθέτουμε ένα παράδειγμα κώδικα μιας μεθόδου, η οποία υλοποιεί πρακτικά την λειτουργία του ορισμού των gradients στον μέσο όρο, με παρόμοιο τρόπο που την κάνει το DistributedDataParallel module. [32]

```
#model parameter averaging
def average(model):
    size = float(dist.get_world_size())
        for param in model.parameters():
        #skip parameters that are not learnable
        if param.grad is not None:
            dist.all_reduce(param.grad.data,op=dist.reduce_op.SUM)
            param.grad.data /= size
```

Κώδικας 63 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, λειτουργία DistributedDataParallel

DistributedDataParallelCPU

Παρόμοιο module με το DistributedDataParallel, μόνο για εκπαίδευση στην κύρια μνήμη. Δεν υπάρχουν παράμετροι device_ids και output_device.

Δουλεύει μόνο με τα MPI και Gloo backends, για εκπαίδευση σε CPU.

DistributedSampler

Ο DistributedSampler είναι ένας Sampler που κατανέμει τα δεδομένα αντιγράφων ενός κοινού μεταξύ διεργασιών Dataset, στις διεργασίες, με τέτοιο τρόπο έτσι ώστε οι διεργασίες να εκπαιδεύουν όλες με διαφορετικά δεδομένα. [21]

Απαίτηση όπως και για τα παραπάνω, είναι η δημιουργία ομάδων επικοινωνίας, με τον τρόπο που παρουσιάστηκε στον Κώδικα 61.

Ένας DataLoader μπορεί με τη σειρά του να δεχθεί ως Sampler τον DistributedSampler, δημιουργώντας batches μόνο για τα δεδομένα της εκάστοτε διεργασίας. Ο DistributedSampler μπορεί να δεχθεί ακόμη σαν seed, για το ανακάτεμα των δεδομένων, τον αριθμό της εκάστοτε εποχής της εκπαίδευσης.

Παρακάτω παρουσιάζουμε μια μέθοδο παράλληλης και παράλληλης κατανεμημένης εκπαίδευσης, εμπνευσμένη από τον Κώδικα 29, ως επέκταση του.

```
#training.py

import torch.nn as nn #neural network
import torch.nn.functional as F #functions
import torch.optim as optim #optimizer
import torch.utils.data as data #utils data
import torch.nn.parallel as parallel #parallel training
```

```
#the following method works both for simple and distributed
#training and is modified to support both GPU (.cuda()) and CPU
#(.cpu()) models.
#parameters include:
#model, training dataset, number of epochs, initial learning
#rate, weight decay, unsqueeze - target dimensionality increase,
#cuda - GPU training mode, distributed mode –
#case where a process belonging to a distributed process world
#calls the training method for distributed training,
#rank – rank of the distributed process,
#world_size – size of the distributed processes world
def train_model(model, train_dataset, epochs=10, lr=0.01,
                wd=0.0, unsqueeze=False, cuda=False,
                distributed_mode=False, rank=None,
                world_size=None):

     sampler = None

     #distributed mode
     if distributed_mode:
         sampler =
         data.distributed.DistributedSampler(train_dataset,
         num_replicas=world_size, rank=rank)

         if cuda:
             model = parallel.DistributedDataParallel(model)
         else:
             model = parallel.DistributedDataParallelCPU(model)

         #non-distributed. set the model to DataParallel to
         #increase training speed
         elif not distributed_mode and cuda:
             model = nn.DataParallel(model)

     dataloader = data.DataLoader(train_dataset, batch_size=100,
     num_workers=2, sampler=sampler)

     #Adam optimizer, with small weight decay
     optimizer = optim.Adam(model.parameters(), lr=learning_rate,
     wd=0.01)

     #set the model in training mode (most of the time not
```

```
    #obligatory)
    model.train()

    for epoch in range(epochs):

        if distributed_mode:
              sampler.set_epoch(i)

        for batch_idx, batch_sample in enumerate(dataloader):

            y_hat = model(batch_sample) #prediction
            loss = F.mse_loss(y_hat, output) #loss
            optimizer.zero_grad() #zero gradients
            loss.backward() #calculate gradients
            #perform a step/recalculate model weights
            optimizer.step()
```

Κώδικας 64 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, επέκταση Κώδικα 29 – Παράλληλη και παράλληλη κατανεμημένη εκπαίδευση

3.3.6 – Εκπαίδευση μέσα σε δίκτυο

Εξηγήσαμε στην υποενότητα 3.3.3 το πως η κάθε διεργασία, που θέλουμε να συμμετέχει στην κατανεμημένη εκπαίδευση, θα πρέπει να γνωρίζει τέσσερα πράγματα πριν την εκκίνηση της: την διεύθυνση IP της διεργασίας master (rank=0), την θύρα (port) της διεργασίας master, τον αριθμό (κόσμο) των διεργασιών που συμμετέχουν στην ομάδα επικοινωνίας (world size) και φυσικά την κατάταξη της ίδιας της διεργασίας (rank).

Εφόσον μιλάμε για εκπαίδευση μέσα σε δίκτυο, η διεργασία θα πρέπει να γνωρίζει ακόμη μια παράμετρο, η οποία είναι το όνομα της διεπαφής δικτύου, για την επικοινωνία με την διεύθυνση IP της διεργασίας master (socket interface name).

Όσον αφορά τον τρόπο επικοινωνίας έχουμε δύο επιλογές. Ο πρώτος τρόπος είναι η μεταφορά των δεδομένων μέσω μηνυμάτων, μέσω του δικτύου, χρησιμοποιώντας το πρωτόκολλο TCP. Διαφορετικά, εάν τα μηχανήματα στα οποία εδρεύουν οι διεργασίες μοιράζονται έναν κοινό χώρο αποθήκευσης, μπορούμε να χρησιμοποιήσουμε ένα αρχείο για την εγγραφή και ανάγνωση των δεδομένων που θα χρειαστεί οι διεργασίες να μοιραστούν. [32][33]

Αυτός είναι λεπτομερώς ο τρόπος που μια ομάδα επικοινωνίας διεργασιών δημιουργείται [32]:

- Καλώντας την μέθοδο init_process_group, δημιουργείται μια κλάση καναλιού με βάση το είδος του backend.
- Έπειτα, δημιουργείται το κανάλι με βάση την κλάση αυτή, προσθέτοντας αρχικά την διεύθυνση της διεργασίας master που έχει αναγνωριστικό rank=0.
- Η διεργασία με rank 0 ονομάζεται master, ενώ όλες οι υπόλοιπες ονομάζονται workers.
- Τι κάνει η master:
    - Δημιουργεί sockets για τις διεργασίες workers
    - Περιμένει όλες οι διεργασίες workers να επικοινωνήσουν με αυτή και να της παραδώσουν τις πληροφορίες τους
    - Τέλος, στέλνει σε όλες τις διεργασίες, όλες τις πληροφορίες όλων των διεργασιών
- Τι κάνει μια worker:
    - Δημιουργεί ένα socket για την επικοινωνία με την master
    - Στέλνει στην master όλες της τις πληροφορίες
    - Δέχεται από την master όλες τις πληροφορίες για τις υπόλοιπες workers
    - Ανοίγει ένα νέο socket και εκτελεί μια χειραψία με τις υπόλοιπες workers
- Όλες οι διεργασίες είναι έτοιμες να επικοινωνήσουν με όλες. Η ομάδα επικοινωνίας έχει δημιουργηθεί.

```
#the master address can either be the IP address of the master
#or any valid broadcast address within the master network
os.environ['MASTER_ADDR'] = '192.168.1.5'
os.environ['MASTER_PORT'] = '29500'
os.environ['WORLD_SIZE'] = str(world_size)
os.environ['RANK'] = str(node)
#interface name to be used by the Gloo backend.
#in Unix systems the interface name can be found with
#the ifconfig console command
os.environ['GLOO_SOCKET_IFNAME'] = 'enp3s0'

#process initialization with the gloo backend, using the tcp
#protocol to transfer data between processes
```

```
dist.init_process_group(backend='gloo',
init_method="tcp://192.168.1.5:29500", world_size=size, rank=rank)
#dist.init_process_group(backend='gloo',
init_method="file://<file_name>", world_size=size, rank=rank)
```

Κώδικας 65 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, δημιουργία ομάδας επικοινωνίας διεργασιών μέσα σε δίκτυο

3.3.7 – NCCL environment variables

Χρησιμοποιώντας το NCCL backend, πέρα από απλώς ορίζοντας το όνομα της διεπαφής NCCL_SOCKET_IFNAME, για κατανεμημένη εκπαίδευση, μπορεί κανείς να χρησιμοποιήσει επιπλέον μεταβλητές, για debugging και για άλλους σκοπούς. [37]

Μερικές από τις μεταβλητές περιλαμβάνουν:

- NCCL_DEBUG: μεταβλητή για debugging. Τιμές περιλαμβάνουν VERSION: εκτύπωση έκδοσης NCCL κατά την αρχή του προγράμματος, WARN: εκτύπωση μηνυμάτων ειδοποίησης σφάλματος, INFO: εκτύπωση πληροφοριών για debugging
- NCCL_BUFFER_SIZE: μέγεθος buffer για μεταφορά μηνυμάτων μεταξύ καρτών γραφικών
- NCCL_NTHREADS: αριθμός νημάτων CUDA ανά μπλοκ CUDA. Default = 256. Αυξάνοντας τον αριθμό αυτό θα αυξήσει τον φόρτο εργασίας στην κάρτα γραφικών. Αντίστοιχα η μείωση θα τον μειώσει.
- NCCL_DEBUG_FILE: αποθήκευση αρχείου καταγραφής debugging σε αρχείο

Περισσότερες πληροφορίες για τις μεταβλητές περιβάλλοντος της NCCL μπορούν να βρεθούν στο documentation της NCCL.

3.3.8 – Multi-GPU operations

Με πολλαπλές κάρτες γραφικών ανά μηχάνημα, είναι εύκολο να φανταστεί κανείς το γεγονός έναν αριθμό tensors να εδρεύουν σε έναν εξίσου ίσο αριθμό καρτών γραφικών, με κάθε tensor να εδρεύει σε μια διαφορετική κάρτα γραφικών.

Ένα κλασσικό πρόβλημα, όπου δεν θέλουμε μόνο να υλοποιήσουμε λειτουργίες κατανεμημένης επικοινωνίας μεταξύ διεργασιών, αλλά και μεταξύ της ίδιας διεργασίας είναι αυτό. Εάν η κάθε διεργασία χρησιμοποιεί για αυτό τον σκοπό όλες τις κάρτες γραφικών ενός μηχανήματος, με μια διεργασία ανά μηχάνημα, είναι

λογικό να θέλουμε να αξιοποιήσουμε τις κάρτες, τοποθετώντας σε κάθε μια από έναν tensor. [33]

Με βάση την παραπάνω λογική, ας υποθέσουμε πως θέλουμε να υλοποιήσουμε μια λειτουργία, πχ. την all-reduce, όχι μόνο για έναν tensor ανά διεργασία, αλλά για κάθε tensor σε κάθε κάρτα γραφικών μέσα σε κάθε διεργασία. Η λύση είναι απλή, και παρέχεται άμεσα από την βιβλιοθήκη PyTorch. Μόνη προϋπόθεση είναι πως για κάθε διεργασία, ένας ίδιος αριθμός tensors υπάρχει για την εκτέλεση της λειτουργίας κατανεμημένης επικοινωνίας. Οι tensors της κάθε διεργασίας θα πρέπει να επισυναφθούν σε μία λίστα, με κάθε διεργασία να έχει ίδιο μήκος λίστας, και φυσικά, tensors παρόμοιων διαστάσεων και τύπου, πχ. float ή double. [33]

Να σημειωθεί ακόμη, πως προς το παρόν, οι λειτουργίες κατανεμημένης επικοινωνίας μεταξύ πολλαπλών καρτών γραφικών υποστηρίζονται μόνο από το NCCL backend (Ιανουάριος 2019, PyTorch 1.0).

```
import torch
import torch.distributed as dist

dist.init_process_group(backend="nccl",
                        init_method="file:///distributed_test",
                        world_size=2,
                        rank=0)
tensor_list = []
for dev_idx in range(torch.cuda.device_count()):
    tensor_list.append(torch.FloatTensor([1]).cuda(dev_idx))

dist.all_reduce_multigpu(tensor_list)
```

Κώδικας 66 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, υλοποίηση λειτουργίας κατανεμημένης επικοινωνίας για ίσο αριθμό πολλαπλών καρτών γραφικών ανά διεργασία

3.3.9 – Χρήσιμες συμβουλές

Launch utility

Ας σκεφτούμε μια περίπτωση, παρόμοια της υποενότητας 3.3.8, όπου όμως αντί να χρησιμοποιούμε πολλές κάρτες γραφικών σε μια διεργασία ανά κόμβο, χρησιμοποιούμε έναν αριθμό διεργασιών, με κάθε διεργασία να χρησιμοποιεί μόνο μια από τις κάρτες γραφικών του μηχανήματος κόμβου.

Μια χρήσιμη λειτουργία που μπορούμε να χρησιμοποιήσουμε για αυτό το σκοπό, είναι η λειτουργία launch που παρέχεται από το πακέτο distributed της PyTorch. Ο

σκοπός της είναι η εκκίνηση ενός script, πολλές φορές, δημιουργώντας έναν αριθμό διεργασιών, με κάθε διεργασία να χρησιμοποιεί μόνο μια από τις κάρτες γραφικών. Για τον λόγο αυτό, θα πρέπει να έχουμε έναν επαρκή αριθμό καρτών γραφικών, ώστε να μπορέσουμε να δημιουργήσουμε έναν αριθμό διεργασιών, μικρότερο ή ίσο με τον αριθμό των καρτών γραφικών που διαθέτουμε. [33]

Αυτό συνεπάγεται βέβαια εκπαίδευση GPU. Μπορούμε όμως ακόμη να χρησιμοποιήσουμε την λειτουργία και για εκπαίδευση CPU, με κάθε tensor ή μοντέλο μιας διεργασίας να μοιράζεται έναν κοινό χώρο στην κύρια μνήμη, μαζί με τους υπόλοιπους tensors ή μοντέλα των άλλων διεργασιών.

Υπάρχουν δύο τρόποι που μπορούμε να χρησιμοποιήσουμε την λειτουργία launch [33]:

A. Ένας κόμβος, πολλαπλές διεργασίες: Κατά αυτή την περίπτωση χρησιμοποιούμε μόνο ένα μηχάνημα, στο οποίο εκκινούμε διεργασίες αριθμού μικρότερου ή ίσου με τον αριθμό καρτών γραφικών που διαθέτουμε στο μηχάνημα.
B. Πολλαπλοί κόμβοι, πολλαπλές διεργασίες: Σε αυτή την περίπτωση έχουμε πολλούς κόμβους, που εδρεύουν σε διαφορετικά σημεία, μέσα σε ένα δίκτυο, με τον κάθε κόμβο να δημιουργεί έναν αριθμό διεργασιών. Θα πρέπει να γνωρίζουμε ακόμη την διεύθυνση IP του master κόμβου (node 0) καθώς και την θύρα (port) για την επικοινωνία μαζί του.

Παρακάτω παρουσιάζουμε το πως μπορεί να υλοποιηθεί ο κάθε τρόπος, με την χρήση της λειτουργίας launch.

```
>>> python -m torch.distributed.launch
           --nproc_per_node=NUM_GPUS_YOU_HAVE
           YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3)
```

Κώδικας 67 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, λειτουργία launch, περίπτωση Α

```
>>> python -m torch.distributed.launch
           --nproc_per_node=NUM_GPUS_YOU_HAVE
           --nnodes=2 --node_rank=0 --master_addr="192.168.1.1"
           --master_port="1234"
           YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3)
```

Κώδικας 68 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, λειτουργία launch, περίπτωση Β

Για περισσότερες πληροφορίες σχετικά με το πως συντάσσεται η εντολή launch, αρκεί να πληκτρολογήσει κανείς `python -m torch.distributed.launch -h` για την εμφάνιση βοήθειας. Η εντολή launch πληκτρολογείται σε τερματικό, χρησιμοποιώντας python interpreter με εγκατεστημένη τη βιβλιοθήκη PyTorch, για συστήματα Unix, ή σε τερματικό Anaconda (Anaconda Prompt), για συστήματα Windows που έχουν εγκατεστημένη την πλατφόρμα Anaconda και την βιβλιοθήκη PyTorch μέσα στο περιβάλλον της πλατφόρμας. Να σημειωθεί πως συνιστάται η χρήση λειτουργικού συστήματος Unix, καθώς πολλές λειτουργίες της PyTorch μπορεί να μην είναι διαθέσιμες για Windows, ειδικά του πακέτου distributed. Για περισσότερες πληροφορίες σχετικά με την πλατφόρμα Anaconda ανατρέξτε στον οδηγό χρήσης λογισμικού.

Όσον αφορά το πρόγραμμα, θα χρειαστεί να γνωρίζουμε με κάποιον τρόπο τις τέσσερις απαραίτητες παραμέτρους για την δημιουργία της ομάδας επικοινωνίας, master IP address, master port, rank και world size.

Η λειτουργία launch καθορίζει τις παραμέτρους world size, master IP address και master port, δημιουργώντας νέες μεταβλητές περιβάλλοντος για κάθε διεργασία. Η μόνη ίσως παράμετρος που θα χρειαστεί να γνωρίζουμε είναι το αναγνωριστικό rank της διεργασίας. Το rank περνά σαν παράμετρος από την ίδια την λειτουργία launch, στο πρόγραμμα, με όνομα local_rank, την οποία θα χρειαστεί και να διαβάσουμε. [33]

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--local_rank", type=int)
args = parser.parse_args()

#each process uses only the GPU with ID corresponding to the
#process rank number
torch.cuda.set_device(arg.local_rank)
```

Κώδικας 69 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, λειτουργία launch, local rank argument parsing και αντιστοίχιση διεργασίας σε κάρτα γραφικών

Όσον αφορά τον τρόπο δημιουργίας ομάδας επικοινωνίας, χρησιμοποιούμε το string "env://" σαν παράμετρο του πεδίου init_method, κατά την κλήση της μεθόδου init_process_group του πακέτου distributed. Με αυτό τον τρόπο, λέμε στο πρόγραμμα, πως μπορεί να βρει τις παραμέτρους master address IP, master port

και world size, στις παραμέτρους περιβάλλοντος (os.environ["<variable_name>"], export <variable_name>) MASTER_ADDR, MASTER_PORT και WORLD_SIZE, αντίστοιχα. Να σημειωθεί πως αυτό ορίζεται ως ένας «νέος» τρόπος δημιουργίας ομάδων επικοινωνίας, πέρα από το TCP και την κοινόχρηστη χρήση ενός κοινού χώρου αποθήκευσης, χωρίς όμως να χρησιμοποιείται ένα διαφορετικό πρωτόκολλο που δεν χρησιμοποιούν οι προηγούμενοι δύο. [33]

Κατά την ενθυλάκωση ακόμη ενός μοντέλου, σε ένα DistributedDataParallel module, θα χρειαστεί να ορίσουμε κατάλληλα τις παραμέτρους device_ids και output_device, ώστε η κάθε διεργασία να χρησιμοποιήσει για την εκπαίδευση, μονάχα την κάρτα γραφικών που της αντιστοιχεί.

```
torch.distributed.init_process_group(backend='YOUR BACKEND',
                                     init_method='env://')

model = torch.nn.parallel.DistributedDataParallel(model,
        device_ids=[arg.local_rank], output_device=arg.local_rank)
```

Κώδικας 70 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, λειτουργία launch, δημιουργία ομάδας επικοινωνίας και ενθυλάκωση μοντέλου σε DistributedDataParallel module

Spawn utility

Ένας οικονομικός τρόπος να γλυτώσει κανείς πολλές σειρές κώδικα, κατά την δημιουργία διεργασιών με το πακέτο multiprocessing, είναι η λειτουργία spawn. Κοιτώντας ξανά τον Κώδικα 62, μπορούμε πολύ εύκολα να εξοικονομήσουμε κώδικα με μια μόνο εντολή. Το μόνο που χρειάζεται να γνωρίζουμε είναι πόσες διεργασίες θα θέλαμε να δημιουργήσουμε, την μέθοδο εκκίνησης διεργασιών (init_processes, Κώδικας 61) και τις παραμέτρους που θα χρειαστούμε, μέσα σε μια δομή tuple.

Η λειτουργία spawn προσφέρεται από την έκδοση PyTorch 1.0 και αργότερα, για εκδόσεις Python 3.4 και νεότερες. [33]

Το παρακάτω παράδειγμα κώδικα κάνει ό,τι ακριβώς και ο Κώδικας 62, σε μία μόνο γραμμή. Να σημειωθεί μόνο, πως η κάθε διεργασία θα πρέπει να αναγνωρίσει το δικό της rank σε αυτή την περίπτωση, μέσα από μια λίστα αναγνωριστικών ή με κάποιο άλλο τρόπο. Για το σκοπό αυτό, η μέθοδος init_processes επιδέχεται επιπλέον αλλαγές, τις οποίες όμως δεν θα δείξουμε εδώ.

```
import torch.multiprocessing as mp #multiprocessing
```

```
mp.spawn(init_processes, args=([0,1], 2, all_reduce), n_procs=2)
```

Κώδικας 71 – Παράλληλες κατανεμημένες εφαρμογές σε PyTorch, λειτουργία spawn

3.4 – Επίλογος

Σε αυτό το κεφάλαιο επεκταθήκαμε πάνω στην βιβλιοθήκη PyTorch και είδαμε, αρχικά, πως μπορεί κανείς να μπορεί κανείς να την χρησιμοποιήσει για να φτιάξει tensors, μια δομή παρόμοια με τα numpy arrays, και ένα δικό του μοντέλο, με την χρήση του nn module, και πως αυτό το μοντέλο μπορεί να εκπαιδευτεί. Έπειτα, παρουσιάσαμε τα αντικείμενα Dataset και DataLoader και δείξαμε πως μπορεί κανείς να διαχωρίσει τα δεδομένα ενός Dataset σε batches, με την χρήση ενός DataLoader, για την εκπαίδευση ενός μοντέλου.

Δώσαμε πληροφορίες για το πως μπορεί κανείς να αποθηκεύσει ένα εκπαιδευμένο μοντέλο, καθώς και άλλες πληροφορίες, και πως μπορούν αυτά να ανακτηθούν σε ένα πρόγραμμα, παρουσιάσαμε συμβουλές για το πως μπορεί κανείς να παρακολουθήσει την χρήσης της μνήμης της κάρτας γραφικών κατά την εκπαίδευση με χρήση GPU και τι θα χρειαστεί να αποφύγει, δώσαμε χρήσιμες συμβουλές για την εξοικονόμηση πόρων και το πως μπορεί κανείς να αναπαράγει ένα αποτέλεσμα, σε διαφορετικές εκτελέσεις ενός προγράμματος, και κλείσαμε την ενότητα παρουσιάζοντας βήμα-βήμα την διαδικασία δημιουργίας και εκπαίδευσης ενός αλγορίθμου σύστασης, βασισμένου σε μάθηση, με την χρήση της PyTorch.

Στην επόμενη ενότητα δείξαμε πως μπορεί κανείς να χρησιμοποιήσει το πακέτο distributed της PyTorch. Με την χρήση του πακέτου, είναι δυνατό ένας αριθμός διεργασιών, που εδρεύουν μέχρι και σε διαφορετικά σημεία, μέσα σε ένα δίκτυο, να επικοινωνήσουν μεταξύ τους, είτε με τη μεταφορά μηνυμάτων, είτε με τη χρήση ενός κοινού αποθηκευτικού χώρου.

Παρουσιάσαμε τα πρωτόκολλα επικοινωνίας, τους τρόπους επικοινωνίας και σε τι πρωτόκολλα αντιστοιχούν, με παραδείγματα, δείξαμε πώς σε ένα κυρίως πρόγραμμα μπορούμε να δημιουργήσουμε άλλες διεργασίες, με σκοπό την μεταξύ τους συνεργασία και επικοινωνία για την εκπαίδευση ενός μοντέλου, και παρουσιάσαμε το πως αυτό το μοντέλο μπορεί να εκπαιδευτεί συνεργατικά από τις διεργασίες αυτές.

Παρουσιάσαμε επιπλέον χρήσιμες συμβουλές, όπως το πως μπορεί κανείς να χρησιμοποιήσει μεταβλητές περιβάλλοντος ενός πρωτοκόλλου επικοινωνίας για σκοπούς debugging, το πως μια διεργασία μπορεί να αξιοποιήσει λειτουργίες κατανεμημένης επικοινωνίας για πολλαπλές συσκευές (κάρτες γραφικών) και δείξαμε τρόπους διευκόλυνσης όσον αφορά την εκτέλεση ενός προγράμματος πολλές φορές, εκκινώντας έναν αριθμό κατανεμημένων διεργασιών σε έναν κόμβο (μηχάνημα), και την δημιουργία διεργασιών μέσα σε ένα κύριο πρόγραμμα.

Χρησιμοποιώντας όλα όσα μάθαμε από τα δύο τελευταία κεφάλαια, σκοπεύουμε εν τέλει να παρουσιάσουμε, στο επόμενο κεφάλαιο, το πως μπορεί κανείς να εκπαιδεύσει ένα μοντέλο αλγορίθμου συστάσεων, βασισμένου σε μάθηση, παραγοντοποίησης, συνεργατικά, με τη χρήση μιας ομάδας κατανεμημένης επικοινωνίας διεργασιών.

Θα παρουσιάσουμε δύο scripts που προορίζονται για εκτέλεση σε έναν κόμβο. Το πρώτο θα χρησιμοποιεί μια μόνο διεργασία, αυτή του κυρίως προγράμματος, ενώ το δεύτερο θα δημιουργεί έναν αριθμό ξεχωριστών διεργασιών, με τις διεργασίες να λαμβάνουν το αναγνωριστικό τους rank μέσα από ένα εύρος τιμών. Οι διεργασίες αυτών των δύο scripts μπορούν να συνεργαστούν, ανεξαρτήτως του είδους του script. Τα scripts αυτά μπορούν να τρέξουν σε ένα cluster υπολογιστών, δεδομένου πως οι υπολογιστές αυτοί μπορούν να επικοινωνήσουν μεταξύ τους, ακόμη και αν βρίσκονται σε διαφορετικά δίκτυα. Η κάθε διεργασία θα χρησιμοποιήσει όλες τις διαθέσιμες κάρτες γραφικών ενός μηχανήματος για την εκπαίδευση (DistributedDataParallel module, default τιμή παραμέτρου device_ids). Τα scripts μπορούν να εκτελεστούν περισσότερες από μια φορές ανά μηχάνημα. Μόνη προϋπόθεση είναι η σωστή διευθέτηση των αναγνωριστικών των διεργασιών κατά την εκτέλεση των scripts.

ΚΕΦΑΛΑΙΟ 4 – Παράλληλη και κατανεμημένη υλοποίηση αλγορίθμων συστάσεων

4.1 – Παράλληλη κατανεμημένη εκπαίδευση ενός αλγορίθμου σύστασης παραγοντοποίησης πινάκων (Matrix Factorization) βασισμένου σε μάθηση (Learning Based) (SVD)

Ας ανακεφαλαιώσουμε. Σε αυτή την ενότητα θα φτιάξουμε ένα σύστημα συστάσεων αλγορίθμου SVD, όπως ορίστηκε στην ενότητα 2.3.2 και υλοποιήθηκε στην ενότητα 3.2.9, το οποίο θα εκπαιδεύσουμε μέσα σε ένα cluster υπολογιστών. [38]

Θα χρειαστούμε:

- Ένα training και ένα test dataset, τα οποία θα είναι κοινά για όλους τους υπολογιστές κόμβους
- Ένα μοντέλο, εξίσου κοινό μεταξύ κόμβων

Οι κόμβοι θα πρέπει να γνωρίζουν εξ' αρχής τα dataset και το μοντέλο, καθώς η εκ νέου δημιουργία αυτών, από τον κάθε κόμβο, μπορεί να δημιουργήσει αποκλίνοντα αποτελέσματα.

Το πρώτο που θα χρειαστούμε είναι να δημιουργήσουμε τα κοινά dataset. Για το σκοπό αυτό θα χρησιμοποιήσουμε και πάλι το MovieLens dataset που χρησιμοποιήσαμε και στην ενότητα 3.2.9 (ml-latest-small) και θα επαναλάβουμε την διαδικασία διαχωρισμού σε test και training sets. [6]

Έπειτα, θα δημιουργήσουμε το μοντέλο, με βάση τον αριθμό των χρηστών και των αντικειμένων του dataset και τέλος θα αποθηκεύσουμε τα training και test sets, καθώς και το μοντέλο, στο σύστημα αρχείων.

```
#part 1: importing

#coded methods
from split_selection import input_select
from data_encode import encode, split
from models import SVD
from datasets import CF_Dataset
```

```
#libraries
#import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
#import os #operating system
#import sys #system
import torch #PyTorch

#reproducibility
#torch.manual_seed(0)
#np.random.seed(0)

#part 2: preparing data

#import data
print(f'\nLoading the data...')
PATH = Path("../Datasets/ml-latest-small")
data = pd.read_csv(PATH/"ratings.csv")

#encode the dataset into unique and contiguous values
data = encode(data)
print(f'Dataset has been encoded')

num_ratings = len(data.rating)
num_users = len(data.userId.unique())
num_items = len(data.movieId.unique())
sparsity = num_ratings/(num_users*num_items)
mean_rating = data.rating.mean()

print(f'\nNumber of users: {num_users}')
print(f'Number of items: {num_items}')
print(f'Number of ratings: {num_ratings}')
print(f'Dataset sparsity: {sparsity}')
print(f'\nAverage rating: {mean_rating}\n')

#splitting variables input
method, value = input_select()

print(f'\nSplitting:\nMethod: {method}\nValue: {value}')

#dataset splitting
print('\nSplitting data into training and testing set\n')
```

```
train_set, test_set = split(data, method, value)

print(f'\nTraining set: {train_set}\n')
print(f'\nTesting set: {test_set}\n')

print(f'\nMaking the torch datasets\n')
train_dataset = CF_Dataset(train_set)
test_dataset = CF_Dataset(test_set)

torch.save(train_dataset, 'movielens_training_set.pkl')
torch.save(test_dataset, 'movielens_test_set.pkl')

#part 3: modeling

model = SVD(num_users, num_items, mean_rating)
#set the model to cuda for gpu training
#model = model.cuda()
print(f'Model initialized: \n{model}')

torch.save(model, 'model.pkl')
```

Κώδικας 72 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, δημιουργία κοινών dataset και μοντέλου

4.1.1 Κόμβος μίας διεργασίας (Node_SingleProc.py)

Καθώς η υλοποίηση ανάμεσα στα είδη κόμβων που θα δημιουργήσουμε είναι διαφορετική, παρά τα κοινά τους σημεία, θα συμπεριλάβουμε τα κοινά αυτά σημεία σε αυτή την υποενότητα.

Έχοντας δημιουργήσει τα datasets και το μοντέλο μας, επόμενο βήμα είναι η φόρτωση τους μέσα από τον κάθε κόμβο. Ας προσπαθήσουμε και πάλι να θυμηθούμε τα 5 βήματα υλοποίησης:

- Εισαγωγή (importing) βιβλιοθηκών και μεθόδων
- Προετοιμασία δεδομένων
- Μοντελοποίηση
- Εκπαίδευση
- Αξιολόγηση

Για την υλοποίηση μας θα χρειαστούμε:

- Τις μεθόδους:

    - Εκπαίδευσης, train_model
    - Υπολογισμού βαρών μη εκπαιδευμένων χρηστών και αντικειμένων, set_model_untrained_weights
    - Αξιολόγησης, test_loss
- Τις βιβλιοθήκες και πακέτα:
    - time, για την χρονομέτρηση της εκπαίδευσης
    - sys, για την ανάγνωση παραμέτρων
    - torch
    - torch.distributed

Ο λόγος που δεν χρειαζόμαστε περισσότερες μεθόδους είναι ότι το μοντέλο και τα datasets μας είναι έτοιμα, έτσι μέθοδοι όπως ο διαχωρισμός των δεδομένων και η εισαγωγή του τρόπου διαχωρισμού δεν μας είναι αναγκαίες.

```
#part 1: importing

from training import train_model, test_loss,
set_model_untrained_weights

import time #time
import sys #system
import torch #PyTorch
import torch.distributed as dist #distributed

#torch.set_num_threads(os.cpu_count())

#reproducibility
#torch.manual_seed(0)
#np.random.seed(0)
```

Κώδικας 73 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, κόμβος μιας διεργασίας, εισαγωγή μεθόδων και βιβλιοθηκών

Για τα βήματα προετοιμασίας των δεδομένων και μοντελοποίησης, δεν θα χρειαστεί παρά μόνο να φορτώσουμε τα έτοιμα datasets και το μοντέλο που δημιουργήσαμε. Φυσικά, ο κάθε κόμβος θα πρέπει να έχει πρόσβαση σε αυτά, οπότε η ύπαρξή τους, σε προσβάσιμο σημείο από τον κάθε κόμβο, είναι απαραίτητη.

```
#part 2: preparing data
```

```
#load the datasets (torch.utils.data.Dataset)
train_dataset = torch.load('movielens_training_set.pkl')
test_dataset = torch.load('movielens_test_set.pkl')

print(f'\nTraining set: {train_dataset.data}')
print(f'\nTesting set: {test_dataset.data}')

#part 3: modeling

#load the model
model = torch.load("model.pkl").cuda()
print(f'\nModel: {model}')
```

Κώδικας 74 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, κόμβος μιας διεργασίας, φόρτωση datasets και μοντέλου

Εφόσον μιλάμε για κόμβο μιας διεργασίας, ο κάθε κόμβος θα πρέπει να γνωρίζει το αναγνωριστικό της διεργασίας του (rank). Ακόμη, θα πρέπει να γνωρίζει το μέγεθος του κόσμου στον οποίο θα συμμετέχει (world size).

Για το σκοπό αυτό, φροντίσαμε κατά την κλήση του προγράμματος να απαιτούνται αυτά τα δύο ως παράμετροι. Ο χρήστης θα χρειαστεί λοιπόν να εισάγει όχι μόνο το όνομα του script, αλλά και τις παραμέτρους αυτές. Το script της μιας διεργασίας μπορεί να κληθεί με τον παρακάτω τρόπο, μέσω τερματικού.

```
#rank = 0, world size = 2

>>> python Node_SingleProc.py 0 2
```

Κώδικας 75 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, κόμβος μιας διεργασίας, χρήση από τερματικό

Θα χρειαστούμε ακόμη, την θύρα και την διεύθυνση IP της διεργασίας master. Εάν ο κόμβος που τρέχουμε είναι η διεργασία 0, τότε η διεύθυνση IP θα είναι του μηχανήματος που χρησιμοποιούμε. Ο αριθμός θύρας θα είναι ένας αριθμός που ορίζουμε εμείς. Όπως αναφέραμε και προηγουμένως, προτιμούμε την θύρα 29500.

Εάν οι κόμβοι εδρεύουν εντός τοπικού δικτύου, τότε μια διεύθυνση του τοπικού δικτύου 192.168.1.0 αρκεί. Διαφορετικά, θα πρέπει να γνωρίζουμε την διεύθυνση και τη θύρα που το router παρουσιάζει στον έξω κόσμο, εάν το router χρησιμοποιεί NAT.

Γνωρίζοντας τις τέσσερις αυτές παραμέτρους, θα δημιουργήσουμε μια νέα διεργασία κατανεμημένης επικοινωνίας. Η κάθε διεργασία που θα δημιουργήσουμε

θα περιμένει τις υπόλοιπες, έως ότου ο αριθμός των διεργασιών συμπληρώνει τον αριθμό του κόσμου.

Τέλος θα εκπαιδεύσουμε το κοινό μοντέλο και ο κάθε κόμβος θα το αξιολογήσει. Εάν όλα πήγαν σωστά, η αξιολόγηση του μοντέλου θα πρέπει να είναι η ίδια για κάθε κόμβο.

```
#part 4: training

backend = 'gloo'
node = int(sys.argv[1])
world_size = int(sys.argv[2])

#os.environ['MASTER_ADDR'] = '192.168.1.5'
os.environ['MASTER_ADDR'] = '127.0.01'
os.environ['MASTER_PORT'] = '29500'
os.environ['WORLD_SIZE'] = str(world_size)
os.environ['RANK'] = str(node)
#os.environ['GLOO_SOCKET_IFNAME'] = 'enp3s0'
#dist.init_process_group(backend,
init_method="tcp://192.168.1.5:29500", world_size=size, rank=rank)
dist.init_process_group(backend, world_size=world_size, rank=node)

start = time.time()
train_model(model, train_dataset, cuda=True,
distributed_mode=True, rank=node, world_size=world_size)
end = time.time()

print(f'\nTime elapsed for training: {end-start} sec')

#average model weights for untrained users/items
set_model_untrained_weights(model, train_dataset, test_dataset,
cuda=True)

#part 5: testing

test_loss(model, test_dataset, cuda=True)
```

Κώδικας 76 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, κόμβος μιας διεργασίας, εκπαίδευση και αξιολόγηση

```
#training.py

import torch
```

```
import torch.nn.functional as F
import torch.distributed as dist
from torch.utils.data import DistributedSampler, DataLoader
from torch.nn.parallel import DistributedDataParallelCPU,
DistributedDataParallel
import threading
import queue

from misc import queue_iter_print

#factorization module training. optimizer is Adam
#by default. options for distributed training
#and cuda training.
#prints training error per every epoch and batch
def train_model(model, train_dataset, epochs=10, lr=0.01,
                wd=0.0, unsqueeze=False, cuda=False,
                distributed_mode=False, rank=None,
                world_size=None):

    rank_prt = f"Rank {rank}: "
    sampler = None

    #printing is assigned to a new thread to minimize training
    #time by removing
    #I/O interrupts
    #a queue will handle the passing of the print string messages
    #to the new thread
    q = queue.Queue()
    print_thread = threading.Thread(target=queue_iter_print,
args=(q,))
    print_thread.start()

    #distributed mode
    if distributed_mode:
        q.put(f"{rank_prt}Creating distributed sampler")
        sampler = DistributedSampler(train_dataset,
num_replicas=world_size, rank=rank)

        if cuda:
            model = DistributedDataParallel(model)
        else:
            model = DistributedDataParallelCPU(model)
```

```
        q.put(f"{rank_prt}Set the model to distributed data
parallel")

    #non-distributed. set the model to DataParallel to increase
    #training speed
    elif not distributed_mode and cuda:
        model = torch.nn.DataParallel(model)

    q.put(f"{rank_prt}Making Dataloader")
    dataloader = DataLoader(train_dataset, batch_size=1000,
                            num_workers=4, sampler=sampler)

    #optimizer created after the model has been set to a definite
    #location
    optimizer = torch.optim.Adam(model.parameters(),lr=lr,
                                 weight_decay=wd)
    q.put(f'{rank_prt}Created optimizer')

    model.train()
    q.put(f'{rank_prt}Model is in training mode\n')

    for i in range(epochs):

        if distributed_mode:
            sampler.set_epoch(i)

        for batch_idx, batch_sample in enumerate(dataloader):

            train_prt = f"Epoch {i}, batch {batch_idx}: "

            users = batch_sample["userId"].long()
            items = batch_sample["movieId"].long()
            ratings = batch_sample["rating"].float()

            if unsqueeze:
                ratings = ratings.unsqueeze(1)

            if cuda:
                users = users.cuda()
                items = items.cuda()
                ratings = ratings.cuda()
```

```
            y_hat = model(users, items) #prediction
            loss = F.mse_loss(y_hat, ratings) #loss
            optimizer.zero_grad() #zero gradients
            loss.backward() #update gradients
            optimizer.step() #step
            q.put(f'{rank_prt}{train_prt}Training loss:
{loss.item()} ')

    q.put(".end") #signal the print function to return
    print_thread.join() #and close the thread


#misc.py

#iteratively prints elements of a queue. end keyword is ".end"
def queue_iter_print(q):

    while True:
        prt = q.get()
        if prt == ".end": break
        print(prt)
```

Κώδικας 77 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, μέθοδος εκπαίδευσης, επέκταση Κώδικα 49

```
#training.py

import torch
import torch.nn.functional as F

#calculates loss error between the prediction and the testing
#set. default method of calculation is mean squared error
def test_loss(model, test_dataset, unsqueeze=False,
              cuda=False):
    model.eval()

    users = torch.LongTensor(test_dataset.data.userId.values)
    items = torch.LongTensor(test_dataset.data.movieId.values)
    ratings = torch.FloatTensor(test_dataset.data.rating.values)

    if cuda == True:
        users = users.cuda()
        items = items.cuda()
        ratings = ratings.cuda()
```

```
    if unsqueeze:
        ratings = ratings.unsqueeze(1)
    y_hat = model(users, items)
    loss = F.mse_loss(y_hat, ratings)
    print(f'\nTest loss: {loss.item()}')
```

Κώδικας 78 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, μέθοδος αξιολόγησης

```
#training.py

import torch

from data_encode import get_untrained

#sets the model weights for items and users not included
#in the training set (untrained and unchanged weights)
#to the mean item or user weight.
#a classic cold start problem solution
def set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=False):

    untrained_users, untrained_items = get_untrained(test_dataset,
train_dataset)

    user_trained_indices =
torch.LongTensor(train_dataset.data['userId'].unique())
    item_trained_indices =
torch.LongTensor(train_dataset.data['movieId'].unique())

    if cuda == True:
        user_trained_indices = user_trained_indices.cuda()
        item_trained_indices = item_trained_indices.cuda()

    user_trained_weights =
torch.index_select(model.user_emb.weight, dim=0, index =
user_trained_indices)
    item_trained_weights =
torch.index_select(model.item_emb.weight, dim=0, index =
item_trained_indices)

    user_trained_average = user_trained_weights.mean(0)
    item_trained_average = item_trained_weights.mean(0)
```

```
    for i in untrained_users:
        model.user_emb.weight[i] = user_trained_average

    for i in untrained_items:
        model.item_emb.weight[i] = item_trained_average
```

Κώδικας 79 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, μέθοδος υπολογισμού βαρών για χρήστες και αντικείμενα που δεν συμμετείχαν στην εκπαίδευση

4.1.2 Κόμβος πολλαπλών διεργασίων (Node_MultiProc.py)

Με παρόμοιο τρόπο θα υλοποιήσουμε και το script για τον κόμβο δημιουργίας πολλαπλών διεργασιών. Μέσα στο πρόγραμμα, θα υλοποιήσουμε μια μέθοδο εκκίνησης διεργασιών με τον τρόπο που παρουσιάστηκε στον Κώδικα 61. Οι διεργασίες θα δημιουργηθούν με τον τρόπο που παρουσιάστηκε στον Κώδικα 62.

Φυσικά θα χρειαστεί να προστατέψουμε όσα δεδομένα δεν θέλουμε να δημιουργηθούν εκ νέου στις νέες διεργασίες, με μια ενότητα if που θα ελέγχει το όνομα του προγράμματος. Υπενθυμίζουμε πως στην κύρια διεργασία το όνομα (__name__) είναι '__main__', ενώ στις υπόλοιπες το όνομα είναι το όνομα του script. [32]

Να σημειωθεί πως η κάθε διεργασία θα πρέπει να χρησιμοποιεί ένα διαφορετικό αντίγραφο του μοντέλου. Ο λόγος είναι πως χρησιμοποιώντας το ίδιο, το μοντέλο θα εκπαιδευτεί από όλες τις διεργασίες ταυτόχρονα, με την ενημέρωση στις παραμέτρους να γίνεται πολλές φορές από διαφορετικές διεργασίες. Αυτό τελικώς μπορεί να αποφέρει μικρότερη ή μεγαλύτερη τιμή σφάλματος, κατά την αξιολόγηση, με κάθε κόμβο να εμφανίζει διαφορετική τιμή, ανάλογα με τον αριθμό των διεργασιών που συμμετείχαν στον κόμβο.

```
#part 1: importing

#coded methods
from training import train_model, test_loss,
set_model_untrained_weights

#libraries
import numpy as np #numpy
import time #time
import os #operating system
import sys #system
```

```
import copy #copy
import torch #PyTorch
import torch.multiprocessing as mp #multiprocessing
import torch.distributed as dist #distributed

#reproducibility
#torch.manual_seed(0)
#np.random.seed(0)

#process initialization method
def init_processes(rank, size, fn, *args, backend='gloo',
**kargs):
    #os.environ['MASTER_ADDR'] = '192.168.1.5'
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    os.environ['WORLD_SIZE'] = str(size)
    os.environ['RANK'] = str(rank)
    #os.environ['GLOO_SOCKET_IFNAME'] = 'enp3s0'
    #dist.init_process_group(backend,
init_method="tcp://192.168.1.5:29500", world_size=size, rank=rank)
    dist.init_process_group(backend, world_size=size, rank=rank)
    fn(*args, rank=rank, world_size=size, **kargs)


#protection of resources and unrestrained spawning control
if __name__ == "__main__":

    #part 2: preparing data

    #load the datasets (torch.utils.data.Dataset)
    train_dataset = torch.load('movielens_training_set.pkl')
    test_dataset = torch.load('movielens_test_set.pkl')

    print(f'\nTraining set: \n{train_dataset.data}\n\n')
    print(f'\nTesting set: \n{test_dataset.data}\n\n')

    #part 3: modeling

    #load the model
    model = torch.load('model.pkl').cuda()
    print(model)
```

```
    #part 4: training

    first_task, last_task = eval(sys.argv[1])
    world_size = int(sys.argv[2])

    #train_args = [model, train_dataset]
    kw_train_args = {'distributed_mode':True, 'cuda':True}

    #parallel section
    start = time.time() #start counter
    mp.set_start_method('spawn')
    processes = []
    print('\nTraining is distributed. One model copy per
process\n\n')
    for rank in range(first_task, last_task+1):
        #new model reference for each process
        model = copy.deepcopy(model)
        #reinstate the train arguments
        train_args = [model, train_dataset]
        #arguments
        arguments = [rank, world_size, train_model]
        arguments.extend(train_args)
        #key-worded arguments
        kw_arguments = {}
        kw_arguments.update(kw_train_args)
        #init
        p = mp.Process(target=init_processes, args=arguments,
kwargs=kw_arguments)
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    end = time.time() #end counter

    print(f'\nTime elapsed for training: {end-start} sec')

    #average model weights for untrained users/items
    set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=True)
```

```
    #part 5: testing

    test_loss(model, test_dataset, cuda=True)
```

Κώδικας 80 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, κόμβος πολλαπλών διεργασιών

```
#task range = 1 to 3, world size = 4

>>> python Node_MultiProc.py 1,3 4
```

Κώδικας 81 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, κόμβος πολλαπλών διεργασιών, χρήση από τερματικό

4.2 – Αξιολόγηση

Για τις δοκιμές μας, θα χρησιμοποιήσουμε τρία διαφορετικά μηχανήματα:

- Υπολογιστής Α:
  - Λειτουργικό σύστημα: Ubuntu 18.04.1 LTS
  - PyTorch version: 1.0.0 (0.4.1 source version)
  - CPU: AMD Ryzen 5 1600X 6-Core (12 threads)
  - RAM: 8 Gb DDR4
  - GPU: GTX 660 2Gb
- Υπολογιστής Β:
  - Λειτουργικό σύστημα: Ubuntu 16.04.6 LTS
  - PyTorch version: 1.0.0
  - CPU: Intel Core i3-7100 @ 3.9 GHz
  - RAM: 16 Gb
  - GPU: Titan Xp 12 Gb
- Υπολογιστής Γ:
  - Λειτουργικό σύστημα: Ubuntu 16.04.6 LTS
  - PyTorch version: 1.0.0
  - CPU: Intel Core i5 750 @ 2.6 GHz

- RAM: 12 Gb
- GPU: GTX 1070 8 Gb

Θέλουμε να υπολογισούμε το μέσο τετραγωνικό σφάλμα (test loss) σε συνάρτηση ως προς:

- Τον αριθμό των εποχών
- Το μέγεθος του batch του DataLoader
- Τον αριθμό των παραγόντων για αντικείμενα και χρήστες
- Το βήμα εκπαίδευσης (learning rate) και την φθίση των βαρών (weight decay)
- Και για την παράλληλη κατανεμημένη εκπαίδευση, τον αριθμό των διεργασιών

4.2.1 – Batch size και αριθμός εποχών

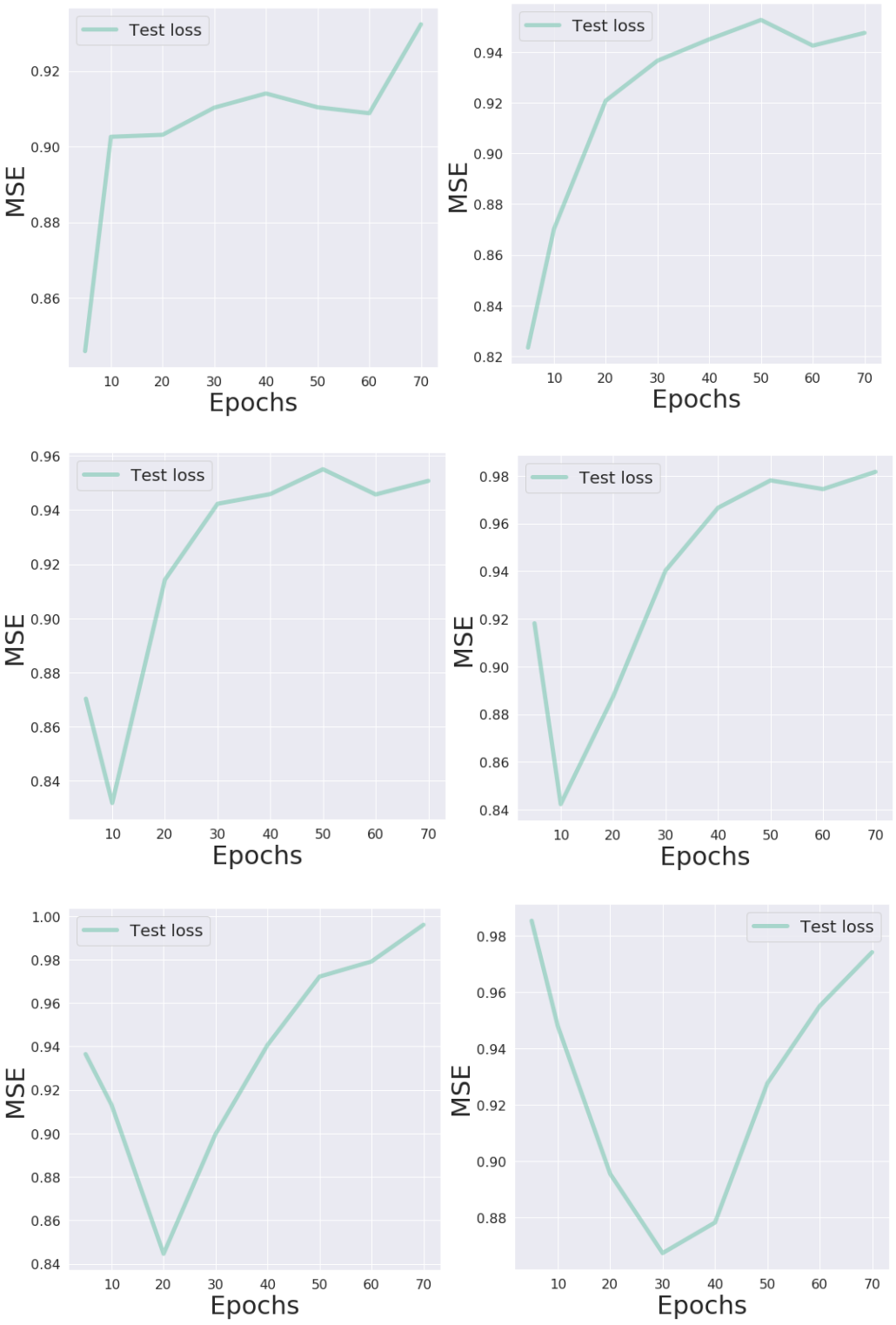
Για την εκπαίδευση θα διαχωρίσουμε τα δεδομένα σε test και training set, μεγέθους 20% και 80% αντίστοιχα. Χρησιμοποιώντας το MovieLens dataset των 100004 ratings (ml-latest-small) που χρησιμοποιήσαμε και στην υποενότητα 3.2.9, έχουμε με αυτόν τον τρόπο ένα training set 800003 ratings και ένα test set 20001 ratings, αρκετά καλών ποσοστών και ποσών για την εκτίμηση του αλγορίθμου μας. Θέτουμε ακόμη torch και numpy seeds ίσα με μηδέν, έτσι ώστε σε κάθε εκτέλεση ο διαχωρισμός των δεδομένων να γίνεται με τον ίδιο ακριβώς τρόπο.

Θα δοκιμάσουμε να εκπαιδεύσουμε με batch size 1001, 5001, 10001, 20001, 40002 και 80003 (ολόκληρο το training set) για 5, 10, 20, 30, 40, 50, 60 και 70 εποχές. Ο υπολογισμός των batch size έγινε έτσι ώστε να διαιρούνται όσο το δυνατόν καλύτερα με το συνολικό μέγεθος του training set. Εάν για παράδειγμα χρησιμοποιούσαμε batch size των 5000 αντί για 5001, θα είχαμε ως αποτέλεσμα έναν αριθμό 17 batches, με το τελευταίο batch να περιέχει μονάχα 3 βαθμολογήσεις. Στόχος είναι η εύρεση του καταλληλότερου μεγέθους batch που προσεγγίζει γρηγορότερα το ελάχιστο σφάλμα εκτίμησης.

Ορίζουμε ακόμη αριθμό παραγόντων ίσο με 100, βήμα εκπαίδευσης ίσο με 0.01 και weight decay ίσο με 0, με optimizer τον Adam. Οι παράγοντες πόλωσης αρχικοποιούνται τυχαία, με μια ομοιόμορφη κατανομή ανάμεσα σε 0 και 0.005, ενώ

οι παράγοντες βαρών αρχικοποιούνται επίσης τυχαία, με μια ομοιόμορφη κατανομή ανάμεσα σε -0.01 και 0.01.

Από όλα τα μεγέθη batch επιλέγουμε το μέγεθος των 5001 βαθμολογήσεων, καθώς από όλες τις δοκιμές μας αποτελεί το μέγεθος που παρέχει το μικρότερο σφάλμα ελέγχου, ήδη από 5 εποχές.

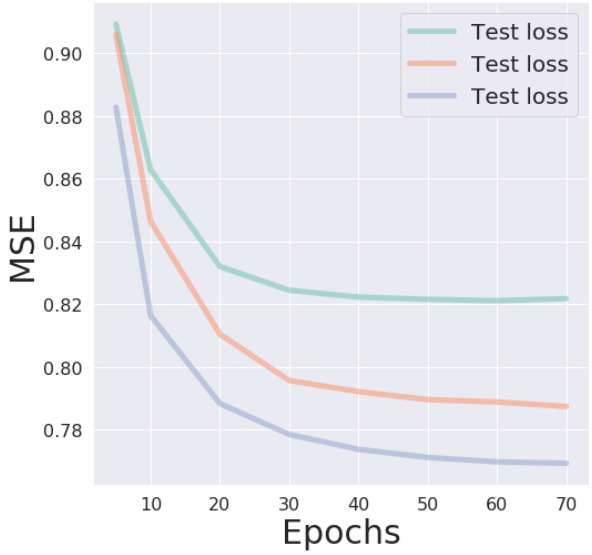


Εικόνες 10 – Εκτίμηση μεγέθους batch, βήμα εκπαίδευσης 0.01, weight decay 0.0, αριθμός παραγόντων 100. Από πάνω προς τα κάτω (batch size): 1001, 5001, 10001, 20001, 40002, 80003

4.2.2 – Αριθμός παραγόντων και weight decay

Επιλέγοντας το batch των 5001 βαθμολογήσεων, θα προσπαθήσουμε να κάνουμε εκ νέου μια εκτίμηση, για 5 έως 70 εποχές, με τον τρόπο που κάναμε και προηγουμένως, εισάγοντας πολύ μικρά ποσά weight decay, 0.0007, 0.0005 και 0.0004.

Παρατηρούμε πως μειώνοντας την φθίση του σφάλματος μπορούμε να επιτύχουμε όλο και μικρότερη απώλεια, κατά την εκτίμηση του μοντέλου μας. Φυσικά σε αυτό υπάρχει και ένα όριο. Στόχος είναι η εύρεση της κατάλληλης φθίσης η οποία ελαχιστοποιεί το σφάλμα. Με μια μεγάλη φθίση ο αλγόριθμος μας θα συγκλίνει σε ένα υψηλότερο σφάλμα, ενώ με μια μικρή ο αλγόριθμος θα συγκλίνει και έπειτα θα αποκλίνει και πάλι, αυξάνοντας το σφάλμα.



Εικόνα 11 – Εκτίμηση weight decay, βήμα εκπαίδευσης 0.01, batch size 5001, αριθμός παραγόντων 100. Με πράσινο: weight decay = 0.0007, με κόκκινο: weight decay = 0.0005, με μπλε: weight decay = 0.0004

Ταυτόχρονα με την εύρεση της κατάλληλης φθίσης και του μικρότερου δυνατού σφάλματος, θα εξετάσουμε τον αριθμό των παραγόντων του μοντέλου μας. Θεωρητικά, αλλάζοντας τον αριθμό των παραγόντων, θα χρειαστεί να χρησιμοποιήσουμε και διαφορετική φθίση. Για το σκοπό αυτό θα εκτελέσουμε πειράματα με 100 εποχές και θα προσπαθήσουμε να βρούμε το ελάχιστο δυνατό σφάλμα για 10, 20, 40, 50, 80, 100, 200 και 400 παράγοντες.

Τα πειράματα μας απέδωσαν τα εξής αποτελέσματα, με βήμα εκπαίδευσης 0.01 και 100 εποχές, για παράγοντες:

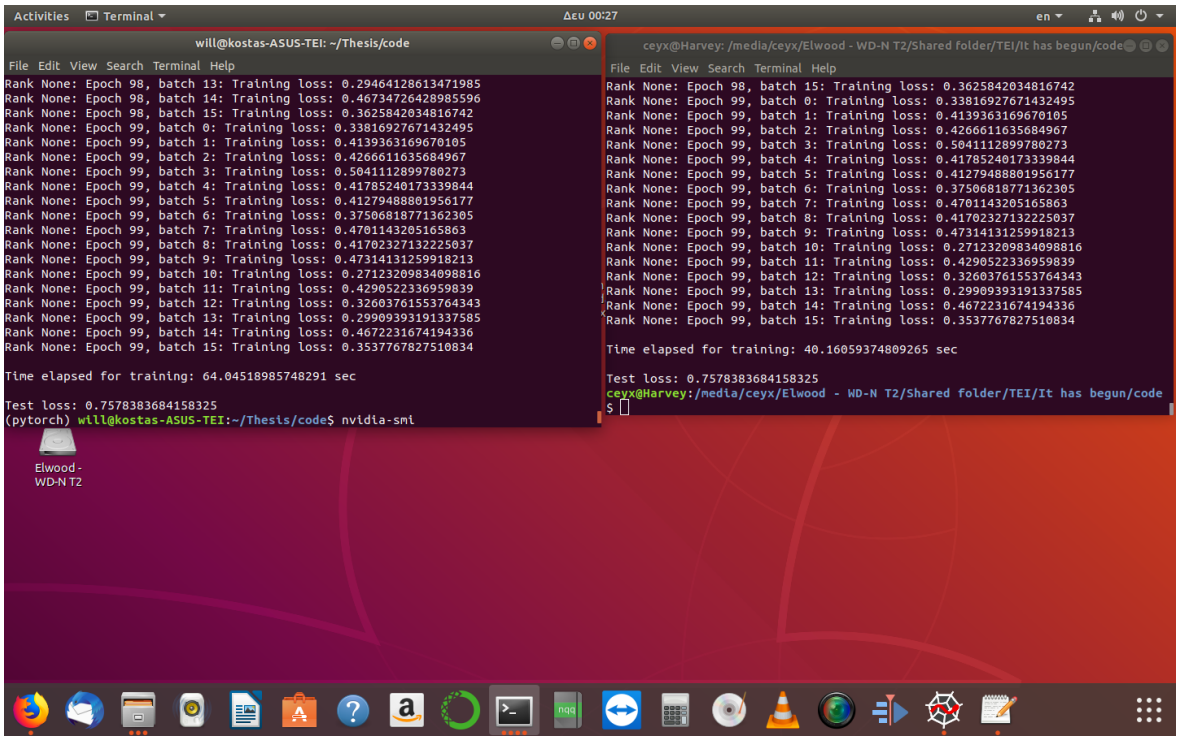
- 10: weight decay=0.00032, mse = 0.7725
- 20: weight decay=0.00032, mse = 0.7640
- 40: weight decay=0.0003, mse = 0.7600
- 50: weight decay=0.00028, mse = 0.7594
- 80: weight decay=0.00028, mse = 0.7587
- 100: weight decay=0.00028, mse = 0.7578
- 200: weight decay=0.00026, mse = 0.7577
- 400: weight decay=0.00026, mse = 0.7572

Παρατηρείται πως όσο περισσότερους παράγοντες διαθέτουμε, τόσο μικρότερη φθίση χρειαζόμαστε για την εύρεση ενός ελαχίστου σφάλματος. Ακόμη, έπειτα από έναν αριθμό 40 παραγόντων το σφάλμα ελαττώνεται όλο και πιο αργά. Θα πρέπει κανείς λοιπόν να αναλογιστεί, όταν δημιουργεί ένα σύστημα συστάσεων παραγοντοποίησης, πόσους παράγοντες να χρησιμοποιήσει για να εξυπηρετηθούν εξίσου οφέλη ακρίβειας αλλά και κατανάλωσης πόρων.

4.2.3 – Αναπαραξιμότητα και παράλληλη κατανεμημένη εκπαίδευση

Αναπαραξιμότητα

Για να αποδείξουμε πως τα αποτελέσματα μας είναι ακριβή, θα επιχειρήσουμε να αναπαράξουμε το αποτέλεσμα των 100 παραγόντων σε έναν άλλο υπολογιστή. Εξάλλου, με ίδια seeds προβλέπεται πως ο διαχωρισμός των δεδομένων και η αρχική κατανομή των παραγόντων θα είναι ακριβώς ίδια. Η εκτέλεση θα γίνει στον υπολογιστή Α και τον υπολογιστή Β.



Εικόνα 12 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, αναπαραξιμότητα και εκτέλεση σε δύο υπολογιστές

Παράλληλη κατανεμημένη εκπαίδευση

Θέλουμε ακόμη να αποδείξουμε πως μια παράλληλη κατανεμημένη υλοποίηση, με δύο διεργασίες, μπορεί να παράξει πανομοιότυπα αποτελέσματα, ασχέτως αν οι διεργασίες εδρεύουν στον ίδιο κόμβο ή σε διαφορετικούς, και πως οι κόμβοι που δημιουργήσαμε, μιας διεργασίας και πολλαπλών διεργασιών, είναι ικανοί να επικοινωνήσουν και να εκπαιδεύσουν ένα κοινό μοντέλο.

Για το σκοπό αυτό δημιουργήσαμε ακόμη ένα script, με όνομα pytorch_SVD_distributed, το οποίο θα δημιουργεί έναν αριθμό διεργασιών, τοπικά, με την χρήση του πακέτου multiprocessing. Σκοπεύουμε να συγκρίνουμε λοιπόν την εκπαίδευση για δύο διεργασίες, με την χρήση αυτού του script, και με παράλληλη κατανεμημένη υλοποίηση, χρησιμοποιώντας τα script των δύο διαφορετικών κόμβων που φτιάξαμε. Για την τοπική εκπαίδευση θα χρησιμοποιήσουμε τον υπολογιστή Α, ενώ οι υπολογιστές Β και Γ θα εκπαιδεύσουν συνεργατικά το μοντέλο μέσα από δίκτυο, ο καθένας τρέχοντας διαφορετικό script. Θα χρησιμοποιήσουμε και πάλι τις παραμέτρους που χρησιμοποιήθηκαν για τον υπολογισμό του ελαχίστου σφάλματος για 100 παράγοντες.

```
#part 1: importing
```

```
#coded methods
from split_selection import input_select, input_tasks
from data_encode import encode, split
from models import SVD
from datasets import CF_Dataset
from training import train_model, test_loss,
set_model_untrained_weights

#libraries
import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
import time #time
import os #operating system
#import sys #system
import copy
#import pickle #pickle
import torch #PyTorch
#import torch.nn as nn #Neural network module
#import torch.nn.functional as F #functions
import torch.multiprocessing as mp #multiprocessing
import torch.distributed as dist #distributed

#sets the number of OpenMP threads to be used by PyTorch to the
maximum.
#the default is half of that amount
#torch.set_num_threads(os.cpu_count())
#torch.set_num_threads(1) #or just one thread...

#reproducibility
torch.manual_seed(0)
np.random.seed(0)

#process initialization method
def init_processes(rank, size, fn, *args, backend='gloo',
**kargs):
    #os.environ['MASTER_ADDR'] = '192.168.1.5'
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    os.environ['WORLD_SIZE'] = str(size)
    os.environ['RANK'] = str(rank)
```

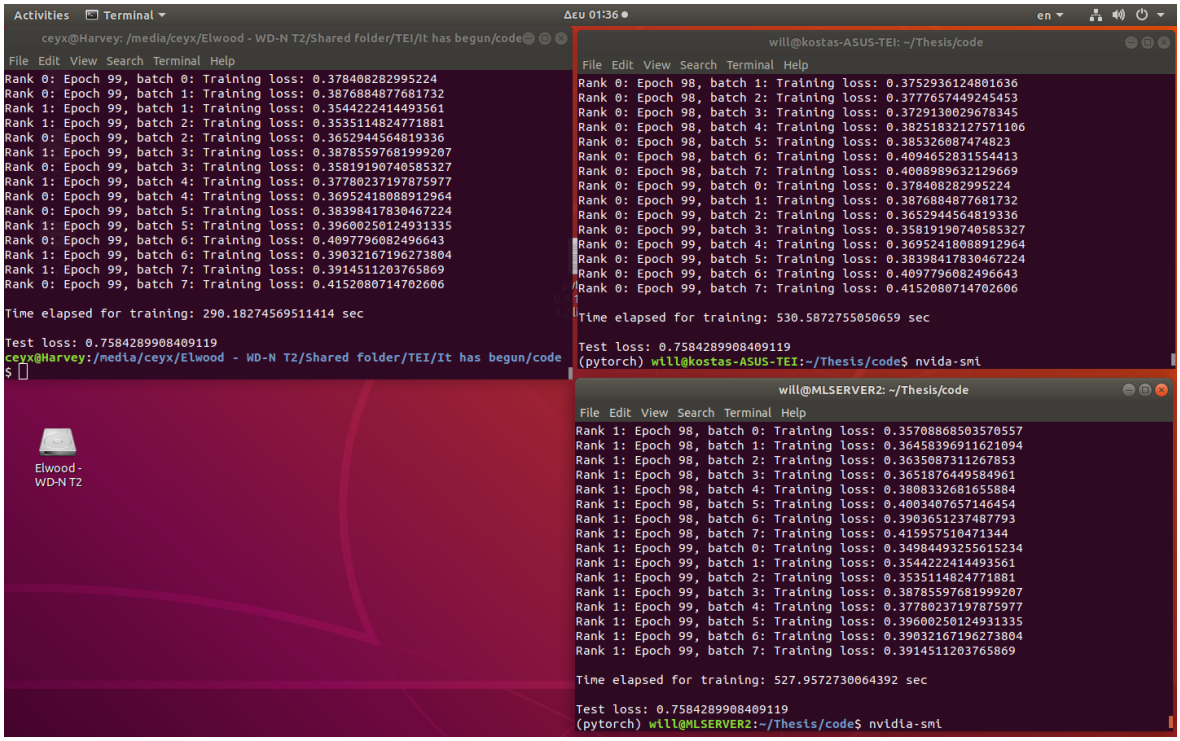
```
    #os.environ['GLOO_SOCKET_IFNAME'] = 'enp3s0'
    #dist.init_process_group(backend,
init_method="tcp://192.168.1.5:29500", world_size=size, rank=rank)
    dist.init_process_group(backend, world_size=size, rank=rank)
    fn(*args, rank=rank, world_size=size, **kargs)


#protection of resources and unrestrained spawning control
if __name__ == "__main__":

    #part 2: preparing data

    #import data
    print(f'\nLoading the data...')
    PATH = Path("../Datasets/ml-latest-small")
    data = pd.read_csv(PATH/"ratings.csv")

    #encode the dataset into unique and contiguous values
    data = encode(data)
    print(f'Dataset has been encoded')

    num_ratings = len(data.rating)
    num_users = len(data.userId.unique())
    num_items = len(data.movieId.unique())
    sparsity = num_ratings/(num_users*num_items)
    mean_rating = data.rating.mean()

    print(f'\nNumber of users: {num_users}')
    print(f'Number of items: {num_items}')
    print(f'Number of ratings: {num_ratings}')
    print(f'Dataset sparsity: {sparsity}')
    print(f'\nAverage rating: {mean_rating}\n')

    #splitting variables input
    method, value = input_select()

    print(f'\nSplitting:\nMethod: {method}\nValue: {value}')

    #dataset splitting
    print('\nSplitting data into training and testing set\n')
    train_set, test_set = split(data, method, value)
```

```
    print(f'\nTraining set: {train_set}\n')
    print(f'\nTesting set: {test_set}\n')

    print(f'\nMaking the torch datasets\n')
    train_dataset = CF_Dataset(train_set)
    test_dataset = CF_Dataset(test_set)

    #torch.save(train_dataset, 'movielens_training_set.pkl')
    #torch.save(test_dataset, 'movielens_test_set.pkl')

    #part 3: modeling

    #initialize the model in cpu()
    #NOTE: be careful not to overload the gpu if it is not
necessary
    model = SVD(num_users, num_items, mean_rating)
    #set the model to cuda for gpu training
    model = model.cuda()
    print(f'Model initialized: {model}')

    #part 4: training

    num_tasks = input_tasks()

    #obligatory training arguments (train_args):
    # #1 model, #2 training dataset
    #train_args = [model, train_dataset]

    #keyworded training arguments (kw_train_args).
    #options include:
    #epochs(10)(int): number of epochs, lr(0.01)(float): learning
    #rate, wd(0.0)(float): weight decay
    #unsqueeze(false)(boolean): add one more dimension to the
    #ratings for the training,
    #cuda(false)(boolean): train model in cuda,
    #distributed_mode(false)(boolean): distributed training,
    #rank(None)(int): distributed training mode only - indicates
    #the rank of the running process,
    #world_size(None)(int): distributed training mode only –
    #indicates the number of tasks/processes
    #participating in the training process
    kw_train_args = {'distributed_mode':True, 'cuda':True}
```

```
    #parallel section
    start = time.time() #start counter
    mp.set_start_method('spawn')
    #use this when start method is fork (default) to
    #share a cpu model with forked processes
    #model.share_memory()
    processes = []
    print('Training is distributed. One model copy per process')
    for rank in range(num_tasks):
        #new model reference for each process
        model = copy.deepcopy(model)
        #reinstate the train arguments
        train_args = [model, train_dataset]
        #arguments
        arguments = [rank, num_tasks, train_model]
        arguments.extend(train_args)
        #key-worded arguments
        kw_arguments = {'wd':0.00028, 'epochs':100}
        kw_arguments.update(kw_train_args)
        #init
        p = mp.Process(target=init_processes, args=arguments,
kwargs=kw_arguments)
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    end = time.time() #end counter

    print(f'\nTime elapsed for training: {end-start} sec')

    #average model weights for untrained users/items
    set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=True)

    #part 5: testing

    test_loss(model, test_dataset, cuda=True)
```

```
#split_selection.py

#returns valid number of tasks for distributed parallel model
#training
def input_tasks(node=False):
    print('\n|--------------------------------|')
    print('\nThis program will distributedly parallelize training
of the given model')
    tasks = input('\n\nPlease insert number of tasks for the
distributed \
training to be split or simply type [d] to use the \
default number of logical processors on your machine \
as the number of tasks:\n\n')
    try:
        if tasks == 'd':
            tasks = os.cpu_count()
        elif int(tasks) > 10:
            print('\nInput number of tasks is potentially
hazardous \
for the system. Please insert a value lower than 10.')
            tasks = input_tasks()
        elif int(tasks) <= 0:
            print('\nNumber of tasks cannot be a negative number!
Again...')
            tasks = input_tasks()
    except ValueError:
        print('\nNot an integer! Again...')
        tasks = input_tasks()
    return int(tasks)
```

Κώδικας 82 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, τοπική παράλληλη κατανεμημένη εκπαίδευση



Εικόνα 13 – Παράλληλη κατανεμημένη υλοποίηση αλγορίθμων συστάσεων, τοπική και απομακρυσμένη παράλληλη κατανεμημένη εκπαίδευση

Παρατηρούμε πως σε σχέση με το σφάλμα της μιας διεργασίας, η παράλληλη κατανεμημένη εκπαίδευση αποφέρει ένα ελαφρώς υψηλότερο σφάλμα. Ο λόγος έγκειται στην τρόπο λειτουργίας του DistributedDataParallel module και της κατανεμημένης εκπαίδευσης. Εάν θυμηθούμε από την ενότητα 3.3.5, το DistributedDataParallel module κάνει τα εξής:

- Το μοντέλο ενθυλακώνεται σε ένα DistributedDataParallel module και δημιουργούνται αντίγραφα στις συσκευές (κάρτες γραφικών) που ορίζουμε σε κάθε διεργασία (παράμετρος devide_ids)
- Forward pass: κάθε αντίγραφο, σε κάθε συσκευή μιας διεργασίας δέχεται ένα κλάσμα του input
- Backward pass:
    - Για κάθε διεργασία: sum gradients στο αρχικό module (module του output_device)
    - Υπολογισμός του μέσου όρου των gradients μεταξύ διαφορετικών αρχικών module / διεργασιών

Το κλειδί στην κατανόηση αυτού του προβλήματος βρίσκεται στην φάση backward. Οι διεργασίες που συμμετέχουν στην εκπαίδευση υπολογίζουν τον μέσο

όρο των gradients μεταξύ τους. Τι σημαίνει πρακτικά αυτό; Εάν έχουμε έναν DataLoader με έναν DistributedSampler, όπως εδώ, το input χωρίζεται ανάμεσα στις διεργασίες, με κάθε διεργασία να λαμβάνει ένα κλάσμα. Αυτό συνεπάγεται πως η κάθε διεργασία θα υπολογίσει τα gradients με διαφορετικό τρόπο, και πως επίσης μπορεί η κάθε διεργασία να μην εκπαιδεύσει για όλους τους χρήστες ή αντικείμενα του batch. Για αυτό το λόγο, όταν ένα αντικείμενο υπάρχει στο input μιας διεργασίας αλλά όχι στο input μιας άλλης, εάν οι διεργασίες που συμμετέχουν είναι δύο, τότε η gradient για αυτό το συγκεκριμένο αντικείμενο ή χρήστη θα είναι μηδέν στην μία, αλλά όχι μηδέν στην άλλη. Αυτό συνεπάγεται λοιπόν, πως κατά τον υπολογισμό της gradient, η gradient θα μειωθεί στη μισή της τιμή.

Έτσι, μπορούμε να φανταστούμε πώς η τιμή της σφάλματος μπορεί να είναι διαφορετική με έναν DistributedSampler. Εάν δεν χρησιμοποιούσαμε DistributedSampler, αλλά εκπαιδεύαμε για κάθε διεργασία με το ίδιο input, τότε επόμενο είναι να έχουμε ίδιο σφάλμα με την μια διεργασία.

Παράλληλη κατανεμημένη εκπαίδευση: ο σκοπός

Αυτός όμως δεν είναι ο σκοπός της παράλληλης κατανεμημένης εκπαίδευσης, εάν μπορούμε να επιτύχουμε το ίδιο αποτέλεσμα, και μάλιστα γρηγορότερα, με μια διεργασία. Ας θυμηθούμε πως εμείς χρησιμοποιήσαμε έναν σταθερό αριθμό torch και numpy seeds, για να προσπαθήσουμε να μιμηθούμε το αποτέλεσμα και να αποδείξουμε πως εκπαιδεύουμε σωστά.

Στην περίπτωση που δεν χρησιμοποιούμε σταθερά seeds, όπως γίνεται και πρακτικά, σκοπός της κατανεμημένης εκπαίδευσης, ταυτόχρονα με την επιτάχυνση που μπορούμε να επιτύχουμε, είναι και το διαφορετικό input από διεργασία σε διεργασία. Λόγος που επιθυμούμε κάτι τέτοιο είναι να μπορέσουμε από κάθε διεργασία να πάρουμε πίσω έναν διαφορετικό gradient tensor, για πολλά διαφορετικά input. Αθροίζοντας αυτά τα input και βρίσκοντας μια μέση τιμή gradient, στοχεύουμε στην καλύτερη γενίκευση και δημιουργία ενός πιο αξιόπιστου μοντέλου.

ΚΕΦΑΛΑΙΟ 5 - Συμπεράσματα και μελλοντική έρευνα

5.1 – Ανασκόπηση

Σε αυτή την εργασία παρουσιάστηκαν αρχικά αλγόριθμοι συστάσεων συνεργατικού φιλτραρίσματος. Έγινε εισαγωγή, πρώτα, στους αλγορίθμους γειτονιών, και έπειτα, στους αλγορίθμους παραγοντοποίησης βασισμένους σε μάθηση. Παρουσιάστηκαν συνοπτικά ακόμη λοιποί αλγόριθμοι, όπως ο ALS, το χρονοσυνειδησιακό μοντέλο, επέκταση των αλγορίθμων παραγοντοποίησης, και αλγόριθμοι γειτονιάς βασισμένοι σε μάθηση.

Ύστερα, έγινε εισαγωγή στην βιβλιοθήκη PyTorch, παρουσιάστηκε υλοποίηση αλγορίθμου συστάσεων με την χρήση της βιβλιοθήκης και αναπτύχθηκε περαιτέρω το πακέτο distributed της PyTorch, και το πως κανείς μπορεί να αναπτύξει, υλοποιήσει και αποσφαλματώσει κατανεμημένες εφαρμογές.

Τέλος, με βάση όσων παρουσιάστηκαν, επιχειρήσαμε να υλοποιήσουμε έναν δικό μας αλγόριθμο παραγοντοποίησης, ονοματικά τον SVD, να εκπαιδεύσουμε ένα μοντέλο με βάση τον αλγόριθμο αυτό με την χρήση πολλαπλών κατανεμημένων διεργασιών, μέσα σε δίκτυο, και να αποδείξουμε την αξιοπιστία του αλγορίθμου αυτού. Ακόμη, επιχειρήσαμε να εκπαιδεύσουμε μοντέλο με τρόπο κατάλληλο, έτσι ώστε ο αλγόριθμος μας να προβλέπει όσο το δυνατόν καλύτερα την βαθμολόγηση χρηστών για ταινίες, και αξιολογήσαμε ένα εύρος αριθμών παραγόντων αντικειμένων και χρηστών.

5.2 – Συμπεράσματα

Τα συστήματα συστάσεων χρησιμοποιούνται σήμερα για να προτείνουν προϊόντα και υπηρεσίες σε πελάτες και συνδρομητές για αγορές, ενοικιάσεις, κρατήσεις κοκ. Ένας πελάτης ή συνδρομητής ονομάζεται χρήστης, ενώ ένα προϊόν ή μια υπηρεσία ονομάζεται αντικείμενο. Σε αυτή την εργασία μελετήθηκαν τα συστήματα συστάσεων συνεργατικού φιλτραρίσματος. Τα συστήματα αυτά αξιοποιούν τη συλλογική δύναμη των ήδη υπαρχόντων βαθμολογήσεων χρηστών αντικειμένων, για να προβλέψουν τη βαθμολόγηση ενός οποιουδήποτε χρήστη για ένα οποιοδήποτε αντικείμενο, αξιολογώντας συχνά ομοιότητες μεταξύ χρηστών και αντικειμένων. [1][2]

Στην εργασία αυτή εξετάσθηκαν δύο είδη συστημάτων συστάσεων, τα συστήματα συστάσεων γειτονιών και τα συστήματα συστάσεων παραγοντοποίησης βασισμένα σε μάθηση, καθένα με τα πλεονεκτήματα του. Από τη μια, τα συστήματα συστάσεων γειτονιών μελετούν συσχετίσεις μεταξύ αντικείμενων και χρηστών και συστήνουν αντικείμενα με βάση παρόμοιων γούστων χρηστών ή σχετικότητας αντικειμένων. Από την άλλη, τα συστήματα συστάσεων παραγοντοποίησης επιχειρούν να προσεγγίσουν όσο το δυνατόν καλύτερα έναν πίνακα προβλεπόμενων βαθμολογήσεων χρηστών για αντικείμενα, με τη χρήση δύο μικρότερων πινάκων παραγόντων και μέσω μιας διαδικασίας μάθησης των πινάκων αυτών. [1][2]

Αλγόριθμοι τέτοιων συστημάτων συστάσεων μπορούν να κωδικοποιηθούν και να υλοποιηθούν με τη χρήση βιβλιοθηκών όπως οι numpy, pandas και scikitlearn της Python. Μια νέα βιβλιοθήκη μηχανικής μάθησης, η PyTorch, αποτελεί μια νέα πρόσθεση στο οικοσύστημα των εργαλείων βαθιάς μάθησης της Python. Βαθιά ενσωματωμένη στην Python, η PyTorch προσφέρει εργαλεία μαθηματικών υπολογισμών πινάκων, παρόμοια με την βιβλιοθήκη numpy, υποστηρίζοντας ταυτόχρονα λειτουργίες επιτάχυνσης με τη χρήση καρτών γραφικών γενικού σκοπού. Παρέχει ακόμα και άλλα εργαλεία, με έτοιμες δομές νευρωνικών στρωμάτων, Datasets και δυνατότητες παράλληλης κατανεμημένης επικοινωνίας διεργασιών.

Μιλώντας για δυνατότητες παράλληλης κατανεμημένης επικοινωνίας και επεξεργασίας, το πακέτο distributed της PyTorch υποστηρίζει την επικοινωνία μεταξύ διαφορετικών διεργασιών που μπορεί να εδρεύουν, ή μη, σε διαφορετικές τοποθεσίες, εντός επικοινωνούντων δικτύων. Μπορεί κανείς να καταλάβει τη χρησιμότητα ενός τέτοιου εργαλείου, εάν αναλογιστεί ένα σενάριο εκπαίδευσης με τη χρήση ενός cluster υπολογιστών. Προβλέποντας πιθανές ανάγκες για χρήση σε παραγωγή, το πακέτο distributed υποστηρίζει λειτουργίες επικοινωνίας μεταξύ τοπικά απομονωμένων διεργασιών. Παράλληλα, το πακέτο multiprocessing (επέκταση της βιβλιοθήκης multiprocessing της Python) παρέχει κατάλληλα εργαλεία για τη δημιουργία τοπικών διεργασιών. Χρησιμοποιώντας τα δύο αυτά πακέτα, μπορεί κανείς να επιτύχει την επικοινωνία, όχι μόνο τοπικά απομονωμένων διεργασιών, αλλά και μεταξύ διεργασιών που εδρεύουν στο ίδιο μηχάνημα.

Ένας αλγόριθμος λοιπόν σύστασης μπορεί να υλοποιηθεί, σχετικά εύκολα, με την χρήση της PyTorch, εκμεταλλευόμενος τις δυνατότητες επιτάχυνσης της PyTorch με τη χρήση καρτών γραφικών γενικού σκοπού, αλλά και τις δυνατότητες παράλληλης κατανεμημένης επικοινωνίας, για την εκπαίδευση μέσα σε ένα cluster υπολογιστών. Εντέλει, κάτι τέτοιο μπορεί να αποδειχθεί εξαιρετικά χρήσιμο, αφού για ιδιαίτερα μεγάλα δεδομένα συστημάτων συστάσεων, όπως είναι του YouTube και του Netflix, η υπολογιστική ισχύ ενός μόνο μηχανήματος δεν θεωρείται επαρκής και χρειάζεται να αξιοποιηθεί ένας μεγάλος αριθμός υπολογιστών σε cluster.

5.3 – Μελλοντικές επεκτάσεις

Με βάση την ύλη που καλύφθηκε σε αυτή την πτυχιακή εργασία, θα μπορούσαμε να επεκταθούμε περαιτέρω όσον αφορά:

1. Τη δημιουργία αλγορίθμων γειτονιών βασισμένες σε μάθηση
2. Την επέκταση του υλοποιημένου αλγορίθμου παραγοντοποίησης
3. Τον πειραματισμό με διαφορετικά datasets

Μέθοδοι γειτονιών βασισμένες σε μάθηση

Αναφερθήκαμε σύντομα στην ενότητα 2.4 πώς μέθοδοι γειτονιών μπορούν να μετατραπούν σε μεθόδους βασισμένες σε μάθηση. Υπάρχουν τρεις τρόποι για να το επιτύχει κανείς αυτό [1][2][9]:

- Υπολογισμός πίνακα ομοιοτήτων με μάθηση: Ο πρώτος και απλούστερος τρόπος είναι η προσέγγιση του πίνακα ομοιοτήτων με τη χρήση μάθησης, αντί της χρήσης μέτρων ομοιοτήτων. Προσπαθούμε λοιπόν, αντί να υπολογίσουμε ολόκληρο τον πίνακα μονομιάς, να ελαχιστοποιήσουμε μια συνάρτηση σφάλματος, προσπαθώντας να προσεγγίσουμε όσο καλύτερα γίνεται την πρόβλεψη με διαδοχικές διορθώσεις στα βάρη του πίνακα ομοιότητας.
- Παραγοντοποίηση πίνακα ομοιοτήτων: Ο δεύτερος τρόπος είναι η παραγοντοποίηση του πίνακα ομοιοτήτων, με τον τρόπο που υλοποιήσαμε τον αλγόριθμο SVD. Δημιουργούμε έτσι δύο πίνακες παραγόντων, οι οποίοι πολλαπλασιαζόμενοι παράγουν τον αρχικό πίνακα ομοιοτήτων. Στόχος είναι ο υπολογισμός των παραγόντων των δύο νέων

πινάκων. έτσι ώστε να ελαχιστοποιήσουμε μια συνάρτηση σφάλματος πρόβλεψης.

- Χρήση μικτών μεθόδων γειτονιών και μεθόδων μάθησης: Οι μέθοδοι αυτές χρησιμοποιούν μια μικτή υλοποίηση. Για τον υπολογισμό της πρόβλεψης χρησιμοποιούνται μικτά μια μέθοδος γειτονιάς και μια μέθοδος βασισμένη σε μάθηση.

Μια πιθανή επέκταση θα ήταν λοιπόν να προσπαθήσουμε να υλοποιήσουμε έναν ακόμη αλγόριθμο ο οποίος, παρόμοια με τον SVD, θα παραγοντοποιούσε τον πίνακα ομοιοτήτων μιας μεθόδου γειτονιάς σε δύο πίνακες παραγόντων, τους οποίους παράγοντες έπειτα θα επιχειρούσαμε να υπολογίσουμε.

Επέκταση αλγορίθμου

Όσον αφορά την επέκταση του υπάρχοντος αλγορίθμου που δημιουργήσαμε, έχουμε τις εξής επιλογές [1][2]:

- Χρήση διαφορετικών μετρικών σφάλματος: Πέρα από την mean squared error, θα μπορούσαμε να επεκτείνουμε τις μεθόδους εκπαίδευσης και αξιολόγησης για να χρησιμοποιούν επιπλέον μετρικές ως επιλογή. Μερικές από τις μετρικές που θα μπορούσαμε ακόμη να χρησιμοποιήσουμε περιλαμβάνουν την regularized squared error, root mean squared error, mean absolute error και τις κανονικοποιημένες normalized mean absolute και normalized root mean squared error.
- Χρήση έμμεσης ανατροφοδότησης (implicit feedback): Μια άλλη επέκταση θα μπορούσε να περιλαμβάνει την χρήση implicit feedback. Ένας απλός τρόπος να υλοποιηθεί κάτι τέτοιο είναι η χρήση ενός ακόμη Embedding layer το οποίο θα περιλαμβάνει παράγοντες προτίμησης, η οποία προτίμηση θα εκφράζεται από την επιλογή του χρήστη να βαθμολογήσει ένα αντικείμενο. Συνυπολογίζοντας τους παράγοντες αυτούς, θα μπορούσαμε να κάνουμε τον υπολογισμό της πρόβλεψης ακόμη πιο ακριβή.
- Εισαγωγή παράγοντα παλαιότητας (time aware model): Μια άλλη τακτική θα ήταν η εισαγωγή ενός παράγοντα παλαιότητας για τις πολώσεις των βαθμολογήσεων. Στο dataset που χρησιμοποιήσαμε, πέρα από τα ratings, δίνονται ακόμη timestamps τα οποία μας ενημερώνουν για την χρονολογία

της κάθε βαθμολόγησης. Λαμβάνοντας υπόψη αυτά και δίνοντας διαφορετικά βάρη πόλωσης βαθμολόγησης, ανάλογα με την χρονολογία, μπορούμε να επιτύχουμε μια ακόμη καλύτερη πρόβλεψη.

Πειραματισμός με άλλα datasets

Στο διαδίκτυο μπορεί να βρει κανείς μια μεγάλη πληθώρα από έτοιμα datasets. Πέρα από το MovieLens dataset των 100 χιλιάδων ratings, υπάρχουν ακόμη μεγαλύτερα datasets που μπορεί κανείς να χρησιμοποιήσει. Το MovieLens, ένα σύστημα συστάσεων ταινιών, παρέχει, πέρα του dataset των 100 χιλιάδων, άλλα datasets μεγέθους 1, 10, 20 και 27 εκατομμυρίων ratings. Πέρα από ταινίες όμως μπορεί κανείς να βρει και άλλα είδους datasets, όπως πχ. το Jester, ένα dataset συστάσεων ανεκδότων, το WikiLens dataset, ένα dataset συστάσεων άρθρων της Wikipedia, και το Last.fm dataset, ένα dataset συστάσεων μουσικής. [6]

Αντί λοιπόν για ταινίες, θα μπορούσαμε να εκπαιδεύσουμε ένα μοντέλο που θα προέβλεπε την βαθμολόγηση ενός χρήστη για ένα μουσικό κομμάτι, για παράδειγμα.

Όλα αυτά είναι πιθανά. Η βιβλιοθήκη PyTorch αποτελεί ένα ισχυρό εργαλείο με το οποίο οποιοσδήποτε από αυτούς τους αλγορίθμους είναι δυνατό να υλοποιηθεί και να εκπαιδευθεί, όχι μόνο με τη χρήση μιας διεργασίας, αλλά και με περισσότερες, με τη χρήση του πακέτου distributed.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Francesco Ricci – Lior Rokach – Bracha Shapira, Recommender Systems Handbook – 2nd Edition, Springer, 2015

[2] Charu C. Aggarwal, Recommender Systems – The Textbook, Springer, 2016

[3] Intro to Recommender Systems: Collaborative Filtering, https://www.ethanrosenthal.com/2015/11/02/intro-to-collaborative-filtering/

[4] Numpy documentation https://docs.scipy.org/doc/numpy/reference/

[5] Pandas documentation https://pandas.pydata.org/pandas-docs/stable/

[6] GroupLens datasets https://grouplens.org/datasets

[7] Seaborn API https://seaborn.pydata.org/api.html

[8] matplotlib.pyplot https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html

[9] Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model, Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 426-434, 2008

[10] Ning, Xia & Karypis, George, SLIM: Sparse Linear Methods for Top-N Recommender Systems, 11th IEEE International Conference on Data Mining, pp. 497-506, 2011

[11] Nikhil Ketkar, Deep Learning with Python – A Hands-on Introduction, Chapter 12, Apress, 2017

[12] Learning PyTorch with Examples https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

[13] torch.Tensor https://pytorch.org/docs/stable/tensors.html

[14] Automatic differentiation package - torch.autograd https://pytorch.org/docs/stable/autograd.html

[15] Autograd mechanics https://pytorch.org/docs/stable/notes/autograd.html

[16] torch.optim https://pytorch.org/docs/stable/optim.html

[17] torch.nn.functional https://pytorch.org/docs/stable/nn.html#torch-nn-functional

[18] Extending PyTorch https://pytorch.org/docs/stable/notes/extending.html

[19] torch.nn https://pytorch.org/docs/stable/nn.html

[20] Intro to Pytorch https://github.com/yanneta/pytorch-tutorials/blob/master/intro-to-pytoch.ipynb

[21] torch.utils.data https://pytorch.org/docs/stable/data.html

[22] Data Loading and Processing Tutorial
https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

[23] Saving and Loading Models
https://pytorch.org/tutorials/beginner/saving_loading_models.html

[24] torch.cuda https://pytorch.org/docs/stable/cuda.html

[25] CUDA semantics https://pytorch.org/docs/stable/notes/cuda.html

[26] Frequently Asked Questions https://pytorch.org/docs/stable/notes/faq.html

[27] Reproducibility https://pytorch.org/docs/stable/notes/randomness.html

[28] Collaborative Filtering with Neural Networks
https://github.com/yanneta/pytorch-tutorials/blob/master/collaborative-filtering-nn.ipynb
https://www.youtube.com/watch?v=10qVo3kxAhI
https://www.youtube.com/watch?v=vrpbDpf4y98

[29] Matrix Factorization in PyTorch,
https://www.ethanrosenthal.com/2017/06/20/matrix-factorization-in-pytorch/

[30] Python threading https://docs.python.org/2/library/threading.html

[31] Python Queue https://docs.python.org/2/library/queue.html

[32] Writing Distributed Applications with PyTorch
https://pytorch.org/tutorials/intermediate/dist_tuto.html

[33] Distributed communication package - torch.distributed
https://pytorch.org/docs/stable/distributed.html

[34] Multiprocessing package - torch.multiprocessing
https://pytorch.org/docs/stable/multiprocessing.html

[35] Python multiprocessing https://docs.python.org/2/library/multiprocessing.html

[36] Multiprocessing best practices
https://pytorch.org/docs/stable/notes/multiprocessing.html

[37] NCCL environment variables https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/env.html

[38] PyTorch 1.0 Distributed Trainer with Amazon AWS
https://pytorch.org/tutorials/beginner/aws_distributed_training_tutorial.html

ΠΑΡΑΡΤΗΜΑΤΑ

Αλγόριθμος γειτονιάς σε Python

```
#python_neighborhood-based_example.py

#neighbourhood-based algorithm

#part 1: importing

#libraries
import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
import time #time
import matplotlib.pyplot as plt #plotting
import seaborn as sns #data visualization based on matplotlib

#methods
from helper import train_test_split, fast_similarity,
predict_topk_nobias, get_mse

#part 2: preparing data

PATH = Path('..\..\Datasets\ml-100k')

#file does not contain headers
#we append them ourselves
names = ['user_id', 'item_id', 'rating', 'timestamp']
#data seperated by tabs, set headers to the names list
df = pd.read_csv(PATH/'u.data', sep='\t', names=names)
print(df.head()) #get the first 5

#number of unique users, items
n_users = df.user_id.unique().shape[0]
n_items = df.item_id.unique().shape[0]
print(f'\n{n_users} users')
print(f'{n_items} items')

#make the ratings matrix
ratings = np.zeros((n_users, n_items))
#make the ratings matrix
#we map user/item IDs to user/item indices by removing the
```

```
#"Python starts at 0" offset between them
for row in df.itertuples():
    ratings[row[1]-1, row[2]-1] = row[3]
print(f"\n {ratings}")

#sparsity
#(get non-zero ratings)
sparsity = float(len(ratings.nonzero()[0]))
sparsity /= (ratings.shape[0]*ratings.shape[1])
#percentage
sparsity *= 100
print(f'\nRatings sparsity: {sparsity}%')

#make the training and test sets
train, test = train_test_split(ratings)

#part 3: make the similarities table

start = time.time()
user_similarity = fast_similarity(train)
item_similarity = fast_similarity(train, kind = 'item')
end = time.time()
print(f'\nTime elapsed for measuring cosine \
similarity with fast method: {end-start}')

#print the first four in the item similarity table
print (f'\nitem_similarity[:4, :4]: {item_similarity[:4, :4]}')

#part 4: make the predictions table

start = time.time()
item_prediction = predict_topk_nobias(train, item_similarity,
kind='item')
user_prediction = predict_topk_nobias(train, user_similarity)
end = time.time()
print(f'\nTime elapsed for predicting \
similarity with Top-k method with bias: {end-start}')

#part 5: testing

print(f'\nUser-based CF MSE: {get_mse(user_prediction, test)}')
print(f'Item-based CF MSE: {get_mse(item_prediction, test)}')
```

```
#let's try a different amount of top-k neighbours
#test (really slow). 50 seems to be optimal amount of top-k
neighbors

k_array = [5, 15, 30, 50, 100, 200]
user_train_mse = []
user_test_mse = []
item_test_mse = []
item_train_mse = []

for k in k_array:
    #top k predictions
    user_pred = predict_topk_nobias(train, user_similarity,
kind='user', k=k)
    item_pred = predict_topk_nobias(train, item_similarity,
kind='item', k=k)

    #sum the mse for each:
    #user
    user_train_mse += [get_mse(user_pred, train)]
    user_test_mse += [get_mse(user_pred, test)]
    #item
    item_train_mse += [get_mse(item_pred, train)]
    item_test_mse += [get_mse(item_pred, test)]

#make a seaborn set
sns.set()

#pick a color palette to draw from (Set2 palette, 2 colors
selected)
pal = sns.color_palette("Set2", 2)

plt.figure(figsize=(8,8)) #8x8
#draw the following
#alpha (intensity) = 0.5
plt.plot(k_array, user_train_mse, c=pal[0], label='User-based
train', alpha=0.5, linewidth=5)
plt.plot(k_array, user_test_mse, c=pal[0], label='User-based
test', linewidth=5)
plt.plot(k_array, item_train_mse, c=pal[1], label='Item-based
train', alpha=0.5, linewidth=5)
```

```
plt.plot(k_array, item_test_mse, c=pal[1], label='Item-based
test', linewidth=5)
#draw the legend at the best location, fonts: 20px
plt.legend(loc='best', fontsize=20)
#make the ticks as following
plt.xticks(fontsize=16);
plt.yticks(fontsize=16);
#and the labels
plt.xlabel('k', fontsize=30);
plt.ylabel('MSE', fontsize=30);
plt.show()


#helper.py

import numpy as np
from sklearn.metrics import mean_squared_error #mse

#|-------------------------------------------------------------
-----------------------
#dataset splitting methods

#split dataset to training and test set
def train_test_split(ratings):
    test = np.zeros(ratings.shape)
    train = ratings.copy()
    #for each user
    for user in range(ratings.shape[0]):
        #select 10 random ratings of a user for an item,
        #without replacement
        test_ratings = np.random.choice(ratings[user,
:].nonzero()[0],
                                        size=10,
                                        replace=False)
        #exclude those ratings from the training set
        train[user, test_ratings] = 0.
        #append them to the test set
        test[user, test_ratings] = ratings[user, test_ratings]
    #assert (test) if the training and test sets contain
    #common ratings (they should not)
    assert(np.all((train*test == 0)))
    return train, test #return the test and training set arrays
```

```
#|-------------------------------------------------------------------
------------------------
#cosine similarity methods

#very slow. not recomended
def slow_similarity(ratings, kind='user'):
    #user case
    if kind == 'user':
        axmax = 0
        axmin = 1
    #item case
    elif kind == 'item':
        axmax = 1
        axmin = 0
        #make the similarity matrix
        #(item*item or user*user depending on the case)
    sim = np.zeros((ratings.shape[axmax],
                    ratings.shape[axmax]))
    #for every user or item
    for u in range(ratings.shape[axmax]):
        #for every user or item (again)
        for uprime in range(ratings.shape[axmax]):
            #sum factors
            rui_sqrd = 0.
            ruprimei_sqrd = 0.
            #for each column or row (depending on kind)
            for i in range(ratings.shape[axmin]):
                #numerator
                sim[u, uprime] = ratings[u, i] * ratings[uprime,
i]
                #denominator root sums
                rui_sqrd += ratings[u,i] ** 2
                ruprimei_sqrd += ratings[uprime, i] ** 2
            #fill the similarity matrix for user or item
            sim[u, uprime] /= np.sqrt(rui_sqrd*ruprimei_sqrd)
    return sim

#fast
#epsilon: very small number for handling divided-by-zero errors
def fast_similarity(ratings, kind='user', epsilon=1e-9):
    #user case
```

```
    if kind == 'user':
        sim = ratings.dot(ratings.T) + epsilon
    #item case (transposed case scenario)
    elif kind == 'item' :
        sim = ratings.T.dot(ratings) + epsilon
    #make the norms
    norms = np.array([np.sqrt(np.diagonal(sim))])
    return (sim/norms/norms.T)

 #|-------------------------------------------------------------
-------------------------
#prediction methods

#normalized sum of cosine similarities (weights)
#multiplied by the ratings

#slow simple
def predict_slow_simple(ratings, similarity,
                        kind = 'user'):
    #make the prediction table
    #(same shape as the ratings table)
    pred = np.zeros(ratings.shape)
    #user case
    if kind == 'user':
        #for each user
        for i in range(ratings.shape[0]):
            #for each user (again)
            for j in range(ratings.shape[1]):
                #make the prediction
                pred[i,j] = similarity[i, :].dot(\
                    ratings[:,j])/np.sum(\
                           np.abs(similarity[i,:]))
   #item case
    elif kind == 'item':
        #for each user
        for i in range(ratings.shape[0]):
            #for each user (again)
            for j in range(ratings.shape[1]):
                #make the prediction (in reverse)
                pred[i,j] = similarity[j, :].dot(\
                    ratings[i,:].T)/np.sum(\
                           np.abs(similarity[j,:]))
```

```
#fast simple
def predict_fast_simple(ratings, similarity,
                        kind = 'user'):
    #user case
    if kind == 'user':
        #dot product of similarities (weights)
        #multiplied with the ratings divided by
        #the sum of the absolute values of the
        #cosine similarities (weights)
        return similarity.dot(ratings)/\
               np.array([np.abs(similarity)\
                         .sum(axis=1)]).T
    #item case
    elif kind == 'item':
        #the same but transposed
        return ratings.dot(similarity)/\
               np.array([np.abs(similarity)\
                         .sum(axis=1)])

#top-k neighbours prediction
def predict_topk(ratings, similarity, kind='user', k=40):
    #make the prediction matrix
    pred = np.zeros(ratings.shape)
    #user case
    if kind == 'user':
        for i in range(ratings.shape[0]):
            #top k users
            top_k_users = [np.argsort(similarity[:,i])[:-k-1:-1]]
            for j in range(ratings.shape[1]):
                #make the prediction for the user
                pred[i,j] =
similarity[i,:][top_k_users].dot(ratings[:,j][top_k_users])
                pred[i, j] /= np.sum(np.abs(similarity[i,
:][top_k_users]))
    #item case
    if kind == 'item':
        for j in range(ratings.shape[1]):
            #top k items
            top_k_items = [np.argsort(similarity[:,j])[:-k-1:-1]]
            for i in range(ratings.shape[0]):
                #make the prediction for the item
```

```
                pred[i, j] = similarity[j,
:][top_k_items].dot(ratings[i, :][top_k_items].T)
                pred[i, j] /= np.sum(np.abs(similarity[j,
:][top_k_items]))
    return pred

#prediction with bias
def predict_nobias(ratings,similarity, kind='user'):
    #user case
    if kind == 'user':
        #mean rating is bias
        user_bias = ratings.mean(axis=1)
        #calculate the ratings again based on bias
        ratings = (ratings - user_bias[:, np.newaxis].copy())
        #make the prediction, then add bias
        #(h()*h()^-1 golden rule)
        pred =
similarity.dot(ratings)/np.array([np.abs(similarity).sum(axis=1)])
.T
        pred += user_bias[:, np.newaxis]
    #item case
    elif kind == 'item':
        #same as before
        item_bias = ratings.mean(axis=0)
        ratings = (ratings - item_bias[np.newaxis, :]).copy()
        pred = ratings.dot(similarity) /
np.array([np.abs(similarity).sum(axis=1)])
        pred += item_bias[np.newaxis, :]
    return pred

#top-k prediction with bias
def predict_topk_nobias(ratings, similarity, kind='user', k=40):
    #make the prediction matrix
    pred = np.zeros(ratings.shape)
    #user case
    if kind == 'user':
        #mean bias
        user_bias = ratings.mean(axis=1)
        #remove bias
        ratings = (ratings - user_bias[:, np.newaxis]).copy()
        #for each user
        for i in range(ratings.shape[0]):
```

```
            #get the top-k users
            top_k_users = [np.argsort(similarity[:,i])[:-k-1:-1]]
            for j in range(ratings.shape[1]):
                #make the prediction for top-k neighbours
                pred[i, j] = similarity[i,
:][top_k_users].dot(ratings[:, j][top_k_users])
                pred[i, j] /= np.sum(np.abs(similarity[i,
:][top_k_users]))
        #add the bias
        #(h()*h()^-1 golden rule)
        pred += user_bias[:, np.newaxis]
        #item case
    if kind == 'item':
        #same as before
        item_bias = ratings.mean(axis=0)
        ratings = (ratings - item_bias[np.newaxis, :]).copy()
        for j in range(ratings.shape[1]):
            top_k_items = [np.argsort(similarity[:,j])[:-k-1:-1]]
            for i in range(ratings.shape[0]):
                pred[i, j] = similarity[j,
:][top_k_items].dot(ratings[i, :][top_k_items].T)
                pred[i, j] /= np.sum(np.abs(similarity[j,
:][top_k_items]))
        pred += item_bias[np.newaxis, :]
    return pred

#|-------------------------------------------------------------------
-----------------------
#test loss methods

#mean squared error
def get_mse(pred, actual):
    #make the prediction table for the given ratings
    pred = pred[actual.nonzero()].flatten()
    #make the evaluation table for the given ratings
    actual = actual[actual.nonzero()].flatten()
    return mean_squared_error(pred, actual)
```

Λειτουργίες παράλληλης κατανεμημένης επικοινωνίας σε PyTorch

```
#pytorch_distributed.py

import os
```

```
import torch
import torch.distributed as dist
#from torch.multiprocessing import Process
import torch.multiprocessing as mp

#**READ THIS BEFORE DOING ANYTHING ELSE**: DO NOT initialize
#ANYTHING in cuda beforehand if you intend to work with
#multiprocessing on cuda. initializing variables in cuda within
#the forked process will cause the program to crash if
#variables have already been initialized outside the processes.
#DO NOT attempt this practice!!!
#Make sure there is nothing on the GPU memory before proceeding
#to initiazize anything in cuda within the processes

#*ADVICE*: upon debugging distributed applications, if the
#program stops unexcepectedly or joins processes without any
#output, run the script using the external console instead of
#the native Spyder console. error messages **CAN BE HIDDEN**.
#similar problems with hidden error messages can infrequently
#occur in other occasions as well

#*NOTICE*: gloo is currently being used as the primary backend
#because of compatibility reasons.
#due to its nature (works on both CPU and GPU)
#tensors can also be cast to CUDA devices in broadcast and
#all_reduce functions.

#functions to be used
#nccl only supports broadcast, all_reduce,
#reduce and all_gather

#Send/Receive
#send/recv not supported by nccl!
def send_rcv(rank, size):
    """ Distributed function to be implemented later. """
    print(f'Hello from task no. {rank}!')
    tensor = torch.zeros(1)
    if rank == 0:
        print('I increase the tensor value by 1\
 and send it to process 1!')
        tensor += 1
        dist.send(tensor=tensor, dst=1)
```

```
    else:
        print('And I receive it!')
        dist.recv(tensor=tensor, src=0)
    print(f'My tensor has this data: {tensor[0]}')
    #pass

#isend/ircv (asynchronous)
#send/recv not supported by nccl!
def isend_ircv(rank, size):
    """ Distributed function to be implemented later. """
    print(f'Hello from task no. {rank}!')
    tensor = torch.zeros(1)
    req = None
    if rank == 0:
        print('I increase the tensor value by 1\
 and send it to process 1!')
        tensor += 1
        req = dist.isend(tensor=tensor, dst=1)
        print('Task 0 started sending')
    else:
        print('And I receive it!')
        req = dist.irecv(tensor=tensor, src=0)
        print('Task 1 started receiving')
    #wait to receive/send data
    req.wait()
    print(f'Tasks synchonized!')
    print(f'My tensor has this data: {tensor[0]}')
    #pass

#all_reduce
def all_reduce(rank, size):
    print('Hello from task ', rank, '!')
    #create a new group with the following
    #processes
    #Notice: all processes belonging in the
    #distributed job MUST be entered, even if
    #they are not to be used.
    #Groups should also be created in the same
    #order in all processes
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
```

```
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #everybody in the group do an all reduce
    #operation summing up the tensors received
    #by everyone else
    dist.all_reduce(tensor=tensor, op=dist.reduce_op.SUM,
group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])


#more functions

#supported by nccl
def broadcast(rank, size):
    print('Hello from task ', rank, '!')
    #create a new group with the following
    #processes
    #Notice: all processes belonging in the
    #distributed job MUST be entered, even if
    #they are not to be used.
    #Groups should also be created in the same
    #order in all processes
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #everybody in the group stores the received tensor inside the
tensor
    #Notice: the tensor should be of the SAME DIMENSIONS for each
process
    dist.broadcast(tensor=tensor, src=0, group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])

def reduce(rank, size):
    print('Hello from task ', rank, '!')
    #create a new group with the following
    #processes
    #Notice: all processes belonging in the
    #distributed job MUST be entered, even if
```

```
    #they are not to be used.
    #Groups should also be created in the same
    #order in all processes
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #rank 0 does a reduce operation,
    #summing up the tensors received
    #by everyone else
    dist.reduce(tensor=tensor, dst=0, op=dist.reduce_op.SUM,
group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])

def all_gather(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #the tensor list should contain correctly sized
    #tensors, matching in dimensions every tensor
    #that is to be received by the processes
    tensor_list = [tensor, tensor]
    #everybody in the group does a gather operation,
    #storing the tensors received
    #by everyone else inside the tensor list
    dist.all_gather(tensor_list=tensor_list, tensor=tensor,
group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])


#not supported by nccl
def gather(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
```

```
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #rank 0 does a gather operation,
    #storing the tensors received
    #by everyone else inside the tensor list
    if rank == 0:
        #the tensor list should contain correctly sized
        #tensors, matching in dimensions every tensor
        #that is to be received by the processes
        tensor_list = [tensor, tensor]
        dist.gather(tensor=tensor, gather_list=tensor_list,
group=group)
    else:
        dist.gather(tensor=tensor, dst=0, group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])


def scatter(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    #rank 0 does a scatter operation,
    #distributing the tensors inside the list
    #to every process inside the group
    if rank == 0:
        tensor_list = [tensor, tensor]
        dist.scatter(tensor=tensor, scatter_list=tensor_list,
group=group)
    else:
        dist.scatter(tensor=tensor, src=0, group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])

def barrier(rank, size):
    print('Hello from task ', rank, '!')
    group = dist.new_group([0,1])
    #make a simple tensor
    tensor = torch.ones(1)#.cuda()
```

```
    print('Before:')
    print('Rank ', rank, ' has data ', tensor[0])
    tensor += 1
    #wait all processes to reach the following line
    dist.barrier(group=group)
    print('After:')
    print('Rank ', rank, ' has data ', tensor[0])


#initialize processes
#declare rank, size of group, function to be run
#and a backend.

#master address, port and world size can be declared outside
#of the processes as well (global)
os.environ['MASTER_ADDR'] = '127.0.0.1'
os.environ['MASTER_PORT'] = '29500'
os.environ['WORLD_SIZE'] = str(2)

#backend in gloo because nccl times out in Kepler 3.0 even when
#built from source...
def init_processes(rank, size, fn, backend='gloo'):
    """ Initialize the distributed environment. """
    #os.environ['MASTER_ADDR'] = '127.0.0.1'
    #os.environ['MASTER_PORT'] = '29500'
    #os.environ['WORLD_SIZE'] = str(size)
    os.environ['RANK'] = str(rank)
    dist.init_process_group(backend, world_size=size, rank=rank)
    fn(rank, size)


#spawn and forkserver methods don't work inside functions...
#def do():
#it is a best practice to use the if __name__=='main' section.
#start methods like spawn and forkserver will produce a copy
#of the process's parent resulting in continuous process
#creation and a runtime error. using the if section ensures
#that no process creation will occur any further, since the
#__name__ variable's value will not be 'main' in the created
#process but a script name instead
if __name__ == "__main__":
    #__name__ will refer to the function and not the script
```

```
    #if set within a function.
    #the functions called below will not belong to the
    #main function therefore initialization will fail with
    #spawn and forkserver methods.
    #also, despite expected outcome, fork method (default)
    #will work with cuda tensors using distributed operations!!
    #if there is no specific worry about sharing tensors
    #otherwise with multiprocessing, it seems obvious that
    #any start method will produce similar results with the
    #distributed package operations.
    #furthermore, it is a best practice to set the start
    #method within the if __name__=='main' section.
    #the spawn method can also be set only *once* inside
    #a **main** script!!!
    #mp.set_start_method('fork') #default
    #or
    #mp.set_start_method('spawn')
    #or
    #mp.set_start_method('forkserver')
    #otherwise, you can create a context object using a
    #specified start method
    #ctx = mp.get_context('spawn')
    size = 2
    processes = []
    for rank in range(size):
        print('Initializing task ', rank)
        #p = ctx.Process(target=init_processes, args=(rank, size,
all_reduce))
        p = mp.Process(target=init_processes, args=(rank, size,
all_reduce))
        print('Run process of task ', rank)
        p.start()
        processes.append(p)

    print('\nActive processes before joining: ',
mp.active_children())
    print('Joining ', len(mp.active_children()) ,' processes')
    for p in processes:
        p.join()

    print('Active processes after joining: ',
mp.active_children())
```

Αλγόριθμος παραγοντοποίησης σε PyTorch, απλή εκπαίδευση

```
#pytorch_SVD.py

#part 1: importing

#coded methods
from split_selection import input_select
from data_encode import encode, split
from models import SVD
from datasets import CF_Dataset
from training import train_model, test_loss,
set_model_untrained_weights

#libraries
import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
import time #time
#import os #operating system
#import sys #system
import torch #PyTorch
#import torch.nn as nn #Neural network module
#import torch.nn.functional as F #functions

#sets the number of OpenMP threads to be used by PyTorch to the
maximum.
#the default is half of that amount
#torch.set_num_threads(os.cpu_count())
#torch.set_num_threads(1)

#reproducibility
torch.manual_seed(0)
np.random.seed(0)

#part 2: preparing data

#import data
print(f'\nLoading the data...')
PATH = Path("../Datasets/ml-latest-small")
data = pd.read_csv(PATH/"ratings.csv")
```

```
#encode the dataset into unique and contiguous values
data = encode(data)
print(f'Dataset has been encoded')

num_ratings = len(data.rating)
num_users = len(data.userId.unique())
num_items = len(data.movieId.unique())
sparsity = num_ratings/(num_users*num_items)
mean_rating = data.rating.mean()

print(f'\nNumber of users: {num_users}')
print(f'Number of items: {num_items}')
print(f'Number of ratings: {num_ratings}')
print(f'Dataset sparsity: {sparsity}')
print(f'\nAverage rating: {mean_rating}\n')

#splitting variables input
method, value = input_select()

print(f'\nSplitting:\nMethod: {method}\nValue: {value}')

#dataset splitting
print('\nSplitting data into training and testing set\n')
train_set, test_set = split(data, method, value)

print(f'\nTraining set: {train_set}\n')
print(f'\nTesting set: {test_set}\n')

print(f'\nMaking the torch datasets\n')
train_dataset = CF_Dataset(train_set)
test_dataset = CF_Dataset(test_set)

#torch.save(train_dataset, 'movielens_training_set.pkl')
#torch.save(test_dataset, 'movielens_test_set.pkl')

#part 3: modeling

model = SVD(num_users, num_items, mean_rating)
#set the model to cuda for gpu training
#model = model.cuda()
print(f'Model initialized: \n{model}')
```

```
#torch.save(model, 'model.pkl')

#the train function automatically sets the model to cuda
#by reinitializing it as a torch.nn.DataParallel module
#when the cuda parameter is set to True.
#cuda = False will initialize a training session based
#solely on cpu computations.
#in the case of a cuda based training, even if the model
#had initially been set to cpu, in order to calculate the
#the untrained weights and get the test loss, cuda must
#also be set to True

#part 4: training

start = time.time()
train_model(model, train_dataset, wd=0.00028, epochs=100,
cuda=True)
end = time.time()
print(f'\nTime elapsed for training: {end-start} sec')

#average model weights for untrained users/items
set_model_untrained_weights(model, train_dataset, test_dataset,
cuda=True)

#part 5: testing

test_loss(model, test_dataset, cuda=True)
```

Αλγόριθμος παραγοντοποίησης σε PyTorch, τοπική παράλληλη κατανεμημένη εκπαίδευση

```
#pytorch_SVD_distributed.py

#part 1: importing

#coded methods
from split_selection import input_select, input_tasks
from data_encode import encode, split
from models import SVD
from datasets import CF_Dataset
from training import train_model, test_loss,
set_model_untrained_weights

#libraries
```

```
import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
import time #time
import os #operating system
#import sys #system
import copy
#import pickle #pickle
import torch #PyTorch
#import torch.nn as nn #Neural network module
#import torch.nn.functional as F #functions
import torch.multiprocessing as mp #multiprocessing
import torch.distributed as dist #distributed

#sets the number of OpenMP threads to be used by PyTorch to the
maximum.
#the default is half of that amount
#torch.set_num_threads(os.cpu_count())
#torch.set_num_threads(1) #or just one thread...

#reproducibility
torch.manual_seed(0)
np.random.seed(0)

#process initialization method
def init_processes(rank, size, fn, *args, backend='gloo',
**kargs):
    #os.environ['MASTER_ADDR'] = '192.168.1.5'
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    os.environ['WORLD_SIZE'] = str(size)
    os.environ['RANK'] = str(rank)
    #os.environ['GLOO_SOCKET_IFNAME'] = 'enp3s0'
    #dist.init_process_group(backend,
init_method="tcp://192.168.1.5:29500", world_size=size, rank=rank)
    dist.init_process_group(backend, world_size=size, rank=rank)
    fn(*args, rank=rank, world_size=size, **kargs)


#protection of resources and unrestrained spawning control
if __name__ == "__main__":
```

```
#part 2: preparing data

#import data
print(f'\nLoading the data...')
PATH = Path("../Datasets/ml-latest-small")
data = pd.read_csv(PATH/"ratings.csv")

#encode the dataset into unique and contiguous values
data = encode(data)
print(f'Dataset has been encoded')

num_ratings = len(data.rating)
num_users = len(data.userId.unique())
num_items = len(data.movieId.unique())
sparsity = num_ratings/(num_users*num_items)
mean_rating = data.rating.mean()

print(f'\nNumber of users: {num_users}')
print(f'Number of items: {num_items}')
print(f'Number of ratings: {num_ratings}')
print(f'Dataset sparsity: {sparsity}')
print(f'\nAverage rating: {mean_rating}\n')

#splitting variables input
method, value = input_select()

print(f'\nSplitting:\nMethod: {method}\nValue: {value}')

#dataset splitting
print('\nSplitting data into training and testing set\n')
train_set, test_set = split(data, method, value)

print(f'\nTraining set: {train_set}\n')
print(f'\nTesting set: {test_set}\n')

print(f'\nMaking the torch datasets\n')
train_dataset = CF_Dataset(train_set)
test_dataset = CF_Dataset(test_set)

#torch.save(train_dataset, 'movielens_training_set.pkl')
#torch.save(test_dataset, 'movielens_test_set.pkl')
```

```
    #part 3: modeling

    #initialize the model in cpu()
    #NOTE: be careful not to overload the gpu if it is not
necessary
    model = SVD(num_users, num_items, mean_rating)
    #set the model to cuda for gpu training
    model = model.cuda()
    print(f'Model initialized: {model}')

    #part 4: training

    num_tasks = input_tasks()

    #obligatory training arguments (train_args):
    # #1 model, #2 training dataset
    #train_args = [model, train_dataset]

    #keyworded training arguments (kw_train_args).
    #options include:
    #epochs(10)(int): number of epochs, lr(0.01)(float): learning
rate,
    #wd(0.0)(float): weight decay
    #unsqueeze(false)(boolean): add one more dimension to the
ratings for the training,
    #cuda(false)(boolean): train model in cuda,
    #distributed_mode(false)(boolean): distributed training,
    #rank(None)(int): distributed training mode only - indicates
the rank of the running process,
    #world_size(None)(int): distributed training mode only -
indicates the number of tasks/processes
    #participating in the training process
    kw_train_args = {'distributed_mode':True, 'cuda':True}

    #parallel section
    start = time.time() #start counter
    mp.set_start_method('spawn')
    #use this when start method is fork (default) to
    #share a cpu model with forked processes
    #model.share_memory()
    processes = []
    print('Training is distributed. One model copy per process')
```

```
    for rank in range(num_tasks):
        model = copy.deepcopy(model) #new model reference for each
process
        train_args = [model, train_dataset] #reinstate the train
arguments
        #arguments
        arguments = [rank, num_tasks, train_model]
        arguments.extend(train_args)
        #key-worded arguments
        kw_arguments = {'wd':0.00028, 'epochs':100}
        kw_arguments.update(kw_train_args)
        #init
        p = mp.Process(target=init_processes, args=arguments,
kwargs=kw_arguments)
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    end = time.time() #end counter

    print(f'\nTime elapsed for training: {end-start} sec')

    #average model weights for untrained users/items
    set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=True)

    #part 5: testing

    test_loss(model, test_dataset, cuda=True)
```

Αλγόριθμος παραγοντοποίησης σε PyTorch, παράλληλη κατανεμημένη εκπαίδευση, κόμβος μιας διεργασίας

```
#Node_SingleProc.py

#part 1: importing

from training import train_model, test_loss,
set_model_untrained_weights

import numpy as np #numpy
```

```
import time #time
import os #operating system
import sys #system
import torch #PyTorch
import torch.distributed as dist #distributed

#torch.set_num_threads(os.cpu_count())

#reproducibility
torch.manual_seed(0)
np.random.seed(0)

#part 2: preparing data

#load the datasets (torch.utils.data.Dataset)
train_dataset = torch.load('movielens_training_set.pkl')
test_dataset = torch.load('movielens_test_set.pkl')

print(f'\nTraining set: {train_dataset.data}')
print(f'\nTesting set: {test_dataset.data}')

#part 3: modeling

#load the model
model = torch.load("model.pkl").cuda()
print(f'\nModel: {model}')

#part 4: training

backend = 'gloo'
node = int(sys.argv[1])
world_size = int(sys.argv[2])

#os.environ['MASTER_ADDR'] = '192.168.1.4'
os.environ['MASTER_ADDR'] = '195.251.122.92'
#os.environ['MASTER_ADDR'] = '127.0.0.1'
os.environ['MASTER_PORT'] = '29500'
os.environ['WORLD_SIZE'] = str(world_size)
os.environ['RANK'] = str(node)
os.environ['GLOO_SOCKET_IFNAME'] = 'enp3s0'
```

```
dist.init_process_group(backend,
init_method="tcp://195.251.122.92:29500", world_size=world_size,
rank=node)
#dist.init_process_group(backend, world_size=world_size,
rank=node)

start = time.time()
train_model(model, train_dataset, cuda=True,
distributed_mode=True, rank=node, world_size=world_size)
end = time.time()

print(f'\nTime elapsed for training: {end-start} sec')

#average model weights for untrained users/items
set_model_untrained_weights(model, train_dataset, test_dataset,
cuda=True)

#part 5: testing

test_loss(model, test_dataset, cuda=True)
```

Αλγόριθμος παραγοντοποίησης σε PyTorch, παράλληλη κατανεμημένη εκπαίδευση, κόμβος πολλών διεργασιών

```
#Node_MultiProc.py

#part 1: importing

#coded methods
from training import train_model, test_loss,
set_model_untrained_weights

#libraries
import numpy as np #numpy
import time #time
import os #operating system
import sys #system
import copy #copy
import torch #PyTorch
import torch.multiprocessing as mp #multiprocessing
import torch.distributed as dist #distributed

#reproducibility
```

```
torch.manual_seed(0)
np.random.seed(0)

#process initialization method
def init_processes(rank, size, fn, *args, backend='gloo',
**kargs):
    #os.environ['MASTER_ADDR'] = '192.168.1.5'
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    os.environ['WORLD_SIZE'] = str(size)
    os.environ['RANK'] = str(rank)
    #os.environ['GLOO_SOCKET_IFNAME'] = 'enp3s0'
    #dist.init_process_group(backend,
init_method="tcp://192.168.1.5:29500", world_size=size, rank=rank)
    dist.init_process_group(backend, world_size=size, rank=rank)
    fn(*args, rank=rank, world_size=size, **kargs)


#protection of resources and unrestrained spawning control
if __name__ == "__main__":

    #part 2: preparing data

    #load the datasets (torch.utils.data.Dataset)
    train_dataset = torch.load('movielens_training_set.pkl')
    test_dataset = torch.load('movielens_test_set.pkl')

    print(f'\nTraining set: \n{train_dataset.data}\n\n')
    print(f'\nTesting set: \n{test_dataset.data}\n\n')

    #part 3: modeling

    #load the model
    model = torch.load('model.pkl').cuda()
    print(model)

    #part 4: training

    first_task, last_task = eval(sys.argv[1])
    world_size = int(sys.argv[2])

    #train_args = [model, train_dataset]
```

```
    kw_train_args = {'distributed_mode':True, 'cuda':True}

    #parallel section
    start = time.time() #start counter
    mp.set_start_method('spawn')
    processes = []
    print('\nTraining is distributed. One model copy per
process\n\n')
    for rank in range(first_task, last_task+1):
        model = copy.deepcopy(model) #new model reference for each
process
        train_args = [model, train_dataset] #reinstate the train
arguments
        #arguments
        arguments = [rank, world_size, train_model]
        arguments.extend(train_args)
        #key-worded arguments
        kw_arguments = {}
        kw_arguments.update(kw_train_args)
        #init
        p = mp.Process(target=init_processes, args=arguments,
kwargs=kw_arguments)
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    end = time.time() #end counter

    print(f'\nTime elapsed for training: {end-start} sec')

    #average model weights for untrained users/items
    set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=True)

    #part 5: testing

    test_loss(model, test_dataset, cuda=True)
```

Αλγόριθμος παραγοντοποίησης σε PyTorch, απλή εκπαίδευση, εξαγωγή αποτελεσμάτων

```
#pytorch_SVD(test).py
```

```
#part 1: importing

#coded methods
from split_selection import input_select
from data_encode import encode, split
from models import SVD
from datasets import CF_Dataset
from training import train_model, test_loss,
set_model_untrained_weights

#libraries
import numpy as np #numpy
import pandas as pd #pandas
from pathlib import Path #path
import time #time
#import os #operating system
#import sys #system
import torch #PyTorch
#import torch.nn as nn #Neural network module
#import torch.nn.functional as F #functions
import matplotlib.pyplot as plt #plotting
import seaborn as sns

#sets the number of OpenMP threads to be used by PyTorch to the
maximum.
#the default is half of that amount
#torch.set_num_threads(os.cpu_count())
#torch.set_num_threads(1)

#reproducibility
torch.manual_seed(0)
np.random.seed(0)

#part 2: preparing data

#import data
print(f'\nLoading the data...')
PATH = Path("../Datasets/ml-latest-small")
data = pd.read_csv(PATH/"ratings.csv")

#encode the dataset into unique and contiguous values
```

```
data = encode(data)
print(f'Dataset has been encoded')

num_ratings = len(data.rating)
num_users = len(data.userId.unique())
num_items = len(data.movieId.unique())
sparsity = num_ratings/(num_users*num_items)
mean_rating = data.rating.mean()

print(f'\nNumber of users: {num_users}')
print(f'Number of items: {num_items}')
print(f'Number of ratings: {num_ratings}')
print(f'Dataset sparsity: {sparsity}')
print(f'\nAverage rating: {mean_rating}\n')

#splitting variables input
method, value = input_select()

print(f'\nSplitting:\nMethod: {method}\nValue: {value}')

#dataset splitting
print('\nSplitting data into training and testing set\n')
train_set, test_set = split(data, method, value)

print(f'\nTraining set: {train_set}\n')
print(f'\nTesting set: {test_set}\n')

print(f'\nMaking the torch datasets\n')
train_dataset = CF_Dataset(train_set)
test_dataset = CF_Dataset(test_set)

#epochs_array = [5, 10, 20, 30, 40, 50, 60, 70]
epochs_array = [100]
#decay_array = [0.0007, 0.0005, 0.0004]

#make a seaborn set
sns.set()

pal = sns.color_palette("Set2", 3)

fig = plt.figure(figsize=(8,8)) #8x8
```

```
#for i, decay in enumerate(decay_array):

test_mse = []

for epochs in epochs_array:

    #part 3: modeling

    model = SVD(num_users, num_items, mean_rating, emb_size=100)
    #set the model to cuda for gpu training
    model = model.cuda()
    print(f'Model initialized: \n{model}')

    #the train function automatically sets the model to cuda
    #by reinitializing it as a torch.nn.DataParallel module
    #when the cuda parameter is set to True.
    #cuda = False will initialize a training session based
    #solely on cpu computations.
    #in the case of a cuda based training, even if the model
    #had initially been set to cpu, in order to calculate the
    #the untrained weights and get the test loss, cuda must
    #also be set to True

    #part 4: training

    start = time.time()
    train_model(model, train_dataset, lr=0.01, wd=0.00028,
epochs=epochs, cuda=True)
    end = time.time()
    print(f'\nTime elapsed for training: {end-start} sec')

    #average model weights for untrained users/items
    set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=True)

    #part 5: testing

    test_mse += [test_loss(model, test_dataset, cuda=True)]

subplot = fig.add_subplot(1,1,1)
#draw the following
#alpha (intensity) = 0.5
```

```
subplot.plot(epochs_array, test_mse, c=pal[0], label='Test loss',
alpha=0.5, linewidth=5)

#draw the legend at the best location, fonts: 20px
plt.legend(loc='best', fontsize=20)
#make the ticks as following
plt.xticks(fontsize=16);
plt.yticks(fontsize=16);
#and the labels
plt.xlabel('Epochs', fontsize=30);
plt.ylabel('MSE', fontsize=30);
plt.show()
```

Μέθοδοι εισαγωγής επιλογών εκπαίδευσης

```
#split_selection.py

import os #operating system

#returns valid splitting method selection and proper values
#for splitting
def input_select():
    print('Choose splitting method:')
    print('1. Testing size')
    print('2. Percentage split')
    print('3. Use default (Percentage split of 0.1)')
    method = input('')
    if method == '1':
        value = input_split()
    elif method == '2':
        value = input_mask()
    elif method == '3':
        method = 2
        value = float(0.1)
    else:
        print('Again...')
        method, value = input_select()
    return int(method), value

#returns splitting value, fixed testing set size case
def input_split():
    try:
        value = int(input('Insert testing set size:'))
```

```
        if value > 0:
            return value
        else:
            print('Not a positive integer! Again...')
            value = input_split()
            return value
    except ValueError:
        print('Not an integer! Again...')
        value = input_split()
        return value

#returns splitting value, testing set percentage size case
def input_mask():
    try:
        value = float(input('Insert splitting mask (test\
 set percentage size):'))
        if 0.0 < value < 1.0:
            return value
        else:
            print('Splitting mask has to be between 0 and\
 1! Again...')
            value = input_mask()
            return value
    except ValueError:
        print('Not a floating number! Again...')
        value = input_mask()
        return value

#returns valid number of tasks for distributed parallel model
#training
def input_tasks():
    print('\n|--------------------------------|')
    print('\nThis program will distributedly parallelize training
of the given model')
    tasks = input('\n\nPlease insert number of tasks for the
distributed \
training to be split or simply type [d] to use the \
default number of logical processors on your machine \
as the number of tasks:\n\n')
    try:
        if tasks == 'd':
            tasks = os.cpu_count()
```

```
        elif int(tasks) > 10:
            print('\nInput number of tasks is potentially
hazardous \
for the system. Please insert a value lower than 10.')
            tasks = input_tasks()
        elif int(tasks) <= 0:
            print('\nNumber of tasks cannot be a negative number!
Again...')
            tasks = input_tasks()
    except ValueError:
        print('\nNot an integer! Again...')
        tasks = input_tasks()
    return int(tasks)
```

Μέθοδοι κωδικοποίησης δεδομένων

```
#data_encode.py

import numpy as np
import sys

#encodes columns of dataframe data into unique values
#and drops all negative records.
#the dataframe data is returned
def encode(data, columns=['userId', 'movieId']):
    for col_name in columns:
        _,col,_ = proc_col(data[col_name])
        data[col_name] = col
        data = data[data[col_name] >= 0]
    return data

#processes a column and returns a dictionary of its indices
#and values, an array of these values and the length of the
#column. the processing of the column involves transforming it
#into unique values before returning any of the aformentioned
#variables
def proc_col(col):
    uniq = col.unique()
    name2idx = {o:i for i,o in enumerate(uniq)}
    return name2idx, np.array([name2idx.get(x,1) for x in col]),\
    len(uniq)

#used in implicit feedback models.
```

```
#return users and items into a combined transposed (human
readable)
#numpy array index
#eg. idx[13] = [16, 78]: index position 13 contains a combination
of
#a user of id 16 and an item of id 78 (user 16 has rated item 78)
def rating_idx(users, items):
    idx = [users, items]
    idx = np.array(idx).transpose()
    return idx

#returns a tuple of indices of extinct users and items of a
training
#set when compared to a testing set
def get_untrained(test_dataset, train_dataset):

    users = np.array([], dtype=int)
    items = np.array([], dtype=int)

    for i in test_dataset.data['userId'].unique():
        if i not in train_dataset.data['userId'].values:
            users = np.append(users, i)

    for i in test_dataset.data['movieId'].unique():
        if i not in train_dataset.data['movieId'].values:
            items = np.append(items, i)

    return users, items

#split dataframe into train and test sets
def split(data, method=2, value=None):
    try:
        #determined by testing set size
        if method == 1:
            if value == None:
                test = data.sample(n=1000)
            else:
                test = data.sample(n=value)
        #determined by split percentage
        elif method == 2:
            if value == None:
                test = data.sample(frac=0.1)
```

```
            else:
                test = data.sample(frac=value)
        train = data.drop(test.index)
        return train, test
    except Exception:
        print('Invalid method, incorrect dataset input, \
invalid testing set size or slicing \
percentage. Program will exit to avoid \
further errors.')
        sys.exit()
```

Μοντέλα αλγορίθμων

```
#models.py

import torch #PyTorch
import torch.nn as nn #Neural network module

#two-layer net
class TwoLayerNet(nn.Module):

    #constructor
    def __init__(self, D_in=1000, H=100, D_out=10):
        super(TwoLayerNet, self).__init__() #call base constructor
        self.linear1 = nn.Linear(D_in, H) #linear layer 1
        self.linear2 = nn.Linear(H, D_out) #linear layer 2

    #forward function (prediction)
    def forward(self, idx):
        hidden_out=self.linear1(idx).clamp(min=0) #hidden layer
output
        output_out = self.linear2(hidden_out) #output layer output
        return output_out

#SVD (singular value decomposition) model with bias and mean
#rating
class SVD(nn.Module):
    def __init__(self, num_users, num_items, mean, emb_size=100):
        super(SVD, self).__init__()
        self.user_emb = nn.Embedding(num_users, emb_size)
        self.user_emb_bias = nn.Embedding(num_users, 1)
        self.item_emb = nn.Embedding(num_items, emb_size)
        self.item_emb_bias = nn.Embedding(num_items, 1)
```

```
        self.user_emb.weight.data.uniform_(0, 0.005)
        self.user_emb_bias.weight.data.uniform_(-0.01, 0.01)
        self.item_emb.weight.data.uniform_(0, 0.005)
        self.item_emb_bias.weight.data.uniform_(-0.01, 0.01)
        self.mean = nn.Parameter(torch.FloatTensor([mean]), False)


    def forward(self, u, v):
        U = self.user_emb(u)
        b_u = self.user_emb_bias(u).squeeze()
        I = self.item_emb(v)
        b_i = self.item_emb_bias(v).squeeze()
        return (I*U).sum(1) + b_u + b_i + self.mean
```

Datasets

```
#datasets.py

import torch
from torch.utils.data import Dataset

#collaborative filtering dataset based on the MovieLens datasets
class CF_Dataset(Dataset):

    #data initiliaziation is best done in the init method.
    #optimize code to reduce training time
    def __init__(self, data):
        self.data = data
        self.length = len(data)
        self.users = torch.LongTensor(data['userId'].values)
        self.items = torch.LongTensor(data['movieId'].values)
        self.ratings = torch.FloatTensor(data['rating'].values)
        self.timestamp =
torch.LongTensor(data['timestamp'].values)

    def __getitem__(self, idx):
        return {'userId':self.users[idx],
'movieId':self.items[idx], 'rating':self.ratings[idx],
'timestamp':self.timestamp[idx]}

    def __len__(self):
        return self.length
```

Μέθοδοι εκπαίδευσης και αξιολόγησης

```
#training.py

port torch
import torch.nn.functional as F
from torch.utils.data import DistributedSampler, DataLoader
from torch.nn.parallel import DistributedDataParallelCPU,
DistributedDataParallel
import threading
import queue

from data_encode import get_untrained
from misc import queue_iter_print

#factorization module training. optimizer is Adam
#by default. options for distributed and manual distributed
training,
#cuda training and high precision mode (average weights after
finish)
#prints training error per every epoch
def train_model(model, train_dataset, epochs=10, lr=0.01,
                wd=0.0, unsqueeze=False, cuda=False,
                distributed_mode=False, rank=None,
world_size=None):

    rank_prt = f"Rank {rank}: "
    sampler = None

    #printing is assigned to a new thread to minimize training
time by removing
    #I/O interrupts
    #a queue will handle the passing of the print string messages
to the new thread
    q = queue.Queue()
    print_thread = threading.Thread(target=queue_iter_print,
args=(q,))
    print_thread.start()

    #distributed mode
    if distributed_mode:
        q.put(f"{rank_prt}Creating distributed sampler")
```

```
        sampler = DistributedSampler(train_dataset,
num_replicas=world_size, rank=rank)

        if cuda:
            model = DistributedDataParallel(model)
        else:
            model = DistributedDataParallelCPU(model)
        q.put(f"{rank_prt}Set the model to distributed data
parallel")

    #non-distributed. set the model to DataParallel to increase
training speed
    elif not distributed_mode and cuda:
        model = torch.nn.DataParallel(model)

    q.put(f"{rank_prt}Making Dataloader")
    dataloader = DataLoader(train_dataset, batch_size=5001,
num_workers=4, sampler=sampler)

    #optimizer created after the model has been set to a definite
location
    optimizer = torch.optim.Adam(model.parameters(),lr=lr,
                                 weight_decay=wd)
    q.put(f'{rank_prt}Created optimizer')

    model.train()
    q.put(f'{rank_prt}Model is in training mode\n')

    for i in range(epochs):

        if distributed_mode:
            sampler.set_epoch(i)

        for batch_idx, batch_sample in enumerate(dataloader):

            train_prt = f"Epoch {i}, batch {batch_idx}: "

            users = batch_sample["userId"].long()
            items = batch_sample["movieId"].long()
            ratings = batch_sample["rating"].float()

            if unsqueeze:
```

```
                ratings = ratings.unsqueeze(1)

            if cuda:
                users = users.cuda()
                items = items.cuda()
                ratings = ratings.cuda()

            y_hat = model(users, items) #prediction
            loss = F.mse_loss(y_hat, ratings) #loss
            optimizer.zero_grad() #zero gradients
            loss.backward() #update gradients
            optimizer.step() #step
            q.put(f'{rank_prt}{train_prt}Training loss:
{loss.item()} ')

    q.put(".end") #signal the print function to return
    print_thread.join() #and close the thread

#calculates loss error between the prediction and the testing
#set. default method of calculation is mean squared error
def test_loss(model, test_dataset, unsqueeze=False,
              cuda=False):
    model.eval()

    #make dataloader to read test dataset. make batch size very
large
    #to retrieve the whole test set
    dataloader = DataLoader(test_dataset, batch_size=1000000)
    test_set = next(iter(dataloader))

    users = test_set['userId']
    items = test_set['movieId']
    ratings = test_set['rating']

    if cuda == True:
        users = users.cuda()
        items = items.cuda()
        ratings = ratings.cuda()

    if unsqueeze:
        ratings = ratings.unsqueeze(1)
    y_hat = model(users, items)
```

```
    loss = F.mse_loss(y_hat, ratings)
    print(f'\nTest loss: {loss.item()}')
    return loss.item()

#sets the model weights for items and users not included
#in the training set (untrained and unchanged weights)
#to the mean item or user weight.
#a classic cold start problem solution
def set_model_untrained_weights(model, train_dataset,
test_dataset, cuda=False):

    untrained_users, untrained_items = get_untrained(test_dataset,
train_dataset)

    user_trained_indices =
torch.LongTensor(train_dataset.data['userId'].unique())
    item_trained_indices =
torch.LongTensor(train_dataset.data['movieId'].unique())

    if cuda == True:
        user_trained_indices = user_trained_indices.cuda()
        item_trained_indices = item_trained_indices.cuda()

    user_trained_weights =
torch.index_select(model.user_emb.weight, dim=0, index =
user_trained_indices)
    item_trained_weights =
torch.index_select(model.item_emb.weight, dim=0, index =
item_trained_indices)

    user_trained_average = user_trained_weights.mean(0)
    item_trained_average = item_trained_weights.mean(0)

    for i in untrained_users:
        model.user_emb.weight[i] = user_trained_average

    for i in untrained_items:
        model.item_emb.weight[i] = item_trained_average


#misc.py
```

```
#iteratively prints elements of a queue. end keyword is ".end"
def queue_iter_print(q):

    while True:
        prt = q.get()
        if prt == ".end": break
        print(prt)
```

ΟΔΗΓΟΣ ΧΡΗΣΗΣ ΛΟΓΙΣΜΙΚΟΥ

Anaconda

Η Anaconda είναι μια ελεύθερη και open source διανομή των γλωσσών Python και R για σκοπούς επιστημονικών υπολογισμών.

Η Anaconda παρέχει ένα διαχειριστή εικονικών περιβαλλόντων, με κάθε περιβάλλον να έχει τις δικές του βιβλιοθήκες. Το περιβάλλον base, το οποίο είναι και το προεπιλεγμένο περιβάλλον στην Anaconda, έρχεται έχοντας προεγκατεστημένες τις περισσότερες από δημοφιλείς βιβλιοθήκες, όπως είναι οι NumPy, Pandas, Scikit-learn, SciPy και Matplotlib.

Τα περιβάλλοντα αυτά είναι προσβάσιμα από τον Anaconda Navigator, αλλά και μέσω τερματικού, πληκτρολογώντας την εντολή `source activate <environment_name>`, για την ενεργοποίηση του περιβάλλοντος.

Η Anaconda παρέχει ακόμη μια πληθώρα εργαλείων ανάπτυξης, από τα οποία εμείς χρησιμοποιήσαμε το Spyder.

Όσον αφορά την εγκατάσταση της PyTorch στο περιβάλλον μας, απαραίτητο είναι να έχουμε εγκαταστήσει την βιβλιοθήκη NumPy, να έχουμε λειτουργικό σύστημα Linux 13.04 ή νεότερο και εάν επιθυμούμε να εκμεταλλευτούμε τις δυνατότητες της κάρτας γραφικών μας να έχουμε μια έκδοση CUDA 8 ή νεότερη. Η εγκατάσταση της PyTorch γίνεται εύκολα με τις εξής εντολές:

- Για εγκατάσταση με το πακέτο Conda:
  - `conda install pytorch torchvision cuda=9.0 -c pytorch` εάν επιθυμούμε την τελευταία έκδοση της PyTorch για CUDA 9.0 με torchvision. Για CUDA 8 ή 10 πληκτρολογούμε αντίστοιχα `cuda=8.0` ή `cuda=10.0`.
  - Εάν δεν επιθυμούμε να έχουμε υποστήριξη επιτάχυνσης με την χρήση κάρτας γραφικών μπορούμε να πληκτρολογήσουμε: `conda install pytorch-cpu torchvision-cpu -c pytorch`
  - Εάν δεν επιθυμούμε ακόμη να έχουμε επιπλέον την βιβλιοθήκη torchvision μπορούμε να την παραλείψουμε
- Για εγκατάσταση με το πακέτο pip:
  - Για Python 3:

- Για CUDA 9:
  `pip3 install torch torchvision`
- Για CUDA 8:
  `pip3 install https://download.pytorch.org/whl/cu80/torch-1.0.1.post2-cp36-cp36m-linux_x86_64.whl`
  `pip3 install torchvision`
- Για CUDA 10:
  `pip3 install https://download.pytorch.org/whl/cu100/torch-1.0.1.post2-cp36-cp36m-linux_x86_64.whl`
  `pip3 install torchvision`
- Για CPU μόνο:
  `pip3 install https://download.pytorch.org/whl/cpu/torch-1.0.1.post2-cp35-cp35m-linux_x86_64.whl`
  `pip3 install torchvision`

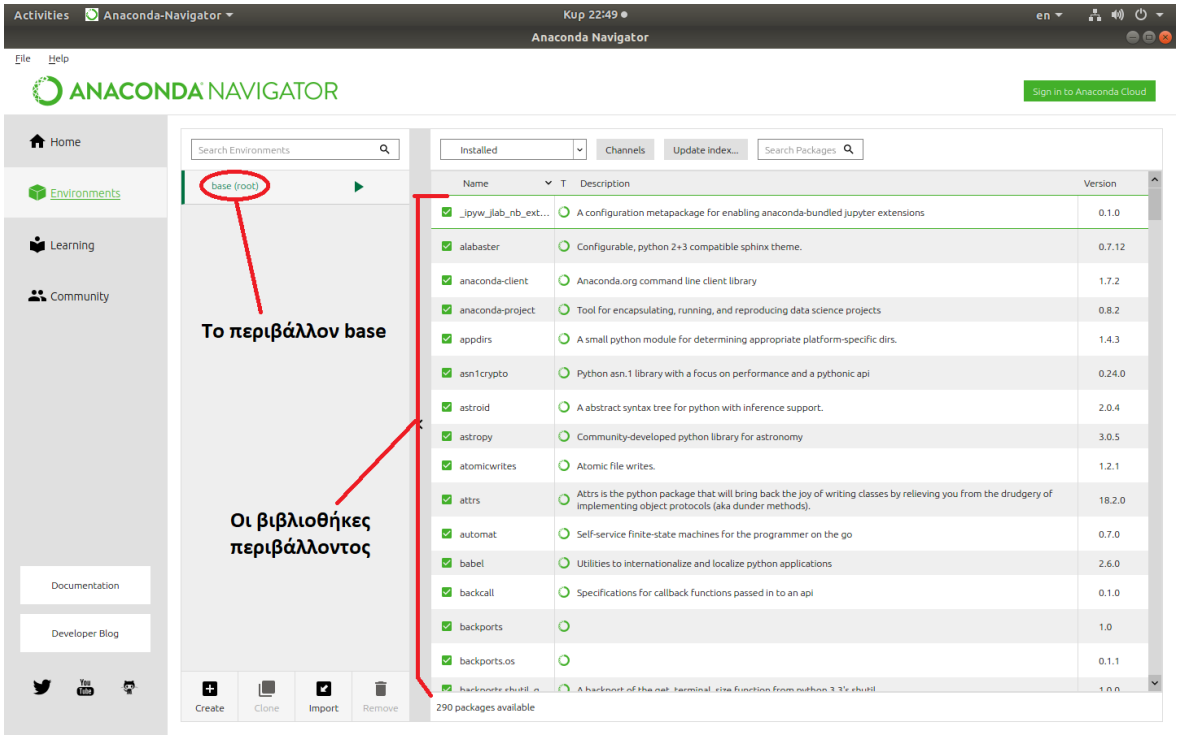
- Για Python 2.7:
  αντικαταστούμε όπου pip3 με pip. Μπορούμε επίσης να παραλείψουμε την εγκατάσταση της torchvision αν το επιθυμούμε

Οι εντολές αυτές πληκτρολογούνται σε τερματικό, στο περιβάλλον που επιθυμούμε να κάνουμε την εγκατάσταση. Εάν δηλαδή, για παράδειγμα, θέλουμε να εγκαταστήσουμε την PyTorch σε ένα περιβάλλον με όνομα environment1, τότε από τερματικό θα χρειαστεί πρώτα να πληκτρολογήσουμε `source activate environment1` για την ενεργοποίηση του περιβάλλοντος και έπειτα τις κατάλληλες εντολές για την εγκατάσταση της αντίστοιχης έκδοσης της PyTorch.

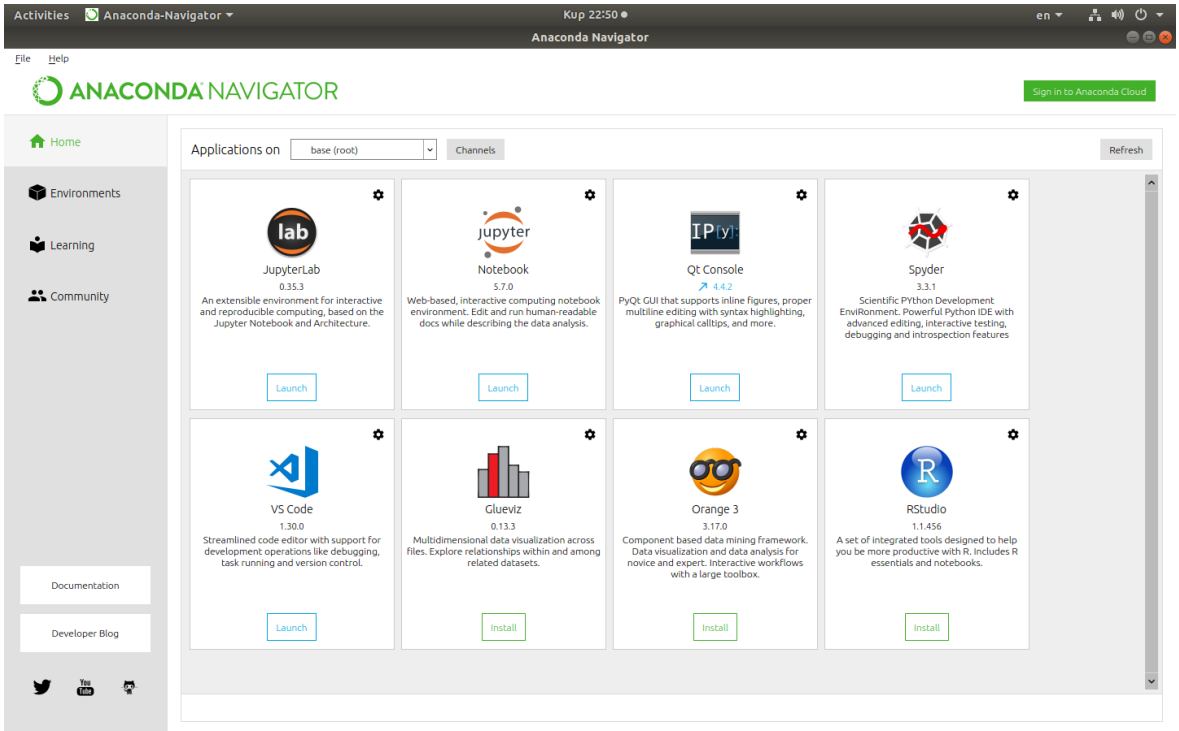
Τα παραπάνω ισχύουν για περιβάλλον Linux. Στην περίπτωση περιβάλλοντος Windows, η εγκατάσταση του Anaconda περιλαμβάνει ένα ακόμη είδος τερματικού, το Anaconda Prompt. Στην περίπτωση Windows λοιπόν, οι ίδιες εντολές θα πρέπει να πληκτρολογηθούν στο τερματικό αυτό.

Για την απενεργοποίηση του περιβάλλοντος θα πρέπει να πληκτρολογήσουμε την εντολή source deactivate.

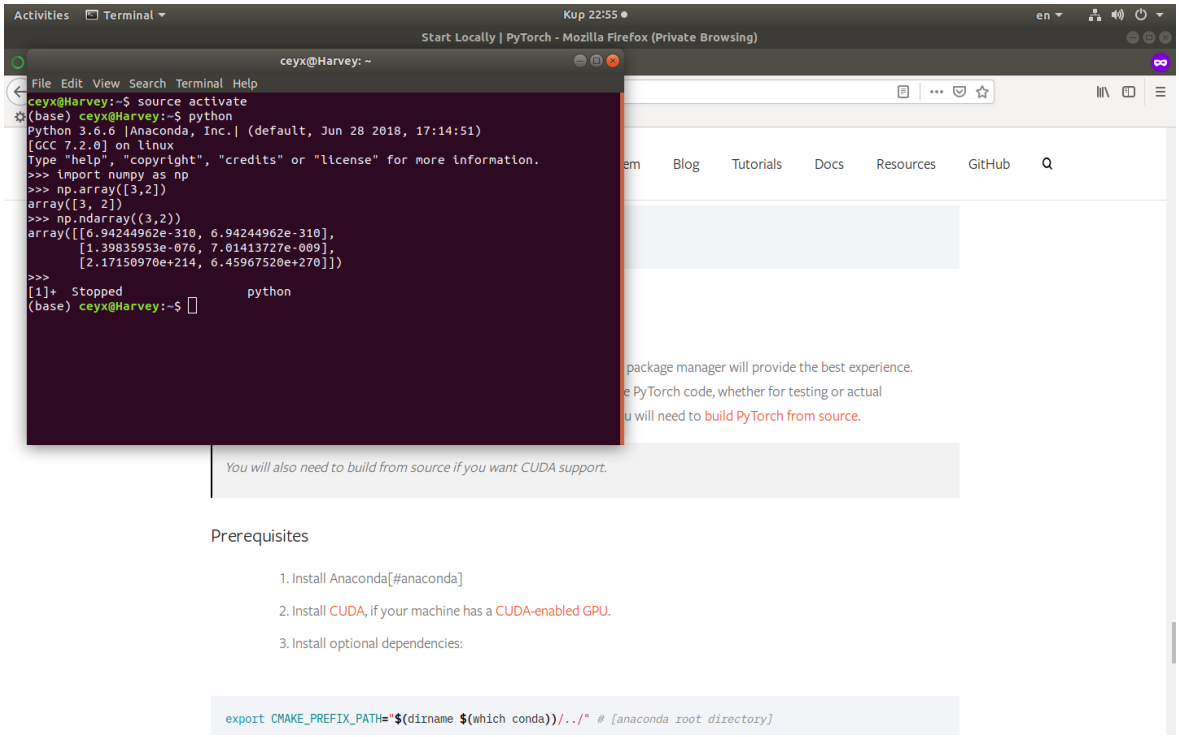
Να σημειωθεί ακόμη πως έχοντας εγκατεστημένη κάποια έκδοση MPI, μπορούμε να χρησιμοποιήσουμε MPI με την PyTorch.



Εικόνα 14 – Anaconda Navigator, περιβάλλον base και βιβλιοθήκες περιβάλλοντος



Εικόνα 15 – Anaconda Navigator, εργαλεία ανάπτυξης



Εικόνα 16 – Περιβάλλον Anaconda, πρόσβαση από τερματικό

Spyder

Το Spyder είναι ένα Python IDE, ένα από τα εργαλεία που προσφέρονται από την διανομή Anaconda.

Το Spyder χωρίζεται σε 4 κύρια βασικά tabs:

- Μενού:

  Βρίσκεται πάνω και περιέχει τις περισσότερες από τις λειτουργίες του IDE. Δίνονται επίσης εργαλεία debugging όπως Step into, Run current, Run selection κτλ.
- Editor:

  Βρίσκεται αριστερά και είναι το tab στο οποίο γράφεται ο κώδικας
- Variable explorer / βοήθεια:

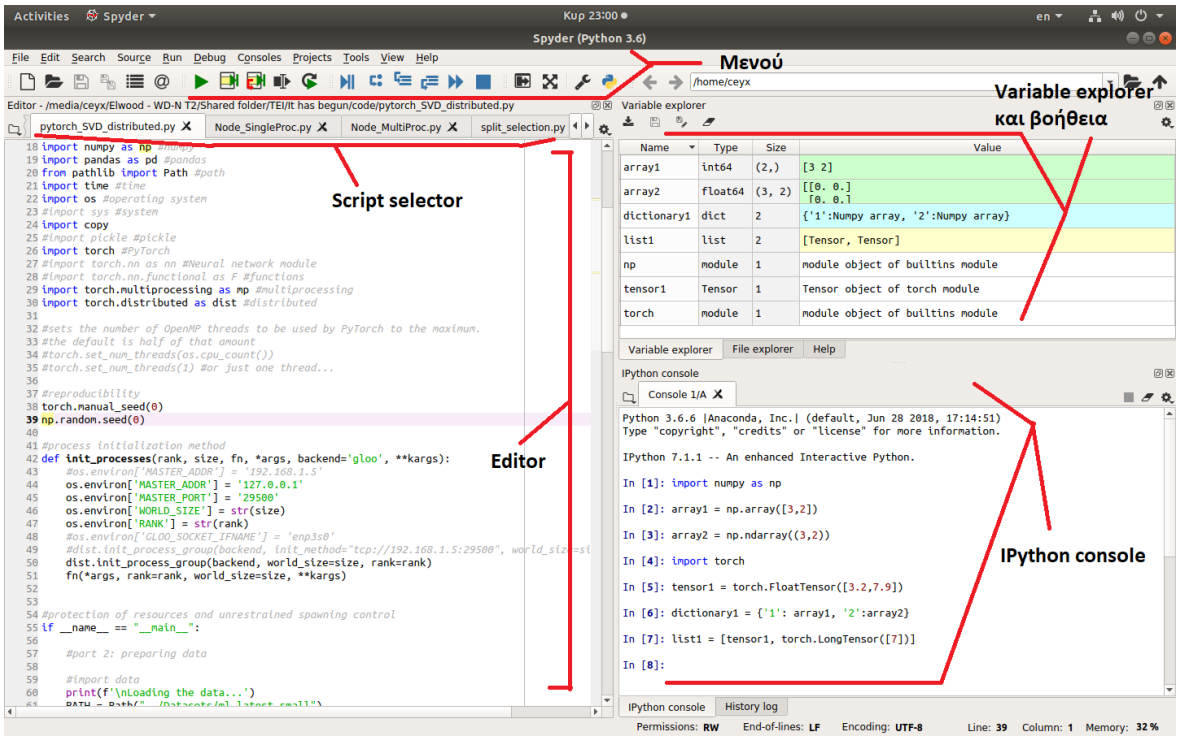
  Βρίσκεται πάνω δεξιά. Το Spyder παρέχει έναν εξερευνητή μεταβλητών όπου μπορεί κανείς να δει όλες τις μεταβλητές που έχει δημιουργήσει και να τις μελετήσει. Ακόμη, στο παράθυρο του Editor, μπορεί κανείς να καλέσει την βοήθεια για οποιαδήποτε μέθοδο ή αντικείμενο πατώντας Ctrl+I, έχοντας ταυτόχρονα τον κέρσορα πάνω σε αυτό. Η βοήθεια μπορεί

να παρέχει συχνά χρήσιμες πληροφορίες για την σύνταξη, γλυτώνοντας χρόνο από αναζήτηση σε αντίστοιχο documentation

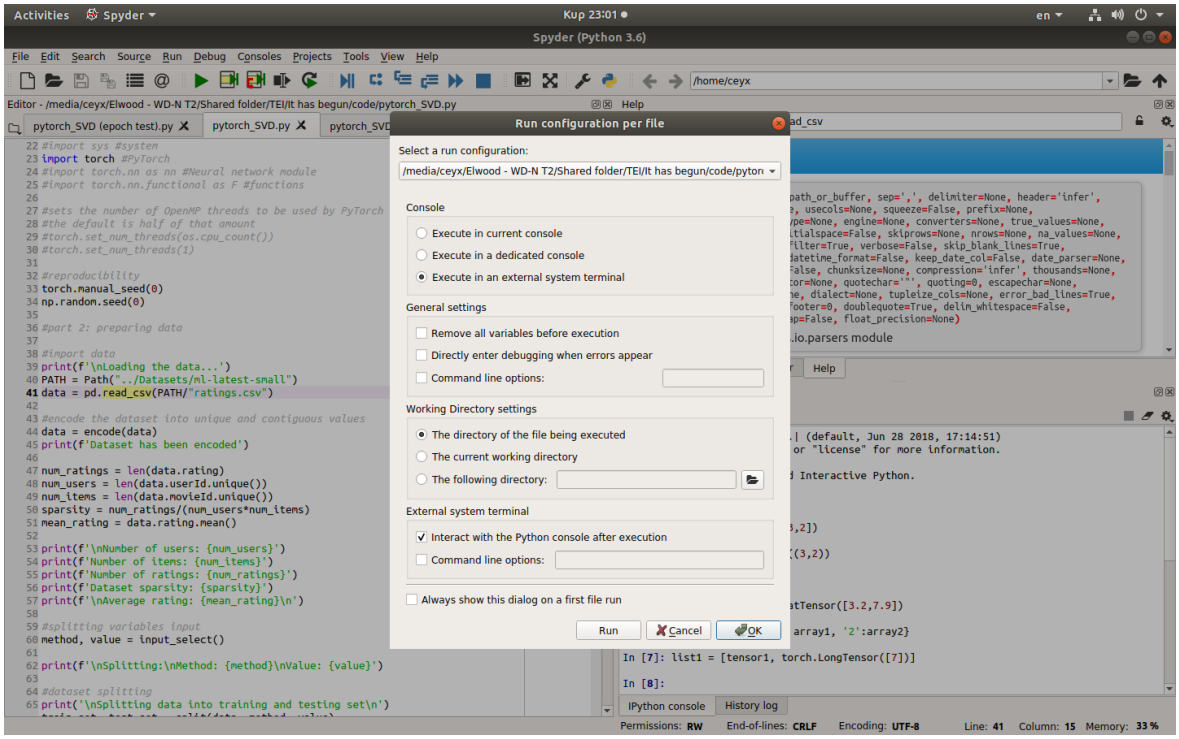
- IPython console:

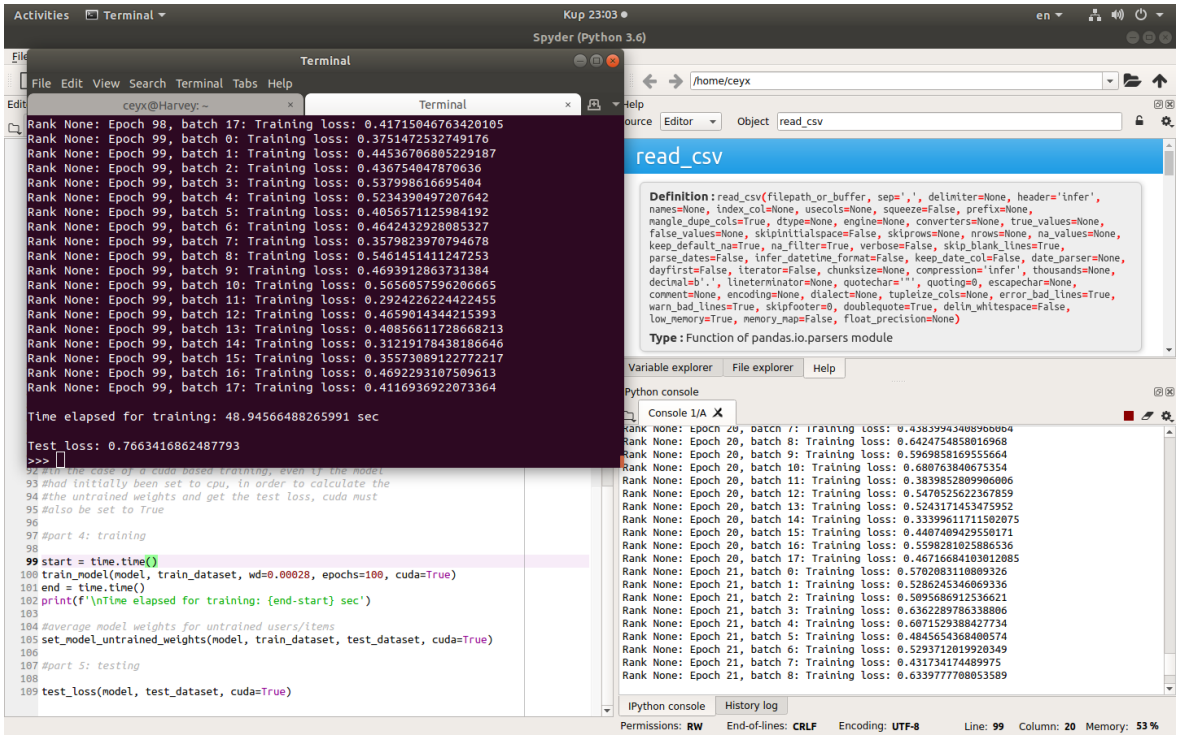
  Το tab κονσόλας IPython βρίσκεται κάτω δεξιά και είναι μια κονσόλα όπου μπορεί κανείς να καλέσει εντολές άμεσα. Οι κονσόλες μπορεί να είναι περισσότερες από μία, με την κάθε μια να έχει τις δικές της μεταβλητές. Οι μεταβλητές αυτές μπορούν να βρεθούν στο tab του Variable explorer, ακριβώς από πάνω. Κάθε πρόγραμμα τρέχει σε μία μόνο από τις κονσόλες.

Να σημειωθεί πως πολλές φορές είναι χρήσιμο να χρησιμοποιείται μια εξωτερική κονσόλα τερματικού, αφού υπάρχουν errors που μπορούν να διαφύγουν από μια κονσόλα IPython. Για να τρέξει κανείς ένα πρόγραμμα σε μια εξωτερική κονσόλα θα πρέπει να το επιλέξει από τις ρυθμίσεις του μενού Run -> Configuration per file.



Εικόνα 17 – Spyder, περιβάλλον ανάπτυξης





Εικόνες 18 – Spyder, εκτέλεση σε εξωτερική κονσόλα

Εγκατάσταση PyTorch from source και dependencies

Εάν το επιθυμεί, μπορεί κανείς να εγκαταστήσει την νεότερη δυνατή έκδοση της PyTorch κάνοντας μια εγκατάσταση from source.

Για την εγκατάσταση θα χρειασθούν τα εξής:

- nVidia drivers
- CUDA 8 ή νεότερη
- cudNN
- NCCL 2 ή νεότερη
- MPI, προαιρετικά

Εγκατάσταση της NCCL

- Κατέβασμα από εδώ: https://developer.nvidia.com/nccl/nccl2-download-survey
- Για λειτουργικά συστήματα Ubuntu 14.04 και 16.04:
  - `sudo dpkg -i nccl-repo-<nccl_version>.deb`
  - `sudo apt update`
  - Για εγκατάσταση και ταυτόχρονα αναβάθμιση στην τελευταία έκδοση CUDA:
    `sudo apt install libnccl2 libnccl-dev`
  - Για εγκατάσταση και επιλογή συγκεκριμένης έκδοσης CUDA:
    `sudo apt install libnccl2=2.0.0-1+cuda8.0 libnccl-dev=2.0.0-1+cuda8.0`
    (παράδειγμα για CUDA 9 και NCCL 2.0)
- Για λοιπά συστήματα Unix:
  - `cd /usr/local`
  - `tar xvf nccl-<version>.txz`
  - Εισαγωγή μεταβλητής περιβάλλοντος NCCL_HOME στο bashrc με την τοποθεσία της NCCL στο σύστημα αρχείων

Για να ελέγξετε πως εγκαταστήσατε την NCCL σωστά μπορείτε να χρησιμοποιήσετε NCCL tests τα οποία θα βρείτε εδώ, μαζί με πληροφορίες στο πως να τα χρησιμοποιήσετε:

https://github.com/NVIDIA/nccl-tests

Εάν επιθυμείτε να μάθετε περισσότερες πληροφορίες σχετικά με την εγκατάσταση της NCCL μπορείτε να πλοηγηθείτε εδώ:

https://docs.nvidia.com/deeplearning/sdk/nccl-install-guide/index.html

Εγκατάσταση της CUDA

Κατεβάστε και ακολουθείστε τις οδηγίες εδώ: https://developer.nvidia.com/cuda-toolkit-archive

Εγκατάσταση της cudNN

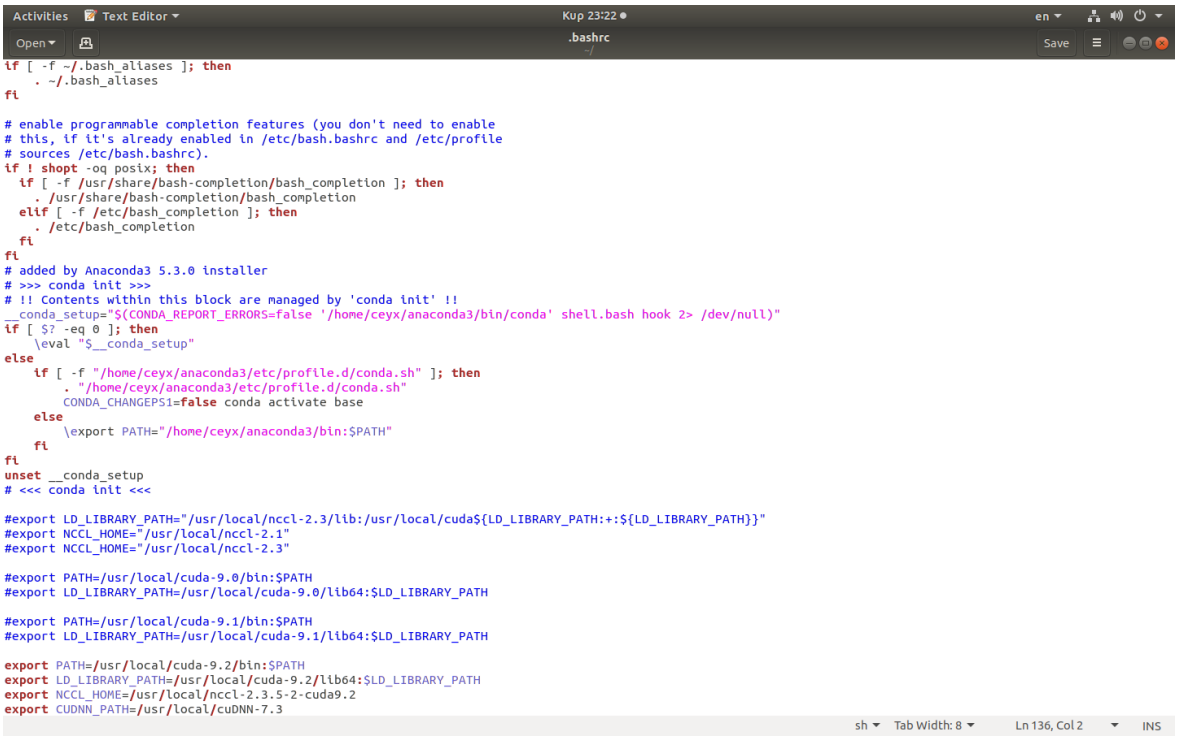
- Κατεβάστε την cudNN από εδώ https://developer.nvidia.com/rdp/cudnn-download
- Για αρχεία tar εκτελέστε τα εξής από τερματικό (παράδειγμα για CUDA 9):
  - Πλοηγηθείτε στον φάκελο του αρχείου
  - `tar -xzvf cudnn-9.0-linux-x64-v7.tgz`
  - `sudo cp cuda/include/cudnn.h /usr/local/cuda/include`
  - `sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64`
  - `sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*`
- Για αρχεία deb (παράδειγμα για CUDA 9):
  - `sudo dpkg -i libcudnn7_7.0.3.11-1+cuda9.0_amd64.deb`
  - `sudo dpkg -i libcudnn7-devel_7.0.3.11-1+cuda9.0_amd64.deb`
  - `sudo dpkg -i libcudnn7-doc_7.0.3.11-1+cuda9.0_amd64.deb`
- Εισαγωγή μεταβλητής CUDNN_PATH στο bashrc με την τοποθεσία της cudNN στο σύστημα αρχείων

Για να σιγουρευτείτε πως η cudNN εγκαταστάθηκε σωστά μπορείτε να κάνετε compile το mnistCUDNN sample. Κάνετε τα εξής:

- Αντιγράψτε το cuDNN sample σε μια προσβάσιμη τοποθεσία
  `cp -r /usr/src/cudnn_samples_v7/ $HOME`
- Πλοηγηθείτε στην τοποθεσία αυτή
  `cd $HOME/cudnn_samples_v7/mnistCUDNN`
- Κάνετε compile το mnistCUDNN sample
  `make clean && make`
- Τρέξτε το
  `./mnistCUDNN`

- Εάν η cuDNN εγκαταστάθηκε σωστά και τρέχει στο σύστημα σας θα δείτε ένα μήνυμα παρόμοιο με αυτό:

  `Test passed!`



Εικόνες 19 – Εγκατάσταση NCCL, CUDA, cuDNN, μεταβλητές περιβάλλοντος

Εγκατάσταση PyTorch from source

Τέλος, για την εγκατάσταση της PyTorch θα χρειαστεί να τρέξετε τα παρακάτω:

- `git clone --recursive` https://github.com/pytorch/pytorch
- `cd pytorch`
- `python setup.py install`