ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Συγκριτική Αξιολόγηση Κατανεμημένων Συστημάτων Βαθιάς Μηχανικής Μάθησης σε Περιβάλλον Υπολογιστικού Νέφους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ανδρέα Κρίσιλια

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

Εθνικο Μετσοβιο Πολυτεχνειο
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Συγκριτική Αξιολόγηση Κατανεμημένων Συστημάτων Βαθιάς Μηχανικής Μάθησης σε Περιβάλλον Υπολογιστικού Νέφους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ανδρέα Κρίσιλια

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 'Ημ/νια παρουσίασης':

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Ιωάννης Κωνσταντίνου
Καθηγητής	Επίκ. Καθηγητής	Επίκ. Καθηγητής
Ε.Μ.Π.	Ε.Μ.Π.	Παν. Θεσσαλίας

Αθήνα, Ιούλιος 2021

.....
Ανδρέας Κρίσιλιας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

Περίληψη

Η δημοτικότητα της βαθιάς μηχανικής μάθησης έχει εκτοξευτεί την τελευταία δεκαετία κυρίως σε εφαρμογές επεξεργασίας ήχου, εικόνας ή φυσικής γλώσσας. Σε συνδυασμό με τον ολοένα αυξανόμενο όγκο των δεδομένων έχει γίνει επιτακτική ανάγκη η εκπαίδευση νευρωνικών δικτύων με κατανεμημένο τρόπο από ομάδες (clusters) υπολογιστικών πόρων. Παράλληλα, έχουν εκδοθεί πολλαπλές βιβλιοθήκες/συστήματα για την διευκόλυνση υλοποίησης και εκπαίδευσης βαθιών νευρωνικών δικτύων. Θα ήταν χρήσιμο, συνεπώς, για τον χρήστη να μπορεί να διακρίνει τις διαφορές μεταξύ των διαφορετικών συστημάτων τόσο σε απόδοση όσο και σε ευχρηστία, ώστε να μπορεί να διαλέξει το κατάλληλο σύμφωνα με τις δικές του ανάγκες.

Στην παρούσα εργασία εξετάζονται τρία από τα πιο διάσημα συστήματα βαθιάς μηχανικής μάθησης που υποστηρίζουν εκπαίδευση σε κατανεμημένο περιβάλλον, τα TensorFlow, PyTorch και MXNet. Αξιολογούνται συγκριτικά πάνω σε cluster τριών μηχανημάτων αποτελούμενα από επεξεργαστές Intel. Τα πειράματα απαρτίζονται από έξι συνδυασμούς μεταξύ τεσσάρων διαφορετικών νευρωνικών δικτύων και δύο συνόλων δεδομένων. Από τα αποτελέσματα φαίνεται ότι τα PyTorch και MXNet είναι πολύ πιο αποδοτικά όταν τρέχουν σε επεξεργαστές Intel λόγω των ειδικά βελτιστοποιημένων τελεστών που χρησιμοποιούν, ενώ το TensorFlow μπορεί να μειώσει το χρόνο φόρτωσης δεδομένων ιδίως σε μικρά δίκτυα κάνοντας χρήση της cache. Ακόμα, γίνεται εντοπισμός αργών υλοποιήσεων μεμονωμένων τελεστών σε όλα τα συστήματα. Το MXNet φαίνεται να υπερέχει ελαφρώς στην ταχύτητα κατανεμημένης επικοινωνίας, παρότι αυτό δεν επηρεάζει ιδιαίτερα το τελικό αποτέλεσμα.

Abstract

The popularity of deep learning has skyrocketed in the last decade mainly in audio, video or natural language processing applications. In combination with the increasing volume of data, it has become imperative to train neural networks in a distributed way using clusters of computational resources. At the same time, multiple libraries/systems have been published to facilitate the implementation and training of deep neural networks. Thus, it would be useful for the user to be able to distinguish the differences between the different systems in both performance and usability, so that they can choose the right one according to their own needs.

This thesis examines three of the most popular deep learning systems that support distributed training, TensorFlow, PyTorch and MXNet. They are comparatively evaluated on a cluster of three machines consisting of Intel processors. The experiments consist of six combinations between four different neural networks and two datasets. The results show that PyTorch and MXNet are much more efficient when running on Intel processors due to the specially optimized operators they use, while TensorFlow can reduce data load time especially on small networks by using cache. Also, slow implementations of individual operators are detected in all systems. MXNet seems to outperform slightly in the speed of distributed communication, although this does not significantly affect the end result.

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω ιδιαιτέρως τον υποψήφιο διδάκτορα Νικόδημο Προβατά, με τον οποίο συνεργάστηκα καθ' όλη τη διάρκεια υλοποίησης της παρούσας εργασίας.

Θα ήθελα, επίσης, να εκφράσω τις ευχαριστίες μου στον επιβλέποντα καθηγητή μου, κύριο Νεκτάριο Κοζύρη, καθώς και στον καθηγητή μου Ιωάννη Κωνσταντίνου που με εμπιστεύτηκαν και με καθοδήγησαν για την εκπόνηση της παρούσας εργασίας. Ακόμα, θα ήθελα να ευχαριστήσω την υποψήφια διδάκτορα Ευδοκία Κασσελά που ήταν πάντα παρούσα και ασχολούνταν άμεσα με όποιο δικτυακό πρόβλημα προέκυπτε, καθώς και όλα τα μέλη του εργαστηρίου Υπολογιστικών Συστημάτων.

Ακόμα, θα ήθελα να ευχαριστήσω την οικογένεια μου και τους φίλους μου για την υποστήριξή τους καθ' όλη τη διάρκεια των σπουδών μου. Τέλος, θα ήθελα να ευχαριστήσω ιδιαιτέρως την κοπέλα μου Δήμητρα που με στήριξε σε όλες τις δύσκολες στιγμές.

Περιεχόμενα

Λίστα Πινάκων

Λίστα Σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Η βαθιά μηχανική μάθηση συναντάται πλέον ολοένα και περισσότερο σε εφαρμογές που διαχειρίζονται μεγάλο όγκο δεδομένων, ειδικά όταν τα δεδομένα αυτά αφορούν ήχο, εικόνα ή φυσική γλώσσα. Με την αύξηση της δημοτικότητας της βαθιάς μηχανικής μάθησης έχουν αυξηθεί και οι διαφορετικές λύσεις που παρέχονται στο χρήστη για την υλοποίησή της. Την τελευταία οκταετία μόνο, έχουν εκδοθεί πάνω από δέκα βιβλιοθήκες/συστήματα βαθιάς μηχανικής μάθησης, μεταξύ των οποίων τα Caffe[1], Chainer[2], Keras[3], MXNet[4], TensorFlow[5] και PyTorch[6].

Καθώς όμως τα δεδομένα αυξάνονται, το ίδιο κάνουν και οι παράμετροι των νευρωνικών δικτύων, ώστε τα μοντέλα μάθησης να έχουν περισσότερα περιθώρια να κατανοήσουν επαρκώς τον αυξανόμενο όγκο δεδομένων. Για τη διαχείριση, λοιπόν, των μεγάλων όγκων δεδομένων αλλά και του επιπρόσθετου υπολογιστικού κόστους που συνεπάγεται η επέκταση των νευρωνικών δικτύων, καθίσταται επιτακτική ανάγκη ο διαμοιρασμός της βαθιάς μηχανικής μάθησης σε παραπάνω από μία διεργασίες και μηχανήματα.

Κατά την κατανεμημένη βαθιά μηχανική μάθηση δύο είναι οι κύριες αποφάσεις που απασχολούν. Η πρώτη αφορά την αρχιτεκτονική των διαθέσιμων υπολογιστικών πόρων και τους ρόλους που επιτελεί ο καθένας. Βασικές επιλογές σε αυτό αποτελούν η αρχιτεκτονική Parameter Server[7] (όπου κάποιοι κόμβοι αναλαμβάνουν την αποθήκευση και ανανέωση των παραμέτρων του μοντέλου και άλλοι επιτελούν το κύριο υπολογιστικό κομμάτι) και η All-Reduce[8] (όπου όλοι οι κόμβοι αναλαμβάνουν τόσο το υπολογιστικό κομμάτι όσο και την επικοινωνία και ανανέωση των παραμέτρων του μοντέλου). Η δεύτερη απόφαση αφορά το αν οι κόμβοι θα βρίσκονται σε συγχρονισμό[9] μεταξύ τους και θα εργάζονται ουσιαστικά σαν ομάδα ή αν θα γίνεται ασύγχρονη[10] εκπαίδευση και ο κάθε κόμβος θα δουλεύει ατομιστικά χωρίς να γνωρίζει σε ποια φάση βρίσκονται οι υπόλοιποι.

Κατά την παρούσα διπλωματική επιλέχθηκαν τρία από τα πιο διάσημα συστήματα βαθιάς μηχανικής μάθησης που υποστηρίζουν κατανεμημένη εκτέλεση. Αυτά είναι τα

TensorFlow, PyTorch και MXNet, τα οποία συγκρίθηκαν τόσο σε θεωρητικό όσο και σε πειραματικό επίπεδο. Σκοπός είναι μέσω της ανάλυσης αυτής να αποκτήσει ο χρήστης μία καλή εικόνα σχετικά με τα σημεία στα οποία υπερτερεί το κάθε σύστημα και να είναι σε θέση να επιλέξει ποιο εργαλείο τον εξυπηρετεί καλύτερα στο συγκεκριμένο πρόβλημα που έχει να επιλύσει.

1.2 Δομή

Η δομή της παρούσας εργασίας συνοψίζεται παρακάτω:

• Στο *Κεφάλαιο 2* γίνεται μία θεωρητική περιγραφή των νευρωνικών - και κατά κύριο λόγο συνελικτικών - δικτύων, της διαδικασίας με την οποία ένα νευρωνικό δίκτυο "μαθαίνει" και του τρόπου με τον οποίο μπορεί να πραγματοποιηθεί η μάθηση αυτή σε κατανεμημένο περιβάλλον.

• Στο *Κεφάλαιο 3* αναλύονται θέματα αρχιτεκτονικής, υλοποίησης και κατανεμημένης εκτέλεσης των τριών συστημάτων που επελέγησαν (TensorFlow, PyTorch, MXNet).

• Στο *Κεφάλαιο 4* παρουσιάζεται η μεθοδολογία που ακολουθήθηκε κατά το πειραματικό στάδιο της παρούσας εργασίας και στην μετέπειτα σύγκριση.

• Στο *Κεφάλαιο 5* παρατίθενται τα αποτελέσματα που προέκυψαν από το πειραματικό στάδιο και εξηγούνται αναλυτικά οι διαφορές που εντοπίστηκαν μεταξύ των συστημάτων.

• Τέλος, στο *Κεφάλαιο 6* γίνεται μία σύνοψη των συμπερασμάτων που εξήχθησαν από την σύγκριση των τριών συστημάτων και προτείνονται τρόποι με τους οποίους η παρούσα εργασία θα μπορούσε να επεκταθεί μελλοντικά.

Κεφάλαιο 2

Νευρωνικά Δίκτυα

Το κεφάλαιο αυτό αποτελεί μια εισαγωγή στα νευρωνικά δίκτυα, με επεξήγηση των βασικών σημείων για τη δομή και τη λειτουργικότητά τους. Συγκεκριμένα, θα περιγραφούν τόσο η δομή όσο και η διαδικασία εκπαίδευσης των νευρωνικών δικτύων και θα αναλυθούν τεχνικές που αξιοποιούνται για την κατανεμημένη εκτέλεσή τους. Τονίζεται ότι η μεγαλύτερη εμβάθυνση θα γίνει γύρω από τα συνελικτικά δίκτυα και τα προβλήματα κατηγοριοποίησης, καθώς αυτά αποτελούν τον κεντρικό άξονα της παρούσας διπλωματικής.

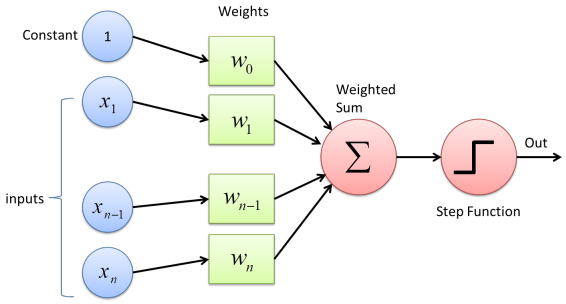
2.1 Εισαγωγή στα τροφοδοτικά νευρωνικά δίκτυα

Τα πολυεπίπεδα τροφοδοτικά νευρωνικά δίκτυα είναι ένα αλγοριθμικό μοντέλο που προσομοιώνει τη λειτουργία των νευρώνων σε ένα ζωντανό οργανισμό. Όπως ακριβώς οι συνάψεις μεταφέρουν σήματα μεταξύ νευρώνων στο νευρικό σύστημα ενός έμβιου όντος, έτσι και οι ακμές μεταξύ των τεχνητών νευρώνων του μαθηματικού αυτού μοντέλου μεταφέρουν πληροφορία μέχρι να βγει το επιθυμητό αποτέλεσμα στην έξοδο. Τα δίκτυα αυτά, λοιπόν, ακολουθούν μια ιεραρχική αρχιτεκτονική[11], υπό την έννοια ότι η εκπαίδευσή τους ξεκινά από τη σύνθεση χαρακτηριστικών στα χαμηλότερα επίπεδα και κατόπιν προχωρά στα υψηλότερα επίπεδα, αξιοποιώντας κάποιες κρυφές μεταβλητές. Πρόκειται, μάλιστα, για πολύ δημοφιλή μοντέλα ευρέως διαδεδομένα και χρησιμοποιούμενα στην επιστημονική κοινότητα αλλά και σε άλλους τομείς, για την επίλυση ποικίλων προβλημάτων μηχανικής μάθησης.

Η ιεραρχική δομή των νευρωνικών δικτύων γίνεται εμφανής στην εικόνα 2.2. Η πληροφορία μεταδίδεται αλυσιδωτά μεταξύ των επιπέδων, υπό την έννοια ότι η έξοδος του ενός αποτελεί την είσοδο του επόμενου. Τα επιμέρους επίπεδα που σχηματίζουν το νευρωνικό δίκτυο μπορούν να χωριστούν σε τρεις βασικές κατηγορίες, το επίπεδου εισόδου, τα κρυφά (ενδιάμεσα) επίπεδα και το επίπεδο εξόδου. Στο επίπεδο εισόδου εισάγονται τα ακατέργαστα δεδομένα,

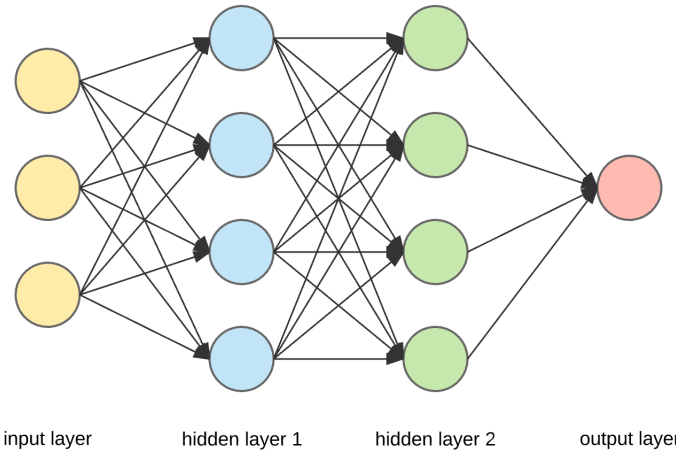
αντιστοιχίζοντας νευρώνες με χαρακτηριστικά, ενώ στα κρυφά επίπεδα αυτά τα δεδομένα υφίστανται επεξεργασία μέχρι να φτάσουν στο επίπεδο εξόδου, όπου έχουν πλέον πάρει την επιθυμητή για το πρόβλημα μορφή. Οι νευρώνες που εμφανίζονται στα διάφορα επίπεδα του δικτύου μπορούν να θεωρηθούν ως Αισθητήρες (Perceptrons)[12]. Το Perceptron σαν έννοια στηρίζεται στο μαθηματικό μοντέλο των McCulloch και Pitts, το οποίο περιγράφεται στην εικόνα 2.1. Μέσω της εικόνας, γίνεται εμφανές ότι η τιμή εξόδου ενός νευρώνα προκύπτει από το άθροισμα των εισόδων x_i του νευρώνα, πολλαπλασιασμένων με κάποια βάρη w_i, σύμφωνα με την εξίσωση:

$$h = \sum_{i=1}^{k} w_i x_i \tag{2.1}$$



Εικόνα 2.1: Δομή Perceptron

Το άθροισμα αυτό κατόπιν προωθείται σε μια συνάρτηση, η οποία ονομάζεται συνάρτηση ενεργοποίησης, όπου θα καθοριστεί κατά πόσον ο νευρώνας θα λάβει μέρος στην παραγωγή του τελικού αποτελέσματος. Ένα παράδειγμα τέτοιας συνάρτησης είναι η και η απλή βηματική συνάρτηση κατωφλίου, όπου ένα κατώφλι ϑ καθορίζει την ενεργοποίηση του νευρώνα. Οι συναρτήσεις ενεργοποίησης που χρησιμοποιούνται συχνότερα στο πεδίο της μηχανικής μάθησης για το επίπεδο εισόδου και τα κρυφά επίπεδα είναι η σιγμοειδής ή λογιστική (sigmoid or logistic)[13], η υπερβολική εφαπτομένη (tanh)[14], καθώς και η μονάδα γραμμικού ανορθωτή (rectified linear unit – ReLU)[15]. Αντίθετα, για το επίπεδο εξόδου η συνάρτηση ενεργοποίησης που χρησιμοποιείται συχνότερα στα προβλήματα κατηγοριοποίησης είναι η Softmax[16].



Εικόνα 2.2: Παράδειγμα πολυεπίπεδου νευρωνικού δικτύου

2.2 Συνελικτικά Τροφοδοτικά Νευρωνικά Δίκτυα

2.2.1 Εισαγωγή

Τα συνελικτικά νευρωνικά δίκτυα (Convolutional Neural Network - CNN)[17] είναι μία κατηγορία νευρωνικών δικτύων βαθιάς μάθησης που έχουν σχεδιαστεί για επεξεργασία δομημένων πινάκων δεδομένων και είναι ιδιαιτέρως διάσημα σε εφαρμογές ανάλυσης εικόνας, όπως αναγνώριση και ταξινόμηση εικόνας και βίντεο, σε συστήματα προτάσεων, στην επεξεργασία φυσικής γλώσσας και σε άλλα πεδία.

Η προ-επεξεργασία που απαιτείται στα CNN είναι πολύ μικρότερη σε σχέση με άλλους αλγορίθμους ταξινόμησης. Ενώ το στάδιο αυτό γίνεται συνήθως χειροκίνητα, στην περίπτωση των CNN υπάρχουν ενσωματωμένα φίλτρα με παραμέτρους τις οποίες μπορεί το νευρωνικό να μάθει και να βελτιστοποιήσει, ώστε να εκθέσει χρήσιμα και πολύπλοκα μοτίβα που περιέχουν τα δεδομένα εισόδου. Τα κρυφά επίπεδα ενός τέτοιου δικτύου αντιστοιχούν σε αρκετά περίπλοκες συναρτήσεις και η συνάρτηση ενεργοποίησης που χρησιμοποιείται συνήθως είναι η ReLU.

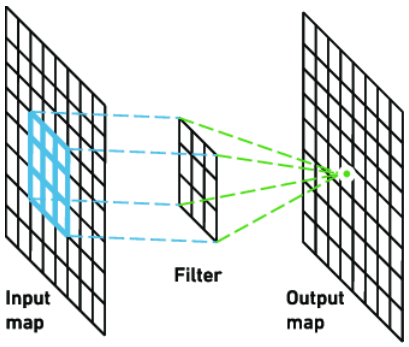
2.2.2 Περιγραφή Επιπέδων

Τα ενδιάμεσα - κρυφά επίπεδα των συνελικτικών νευρωνικών δικτύων περιλαμβάνουν διάφορες κατηγορίες επιπέδων, οι οποίες περιγράφονται παρακάτω.

Συνελικτικό επίπεδο (Convolutional Layer)

Το συνελικτικό επίπεδο εφαρμόζει συνέλιξη στα δεδομένα εισόδου, εξάγοντας έτσι χαρακτηριστικά από αυτά, γι' αυτό και η έξοδος ονομάζεται συχνά χάρτης

χαρακτηριστικών (feature map). Οι κύριες υπερπαράμετροι που χρησιμοποιεί είναι ένας πυρήνας (πολυδιάστατο διάνυσμα που χρησιμοποιείται ως συνελικτικό φίλτρο), τα κανάλια εισόδου (συμπίπτουν με τα κανάλια της εικόνας που εισάγεται ως είσοδος) και τα κανάλια εξόδου (καθορίζουν τα κανάλια που θα έχει η έξοδος). Ακόμα, συχνά χρησιμοποιούμενες είναι οι παράμετροι βήμα ολίσθησης (stride), που ορίζει κατά πόσες μονάδες θα ολισθαίνει το kernel κάθε φορά, και το παραγέμισμα (padding), που ορίζει αν θα προστεθούν έξτρα φτιαχτά δεδομένα στα περιθώρια της εισόδου. Τα συνελικτικά είναι ουσιαστικά τα επίπεδα που εξάγουν τα χαρακτηριστικά της εισόδου, τα οποία στη συνέχεια θα ταξινομηθούν από μία αλληλουχία πλήρως συνδεδεμένων επιπέδων. Αποτελούν επίσης το πιο βαρύ υπολογιστικά επίπεδο ενός συνελικτικού δικτύου στις περισσότερες περιπτώσεις. Σχηματικά μία απλή απεικόνιση του υπολογισμού της συνέλιξης φαίνεται στο σχήμα 2.3.



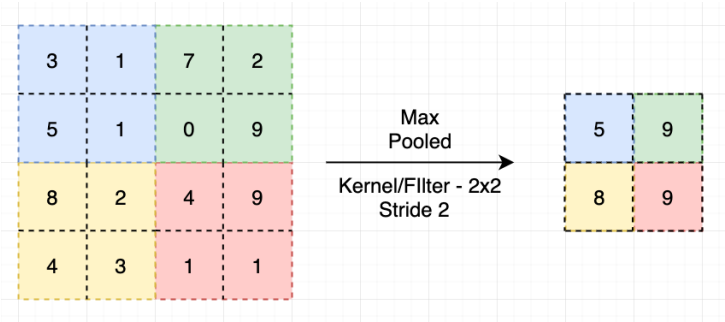
Εικόνα 2.3: Υπολογισμός συνέλιξης με kernel 3x3

Επίπεδο ομαδοποίησης (Pooling Layer)

Το επίπεδο ομαδοποίησης χρησιμοποιείται για να μειώσει τη διαστατικότητα, ώστε να περιορίσει το υπολογιστικό κόστος. Εφαρμόζει ουσιαστικά μία (υπό)δειγματοληψία στην είσοδο που δέχεται, κρατώντας συνήθως είτε το μέγιστο (max pooling) είτε το μέσο όρο (average pooling) της ομάδας που ορίζει ο πυρήνας του. Στην εικόνα 2.4 φαίνεται ένα παράδειγμα για max pooling με kernel μεγέθους 2x2.

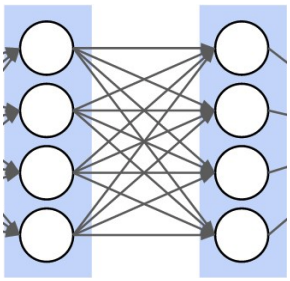
Πλήρως συνδεδεμένο επίπεδο (Fully Connected Layer)

Τα πλήρως συνδεδεμένα επίπεδα αποτελούν ουσιαστικά τον ταξινομητή του συνελικτικού δικτύου και γι' αυτό συναντώνται συνήθως ως τα τελευταία επίπεδα του νευρωνικού. Δομικά αποτελούνται από μία στοίχιση απλών νευρώνων, καθένας από τους οποίους συνδέεται με όλους τους νευρώνες ενός διαδοχικού επιπέδου. Από την έξοδο του τελευταίου πλήρως συνδεδεμένου επιπέδου και με εφαρμογή της συνάρτησης ενεργοποίησης softmax πάνω σε αυτήν προκύπτει ένα μονοδιάστατο διάνυσμα με μήκος ίσο με το πλήθος των κατηγοριών του προβλήματος (ή και 1 για



Εικόνα 2.4: Max Pooling με kernel 2x2

δυαδική ταξινόμηση). Το διάνυσμα αυτό περιέχει για κάθε κατηγορία την ψευδο-πιθανότητα που δίνει το νευρωνικό για την αρχική είσοδο να ανήκει σε αυτήν την κατηγορία. Η σύνδεση μεταξύ δύο πλήρως συνδεδεμένων επιπέδων φαίνεται στην εικόνα 2.5.



Εικόνα 2.5: Σύνδεση μεταξύ δύο πλήρως συνδεδεμένων επιπέδων

Επίπεδο κανονικοποίησης (Batch Normalization layer)

Τα επίπεδα κανονικοποίησης, όπως προδίδει και το όνομά τους, εφαρμόζουν κανονικοποίηση στην έξοδο του προηγούμενου επιπέδου ώστε η είσοδος του επόμενου να είναι κανονικοποιημένη. Η κανονικοποίηση αυτή βοηθάει στο να περιορίσει την κατανομή του σήματος εισόδου σε κάθε επίπεδο, αφού μέσω μικρών αλλαγών σε προηγούμενα επίπεδα είναι δυνατόν η είσοδος σε κάποιο επόμενο επίπεδο να "ξεφύγει"[18]. Ως αποτέλεσμα το νευρωνικό παράγει καλύτερα διανύσματα κλίσης για την ανανέωση των βαρών.

Ο μετασχηματισμός που εκτελείται φαίνεται στην εικόνα 2.6. Αρχικά η είσοδος x_i κανονικοποιείται με τη μέση τιμή και την τυπική απόκλιση του ίδιου του batch. Ύστερα κλιμακώνεται και μετατοπίζεται με τις παραμέτρους γ και β αντίστοιχα. Αυτές οι δύο παράμετροι είναι εκπαιδεύσιμες και άρα βελτιστοποιούνται σε κάθε επανάληψη του βρόχου μάθησης.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Εικόνα 2.6: Μετασχηματισμός Batch Normalization

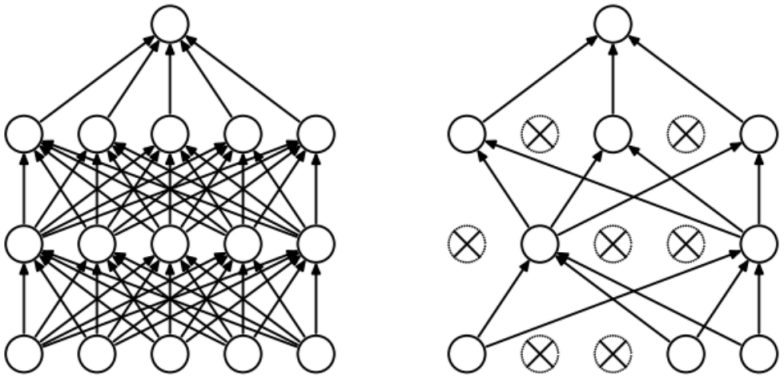
Επίπεδο εγκατάλειψης (Dropout layer)

Τα επίπεδα εγκατάλειψης προσθέτουν μία τυχαιότητα στη μάθηση του νευρωνικού. Ουσιαστικά απενεργοποιούν ορισμένους νευρώνες του προηγούμενου επιπέδου με κάποια προκαθορισμένη πιθανότητα. Έτσι, σε κάθε βήμα του νευρωνικού διαφορετικοί νευρώνες λαμβάνουν μέρος και ανανεώνουν τα βάρη τους. Τα επίπεδα εγκατάλειψης αποτελούν μία τεχνική για την αντιμετώπιση του φαινομένου της υπερ-προσαρμογής (overfit). Κατά το overfit, το νευρωνικό μαθαίνει υπερβολικά καλά να ταξινομεί τα δείγματα του συνόλου προπόνησης. Προσαρμόζει δηλαδή τα βάρη του πολύ αυστηρά με βάση το συγκεκριμένο σύνολο δεδομένων. Το πρόβλημα εμφανίζεται όταν χρειαστεί να ταξινομήσει ένα δείγμα που βρίσκεται έξω από αυτό το σύνολο, να γενικεύσει δηλαδή τις προβλέψεις του. Χάρη στα επίπεδα εγκατάλειψης λοιπόν, η είσοδος σε συγκεκριμένα επίπεδα του νευρωνικού θα έχει πάντα κάποιες μικρές αλλαγές και θα αποτρέψει το νευρωνικό από το να προσαρμόσει τα βάρη του σε συγκεκριμένα δείγματα. Ένα παράδειγμα νευρωνικού δικτύου με και χωρίς επίπεδο εγκατάλειψης φαίνεται στην εικόνα 2.7.

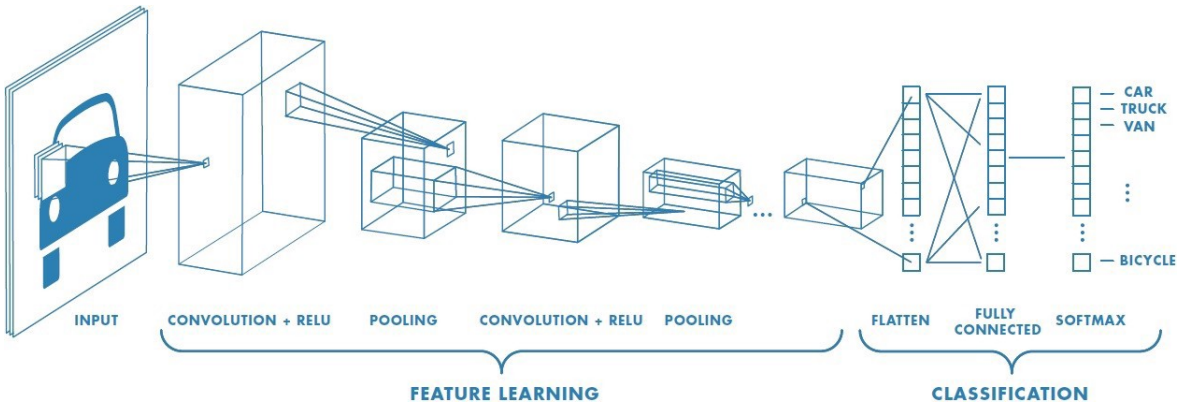
Μία απεικόνιση ενός πλήρους συνελικτικού νερωνικού δικτύου φαίνεται στην εικόνα 2.8.

2.2.3 Συνελικτικά Δίκτυα της διπλωματικής

Στην παρούσα διπλωματική, στο πειραματικό στάδιο, θα χρησιμοποιηθούν τα συνελικτικά δίκτυα LeNet5, AlexNet, ResNet18 και ResNet50. Στον πίνακα 2.1 συνοψίζονται το πλήθος εκπαιδεύσιμων επιπέδων και παραμέτρων για καθένα από αυτά.



Εικόνα 2.7: Νευρωνικό δίκτυο με και χωρίς επίπεδο εγκατάλειψης



Εικόνα 2.8: Παράδειγμα αρχιτεκτονικής συνελικτικού νευρωνικού δικτύου

Νευρωνικού	**Εκπαιδεύσιμα επίπεδα**	**Παράμετροι**
LeNet-5	5	60,000
AlexNet	8	62,000,000
ResNet-18	18	11,000,000
ResNet-50	50	25,000,000

Πίνακας 2.1: Πλήθος επιπέδων και παραμέτρων ανά νευρωνικό

LeNet5

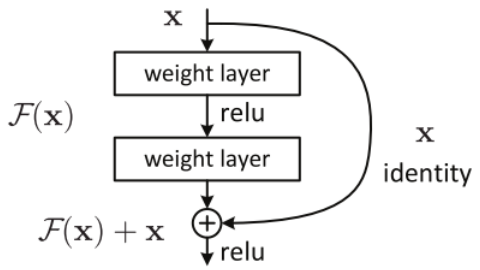
Το LeNet[19] αποτελείται από δύο συνελικτικά επίπεδα και δύο επίπεδα ενεργοποίησης που εναλλάσσονται το ένα με το άλλο, καταλήγοντας τελικά σε τρία πλήρως διασυνδεδεμένα επίπεδα.

AlexNet

Το AlexNet[20] περιλαμβάνει πέντε συνελικτικά επίπεδα, μερικά από αυτά ακολουθούμενα από επίπεδα ομαδοποίησης, και τρία πλήρως συνδεδεμένα επίπεδα.

ResNet

Το ResNet[21] είναι το συνελικτικό δίκτυο που παρουσιάζει το μεγαλύτερο ενδιαφέρον. Η κύρια ιδέα που εισάγει το ResNet είναι η σύνδεση συντόμευσης ταυτότητας (identity shortcut connection), η οποία είναι ουσιαστικά μία ακμή που μπορεί να προσπεράσει ένα ή περισσότερα επίπεδα του νευρωνικού. Έτσι, η βασική δομική μονάδα του ResNet, το λεγόμενο παραμένον μπλοκ (residual block) αποτελείται από μία αλληλουχία επιπέδων, η αρχική είσοδος των οποίων προστίθεται στην έξοδό της. Μία απεικόνιση του residual block φαίνεται στην εικόνα 2.9. Η αρίθμηση στην ονομασία των ResNet (π.χ. ResNet18, ResNet50) προκύπτει από το πλήθος των εκπαιδεύσιμων επιπέδων που περιέχουν.



Εικόνα 2.9: Residual block

2.3 Εκπαίδευση Νευρωνικών Δικτύων

Στη συγκεκριμένη ενότητα θα γίνει αναφορά στα βήματα που ακολουθούνται κατά την εκπαίδευση ενός νευρωνικού δικτύου, καθώς και στην έννοια της βελτιστοποίησης.

2.3.1 Βελτιστοποιητές

Η εκπαίδευση ενός νευρωνικού δικτύου στηρίζεται στη βελτιστοποίηση μιας συνάρτησης, η οποία ονομάζεται συνάρτηση κόστους. Συγκεκριμένα, το δίκτυο

δημιουργεί μια πρόβλεψη για το πρόβλημα, η οποία αξιολογείται μέσω της συνάρτησης κόστους ως προς την απόκλιση της από το επιθυμητό αποτέλεσμα. Στη συνέχεια υπολογίζεται η μερική παράγωγος/κλίση της συνάρτησης αυτής ως προς κάθε παράμετρο του νευρωνικού. Η κλίση αυτή χρησιμοποιείται για την ανανέωση των παραμέτρων του δικτύου, προκειμένου να μειωθεί η απόκλιση της πρόβλεψης από το πραγματικό αποτέλεσμα. Ειδικότερα στα προβλήματα κατηγοριοποίησης, η συνάρτηση κόστους που χρησιμοποιείται συχνότερα είναι η κατηγορική εγκάρσια εντροπία (categorical cross-entropy)[22].

Η βελτιστοποίηση της συνάρτησης κόστους, λοιπόν, είναι ένα πολύ σημαντικό κομμάτι της εκπαίδευσης του δικτύου, γι' αυτό και υπάρχουν πολλές επιλογές αλγορίθμων που μπορούν να αναλάβουν το έργο. Ωστόσο, δεν υπάρχει κάποιος αλγόριθμος που να υπερέχει των άλλων σε όλα τα προβλήματα κατηγοριοποίησης και συνεπώς η διαδικασία της βελτιστοποίησης πρέπει να ακολουθεί πιστά τις ανάγκες του εκάστοτε προβλήματος και να επικεντρώνεται στις ιδιαιτερότητές του. Στην παρούσα διπλωματική, ο αλγόριθμος που επελέγη είναι ο Στοχαστικός Αλγόριθμος Καθόδου Κλίσεων (Stochastic Gradient Descent - SGD)[23], ο οποίος ανήκει στην κατηγορία των αλγορίθμων καθόδου κλίσεων (Gradient Descent – GD)[24] και αποτελεί μια παραλλαγή του βασικού.

Ο Gradient Descent είναι ένας επαναληπτικός αλγόριθμος που αποσκοπεί στη σύγκλιση της συνάρτησης κόστους σε ένα ελάχιστο, τοπικό ή ολικό. Αυτή η σύγκλιση επιτυγχάνεται με εφαρμογή του τελεστή της κλίσης πάνω στη συνάρτηση κόστους, με εξάρτηση πάντα από τα βάρη του δικτύου. Η ταχύτητα σύγκλισης του αλγορίθμου αυτού προς το ελάχιστο ανά επανάληψη προκύπτει από μία παράμετρο του αλγορίθμου που ονομάζεται ρυθμός μάθησης (learning rate – η) και ουσιαστικά περιγράφει το μέγεθος του βήματος προς το ελάχιστο. Η εύρεση της ιδανικής τιμής γι αυτή την παράμετρο αποτελεί ένα πολύ σημαντικό στοιχείο για τον αλγόριθμο. Αν αυτό το βήμα είναι πολύ μεγάλο υπάρχει κίνδυνος κατά την αναζήτηση να υπερπηδήσουμε το ελάχιστο και αυτό να χαθεί. Αντίθετα, αν το βήμα είναι πολύ μικρό θα καθυστερήσει πολύ η σύγκλιση ή μπορεί ακόμα και να κολλήσει η αναζήτηση σε κάποιο ανεπιθύμητο τοπικό ελάχιστο.

Κάθε επανάληψη του αλγορίθμου χωρίζεται σε δύο βήματα. Αρχικά, για κάθε βάρος του νευρωνικού δικτύου w_{ij} υπολογίζεται μία κλίση g_{ij} ως η μερική παράγωγος της συνάρτησης κόστους L ως προς το βάρος αυτό, όπως φαίνεται και στην εξίσωση 2.2. Στην εξίσωση αυτή, t είναι η έξοδος-στόχος και y είναι η πραγματική έξοδος του μοντέλου. Ο υπολογισμός αυτός ξεκινάει από το τέλος του νευρωνικού δικτύου και συνεχίζει προς την αρχή του, γι' αυτό και ονομάζεται οπισθοδρόμηση (backpropagation). Οι μερικοί παράγωγοι των βαρών υπολογίζονται με χρήση του κανόνα της αλυσίδας. Το διάνυσμα που περιέχει όλες τις μερικές παραγώγους των βαρών του δικτύου ονομάζεται διάνυσμα κλίσης.

$$g_{ij} = \frac{\partial L(t, y)}{\partial w_{ij}} \tag{2.2}$$

Κατά το δεύτερο βήμα του αλγορίθμου γίνεται η ανανέωση των βαρών. Από κάθε

βάρος αφαιρείται το γινόμενο της μερικής παραγώγου που υπολογίστηκε προηγουμένως επί το ρυθμό μάθησης η, όπως περιγράφεται και από την εξίσωση 2.3.

$$w_{ij,new} = w_{ij} - \eta * g_{ij} \tag{2.3}$$

Ο βασικός Gradient Descent, όμως, εμφανίζει το μειονέκτημα ότι μπορεί να κολλήσει σε ένα τοπικό ελάχιστο και να μην καταφέρει να φτάσει σε ένα ολικό. Για αυτό το λόγο έχουν υλοποιηθεί παραλλαγές του που καταφέρνουν να ξεπερνούν αυτή τη δυσκολία. Μία τέτοια παραλλαγή είναι ο αλγόριθμος Stochastic Gradient Descent. Αυτός ο αλγόριθμος μπορεί να θεωρηθεί ως μια στοχαστική προσέγγιση του βασικού καθώς αντικαθιστά την πραγματική κλίση, που υπολογίζεται από ολόκληρο το σύνολο δεδομένων, με μια εκτίμηση κλίσης, που υπολογίζεται από ένα τυχαία επιλεγμένο σύνολο δεδομένων. Συγκεκριμένα, σε κάθε επανάληψη, χρησιμοποιείται μόνο ένα δείγμα από τα δεδομένα εκπαίδευσης, με αποτέλεσμα να μειώνεται σημαντικά το κόστος των υπολογισμών κάθε βήματος αλλά και να αυξάνεται ταυτόχρονα το πλήθος των βημάτων. Παρότι ο SGD παρουσιάζει μία αυξημένη διακύμανση στις τιμές της συνάρτησης κόστους λόγω της διαφοράς που παρουσιάζουν τα δεδομένα μεταξύ διαφορετικών επαναλήψεων, τείνει να συγκλίνει τελικά σε πραγματικό ολικό ελάχιστο.

Ο mini-batch SGD συνδυάζει τους δύο προηγούμενους αλγορίθμους και παρέχει μία μέση λύση. Σε κάθε βήμα εκτέλεσης επεξεργάζεται ένα μέρος (batch) των δειγμάτων του συνόλου δεδομένων. Ως συνέπεια, κρατά μικρή τη διακύμανση της συνάρτησης κόστους, μειωμένο το κόστος υπολογισμού παράλληλα με τη μείωση των βημάτων σε σχέση με τον SGD και μπορεί να συγκλίνει σε ικανοποιητικά τοπικά ή ολικά ελάχιστα.

2.3.2 Διαδικασία εκπαίδευσης

Η εκπαίδευση των νευρωνικών δικτύων αποτελεί μια επαναληπτική διαδικασία κατά την οποία εφαρμόζεται μια σειρά επιμέρους βημάτων πάνω στα δεδομένα εισόδου. Το πλήθος των φορών που θα επαναληφθούν τα βήματα για ολόκληρο το σύνολο των δεδομένων εισόδου καθορίζεται από μια υπερπαράμετρο του αλγορίθμου που ονομάζεται εποχή (epoch). Η διαδικασία αυτή τερματίζεται είτε όταν συμπληρωθεί ο απαιτούμενος αριθμός των εποχών που έχει οριστεί, είτε με την επαλήθευση μιας συνθήκης τερματισμού που δηλώνει την επίτευξη μιας ικανοποιητικής τιμής σε κάποια προκαθορισμένη μετρική[25].

Ο αλγόριθμος εκπαίδευσης, λοιπόν, ακολουθεί κατ'επανάληψη μία σειρά βημάτων που είναι η εξής:

• Εμπρόσθιο πέρασμα (forward pass) των δεδομένων εισόδου μέσα από το νευρωνικό συνήθως σε μικρο-ομάδες (batches) και υπολογισμός της ψευδο-πιθανότητας για κάθε στοιχείο.

• Εφαρμογή των ψευδο-πιθανοτήτων στη συνάρτηση κόστους και υπολογισμός της απόκλισης από τον πραγματικό στόχο.

• Αντίστροφο πέρασμα των δεδομένων (backward pass) μέσω του κανόνα αλυσίδας για τον υπολογισμό της συνεισφοράς του κάθε νευρώνα (μερική παράγωγος κόστους ως προς τα βάρη) στη συνάρτηση κόστους.

• Ενημέρωση των παραμέτρων του δικτύου με χρήση της υπολογισμένης κλίσης προς την κατεύθυνση που μειώνεται η συνάρτηση κόστους.

2.4 Κατανεμημένη Εκπαίδευση Δικτύων

Η σημερινή εποχή χαρακτηρίζεται από ολοένα και αυξανόμενο πλήθος δεδομένων, επειδή αυτά συλλέγονται με εύκολο τρόπο από διάφορες συσκευές που είναι συνδεδεμένες στο διαδίκτυο. Ο χειρισμός αυτού του τεράστιου όγκου δεδομένων, όπως είναι εμφανές, απαιτεί πόρους και νέες καινοτομίες ώστε να υποστηριχτούν οι αλγόριθμοι μηχανικής μάθησης ευρείας κλίμακας. Μία πρώτη κίνηση προς αυτή την κατεύθυνση ήταν η χρήση των συσκευών GPU για την εκπαίδευση νευρωνικών δικτύων, καθώς αυτές υπερέχουν σημαντικά έναντι των απλών CPU σε ταχύτητα επεξεργασίας. Στην περίπτωση, όμως, ενός μεγάλου μοντέλου που δε χωρά να φορτωθεί στη μνήμη της GPU, η πραγματική επιτάχυνση είναι πολύ μικρή σε σχέση με την αναμενόμενη κι επιθυμητή. Έτσι, τελικά εμφανίστηκε μια νέα καινοτομία που υπέδειξε ως εναλλακτική λύση της εύρεσης πιο γρήγορων μονάδων επεξεργασίας, την προσθήκη περισσότερων μηχανημάτων που θα συνεργάζονται μεταξύ τους για την επίτευξη του κοινού τους στόχου. Αυτές οι ομάδες μηχανημάτων, λοιπόν, διαμοιράζονται το φορτίο τόσο σε επίπεδο μοντέλου όσο και σε επίπεδο δεδομένων και με αυτόν τον τρόπο επιταχύνουν σημαντικά τη διαδικασία, διατηρώντας μάλιστα την κλιμακωσιμότητα του συστήματος.

2.4.1 Τεχνικές παραλληλοποίησης

Όπως αναφέρθηκε προηγουμένως η κατανομή των εργασιών που σχετίζονται με την εκπαίδευση ενός νευρωνικού δικτύου αφορά σε δύο διαφορετικά επίπεδα, το επίπεδο δεδομένων και το επίπεδο μοντέλου. Μία οπτικοποίηση των τεχνικών αυτών, οι οποίες αναλύονται στη συνέχεια, παρέχεται στην εικόνα 2.10 για ένα πρόβλημα ταξινόμησης χειρόγραφων ψηφίων.

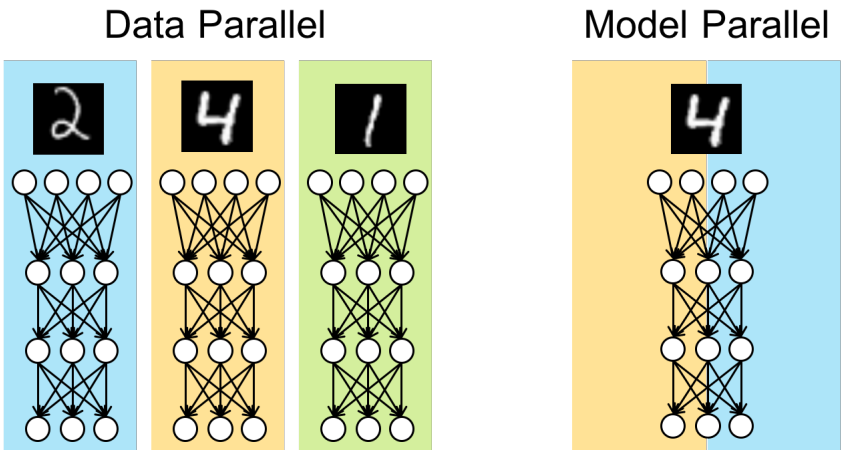
Παραλληλοποίηση δεδομένων (data parallelism)

Η παραλληλοποίηση των δεδομένων[26] περιλαμβάνει το χωρισμό των δεδομένων εισόδου σε ισομεγέθη τμήματα και το διαμοιρασμό τους μεταξύ των κόμβων του συστήματος. Κάθε κόμβος κρατάει ένα δικό του αντίγραφο του μοντέλου - νευρωνικού και ανανεώνει τα βάρη του όχι μόνο με βάση τα δεδομένα που έχει αναλάβει να επεξεργαστεί, αλλά και χρησιμοποιώντας και τις ανανεώσεις των βαρών

που προκύπτουν από το έργο που παράγουν και οι υπόλοιποι κόμβοι.

Παραλληλοποίηση μοντέλου (model parallelism)

Η παραλληλοποίηση μοντέλου[27], αντίθετα, επιμερίζει τα διάφορα τμήματα της αρχιτεκτονικής του μοντέλου σε διαφορετικούς κόμβους και με αυτόν τον τρόπο κατανέμει τη διαδικασία της εκπαίδευσης. Αυτό το επίπεδο παραλληλισμού προτιμάται κυρίως στην περίπτωση που το μοντέλο είναι πολύ μεγάλο και δε χωρά στη μνήμη ενός μόνο μηχανήματος.



Εικόνα 2.10: Παραλληλοποίηση δεδομένων και μοντέλου

2.4.2 Λειτουργίες συγχρονισμού

Κατά την κατανεμημένη εκτέλεση παραλληλοποίησης δεδομένων είναι απαραίτητο να υπάρχει μία επικοινωνία μεταξύ των διαφορετικών κόμβων ώστε να εκθέτει ο καθένας τους τα διανύσματα κλίσης που έχει υπολογίσει ή τις παραμέτρους στις οποίες έχει καταλήξει στους υπόλοιπους κόμβους. Για την επικοινωνία αυτή υφίστανται δύο λειτουργίες συγχρονισμού, η σύγχρονη και η ασύγχρονη, οι οποίες περιγράφονται παρακάτω.

Σύγχρονη λειτουργία

Στη σύγχρονη λειτουργία υπάρχει ένα σημείο συγχρονισμού μεταξύ όλων των κόμβων-εργατών (workers), ακριβώς αφότου υπολογιστούν τα διανύσματα κλίσης. Στο σημείο προστίθεται μία καθυστέρηση ώστε να ολοκληρώσουν την επεξεργασία τους οι πιο αργοί κόμβοι του κατανεμημένου συστήματος (cluster). Στη συνέχεια προκύπτει ένα μέσο διάνυσμα κλίσης από τις επιμέρους κλίσεις που έχει υπολογίσει ο κάθε worker ξεχωριστά. Το κοινό αυτό διάνυσμα χρησιμοποιείται για την

ανανέωση όλων των παραμέτρων που έχει στο τοπικό του μοντέλο ο κάθε worker, εξασφαλίζοντας έτσι ότι μετά από κάθε επανάληψη τα μοντέλα των κόμβων θα έχουν τελικά τις ίδιες παραμέτρους. Ο υπολογισμός του μέσου διανύσματος κλίσης μπορεί να γίνει είτε κεντρικά από ένα διακομιστή παραμέτρων (parameter server), δηλαδή ξεχωριστή διεργασία που τρέχει σε έναν ή περισσότερους κόμβους του cluster, είτε να το υπολογίσει ο κάθε worker μόνος του όταν συλλέξει τα διανύσματα κλίσης από όλους τους άλλους workers.

Ασύγχρονη λειτουργία

Σε αντίθεση με την σύγχρονη λειτουργία, στην ασύγχρονη η ύπαρξη ενός τουλάχιστον parameter server είναι απαραίτητη. Οι παράμετροι του ολικού μοντέλου βρίσκονται αποθηκευμένες στον parameter server. Η διαδικασία από την πλευρά του worker είναι η εξής. Εκτελεί μία τυπική επανάληψη και υπολογίζει το διάνυσμα κλίσης για το batch που επεξεργάστηκε. Στη συνέχεια στέλνει το διάνυσμα κλίσης στον parameter server, ο οποίος το χρησιμοποιεί για να ανανεώσει τις παραμέτρους που έχει ο ίδιος αποθηκευμένες. Όταν γίνει η ανανέωση αυτή, ο worker λαμβάνει τις ανανεωμένες παραμέτρους τις οποίες και χρησιμοποιεί για να εκτελέσει την επόμενη επανάληψη μάθησης. Η κύρια διαφορά σε αυτή τη λειτουργία είναι ότι ο worker δε χρειάζεται να περιμένει όλους τους άλλους κόμβους κάθε φορά, αλλά μπορεί να συνεχίσει την εκτέλεσή του ανεξάρτητα από αυτούς.

Σύγκριση λειτουργιών συγχρονισμού

Όπως είναι φυσικό, οι δύο αυτές λειτουργίες έχουν τα πλεονεκτήματα και τα μειονεκτήματά τους και το ποια υπερέχει της άλλης πρέπει πάντα να καθορίζεται για το συγκεκριμένο πρόβλημα.

Τα σημαντικότερα σημεία σύγκρισης που αξίζει να σημειωθούν είναι τα εξής:

• *Καθυστέρηση συγχρονισμού (synchronization overhead)*

Κατά τη σύγχρονη λειτουργία, υπάρχει ένα επιπρόσθετο κόστος στο τέλος κάθε επανάληψης ώστε να συγχρονιστούν οι παράμετροι μεταξύ των διαφορετικών κόμβων. Άρα στην ασύγχρονη λειτουργία, όπου η καθυστέρηση αυτή δεν υφίσταται, ο ίδιος αριθμός επαναλήψεων θα εκτελεστεί σίγουρα σε μικρότερο χρονικό διάστημα.

• *Ρυθμός σύγκλισης*

Στην ασύγχρονη λειτουργία, ακριβώς επειδή ο κάθε κόμβος δεν περιμένει τους υπόλοιπους να υπολογίσουν το διάνυσμα κλίσης, τα βάρη του νευρωνικού (κυρίως για τους πιο γρήγορους κόμβους) δε θα είναι πλήρως ανανεωμένα. Έτσι, θα χρειαστούν περισσότερες επαναλήψεις σε σχέση με τη σύγχρονη λειτουργία ώστε να βρεθεί το σημείο σύγκλισης. Όπως είναι φανερό, ο ρυθμός σύγκλισης και η ταχύτητα εκτέλεσης ανά επανάληψη αποτελούν ένα trade-off.

• *Αντιμετώπιση αποτυχίας κόμβου*

Σε περίπτωση που ένας κόμβος πέσει έχουμε τις εξής περιπτώσεις. Για την

σύγχρονη λειτουργία αναμένεται από όλους τους κόμβους του cluster να μοιραστούν το διάνυσμα κλίσης. Επομένως η διαδικασία δε θα μπορεί να συνεχιστεί και πιθανώς μετά από κάποιο χρονικό όριο (timeout) να τερματιστεί. Στην ασύγχρονη λειτουργία η απουσία ενός κόμβου δε θα αποτελέσει πρόβλημα στη διαδικασία, αλλά φυσικά θα έχει επιπτώσεις στην ταχύτητα σύγκλισης.

2.4.3 Κατανεμημένες Αρχιτεκτονικές Εκπαίδευσης

Δύο ευρεία χρησιμοποιούμενες αρχιτεκτονικές που υποστηρίζουν την κατανεμημένη εκπαίδευση νευρωνικών δικτύων είναι η αρχιτεκτονική Διακομιστή Παραμέτρων (Parameter Server) και η τεχνική All-Reduce.

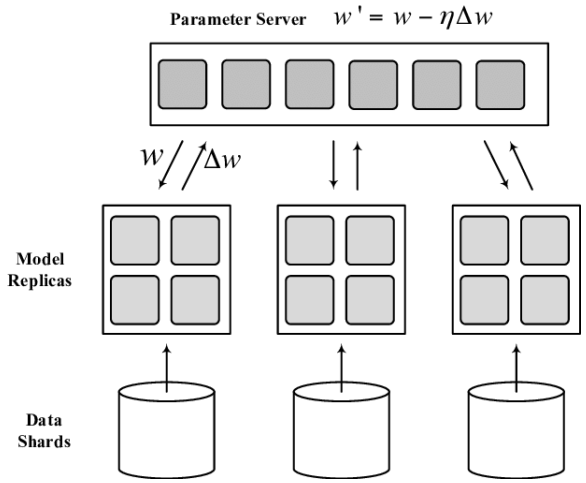
Parameter Server

Στην αρχιτεκτονική Parameter Server[28] κάποιοι από τους κόμβους λειτουργούν ως διακομιστές παραμέτρων και κάποιοι ως εργάτες (workers). Οι parameter servers αποθηκεύουν τις παραμέτρους του νευρωνικού μοντέλου και είναι συνήθως υπεύθυνοι για την ανανέωση τους όταν λάβουν τα διανύσματα κλίσης από τους workers αλλά και την αποστολή τους πίσω σε αυτούς. Οι workers είναι υπεύθυνοι για τον κύριο υπολογισμό που συμβαίνει κατά την εκπαίδευση του νευρωνικού, δηλαδή για το forward pass, το backward pass και άρα τον υπολογισμό των κλίσεων. Στη συνέχεια στέλνουν τα διανύσματα κλίσης στους parameter servers και περιμένουν να λάβουν πίσω τα ανανεωμένα βάρη του δικτύου, ώστε να συνεχίσουν με την επεξεργασία του επόμενου batch δεδομένων. Σε ορισμένες περιπτώσεις υπάρχει ακόμα ένα είδος διεργασίας, που συνήθως είναι μοναδικό ανά cluster, ο λεγόμενος δρομολογητής (scheduler). Κάποια από τα καθήκοντα του δρομολογητή είναι ο διαμοιρασμός του συνόλου δεδομένων μεταξύ των εργατών καθώς επίσης και η ανάθεση εργασιών σε αυτούς. Στην εικόνα 2.11 φαίνεται η αρχιτεκτονική ενός parameter server συστήματος.

Τεχνική All-Reduce

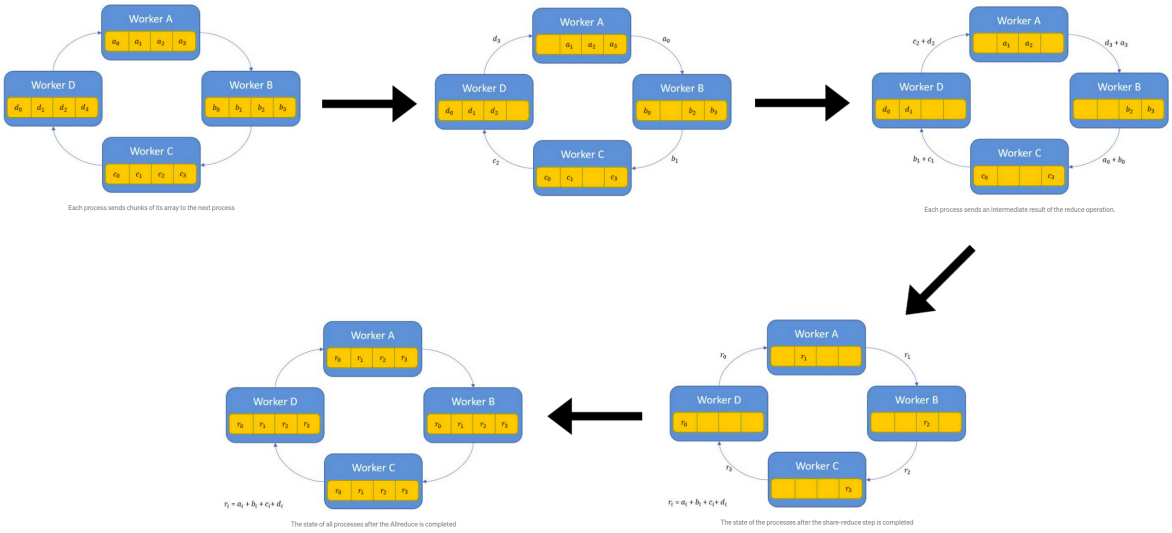
Μία διαφορετική προσέγγιση της επικοινωνίας μεταξύ των κόμβων εμφανίζεται με τους αλγορίθμους All-Reduce, οι οποίοι εισάγουν την καινοτομία της ανταλλαγής δεδομένων από κόμβο σε κόμβο (peer-to-peer). Αυτοί οι αλγόριθμοι χρησιμοποιούνται ευρέως στη σύγχρονη εκπαίδευση νευρωνικών δικτύων. Σε αυτή την υλοποίηση οι συμμετέχοντες κόμβοι μοιράζονται όλα τα δεδομένα (στην περίπτωση μας τα διανύσματα κλίσης) μεταξύ τους και μετά από μία πράξη (π.χ. μέσος όρος) πάνω σε αυτά παράγουν ένα αποτέλεσμα (μέσο διάνυσμα κλίσης) το οποίο αναμεταδίδουν στους υπόλοιπους.

Οι υλοποιήσεις του All-Reduce διαφοροποιούνται συνήθως στον τρόπο με τον οποίο οι κόμβοι επικοινωνούν τα δεδομένα τους στο υπόλοιπο δίκτυο. Ο πιο ευρεία χρησιμοποιούμενος αλγόριθμος της κατηγορίας All Reduce,τον οποίο αξιοποιούν



Εικόνα 2.11: Αρχιτεκτονική διακομιστή παραμέτρων

γνωστές βιβλιοθήκες μηχανικής μάθησης όπως το TensorFlow είναι ο Ring All-Reduce[29]. Στην πρώτη φάση αυτού του αλγορίθμου εκτελείται η διαδικασία share-reduce, κατά την οποία κάθε κόμβος i του δικτύου στέλνει δεδομένα στον επόμενό κόμβο, δηλαδή τον $(i+1)\%n$, όπου n το πλήθος των κόμβων του δικτύου και «%» ο τελετής υπολοίπου modulo. Κατόπιν, κάθε κόμβος εφαρμόζει την πράξη στο τμήμα των δεδομένων που κατέχει και το στέλνει στον επόμενο κόμβο για να συνεχιστεί η διαδικασία, με αυτή την αλληλουχία των ενεργειών να επαναλαμβάνεται $n-1$ φορές. Στη δεύτερη φάση του αλγορίθμου εκτελείται η διαδικασία share-only, κατά την οποία κάθε κόμβος μοιράζεται το τμήμα του αποτελέσματος που διαθέτει με όλους τους υπόλοιπους μέσω μετάδοσης, χωρίς να εκτελεί κάποια πράξη πάνω στα δεδομένα. Οι συγκεκριμένες ενέργειες αποστολής των δεδομένων, μάλιστα, επαναλαμβάνονται επίσης $n-1$ φορές, όπως και οι αντίστοιχες της πρώτης φάσης. Οπτικά οι ακμές που δημιουργούνται μεταξύ των κόμβων κατά τον τρόπο επικοινωνίας που περιγράφτηκε σχηματίζουν έναν κύκλο-δαχτυλίδι (ring), από όπου και παίρνει το όνομά της η συγκεκριμένη υλοποίηση του αλγορίθμου. Μία απεικόνιση του Ring All-Reduce φαίνεται στην εικόνα 2.12.



Εικόνα 2.12: Διαδικασία Ring All-Reduce

Κεφάλαιο 3

Συστήματα Εκπαίδευσης Νευρωνικών Δικτύων

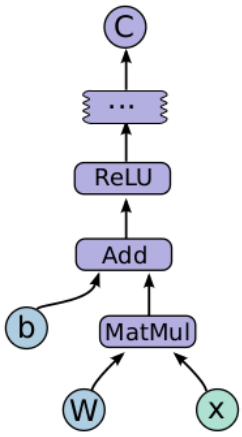
3.1 TensorFlow

Το TensorFlow[30] είναι ένα σύστημα για υλοποίηση και ανάπτυξη μοντέλων μηχανικής μάθησης σε μεγάλη κλίμακα. Αναπτύχθηκε από την Google, και συγκεκριμένα αποτελεί προϊόν του πρότζεκτ Google Brain, ως διάδοχος του DistBelief[31]. Ο υπολογισμός στο TensorFlow περιγράφεται από ένα μοντέλο ροής δεδομένων, και μπορεί να εκτελεστεί σε ευρεία ποικιλία (hardware) πλατφορμών που περιέχουν από εκτέλεση συμπερασμού (inference) σε συσκευές κινητής τηλεφωνίας (Android, iOS) μέχρι και εκπαίδευση μοντέλου μηχανικής μάθησης σε εκατοντάδες εξειδικευμένα μηχανήματα με χιλιάδες κάρτες γραφικών (GPUs).

3.1.1 Δομή

Ο υπολογισμός στο TensorFlow περιγράφεται από έναν κατευθυνόμενο γράφο που αποτελείται από ένα σύνολο κόμβων και ακμών. Ο γράφος αναπαριστά έναν υπολογισμό ροής δεδομένων και επιτρέπει σε κάποιους κόμβους να διατηρούν και να ανανεώνουν εσωτερική κατάσταση, ενώ υποστηρίζει ακόμα διακλάδωση και βρόχους. Το TensorFlow είναι υλοποιημένο (backend) σε C++. Για να κατασκευάσει ένας χρήστης (client) έναν υπολογιστικό γράφο μπορεί να χρησιμοποιήσει μία από τις γλώσσες προγραμματισμού στις οποίες υπάρχουν διεπαφές με το backend του TensorFlow (π.χ. python, R). Ο καθένας από τους κόμβους σε έναν γράφο του TensorFlow έχει μηδέν ή παραπάνω εισόδους και μηδέν ή παραπάνω εξόδους και επιτελεί μία βασική λειτουργία (operation). Οι τιμές που ρέουν στις κανονικές ακμές του γράφου λέγονται τανυστές (tensors). Στον γράφο μπορεί να υπάρχουν επίσης ειδικές ακμές, ονομαζόμενες εξαρτήσεις ελέγχου, στις οποίες δε ρέουν δεδομένα. Οι ακμές αυτές δηλώνουν ότι ο κόμβος-πηγή πρέπει να εκτελεστεί προτού ο κόμβος-προορισμός ξεκινήσει να εκτελείται. Ένα παράδειγμα υπολογιστικού γράφου

του TensorFlow φαίνεται στην εικόνα 3.1.



Εικόνα 3.1: Υπολογιστικός γράφος TensorFlow

Λειτουργία και Πυρήνες

Μία λειτουργία (operation) αναπαριστά έναν αφαιρετικό υπολογισμό. Οι συγκεκριμένες υλοποιήσεις ενός operation λέγονται πυρήνες (kernel) και μπορούν να τρέξουν σε συγκεκριμένου τύπου συσκευές (π.χ. CPU ή GPU). Για την αλληλεπίδραση μεταξύ των client προγραμμάτων και του TensorFlow χρησιμοποιείται μία συνεδρία (Session). Η κυριότερη λειτουργία του Session είναι η εκτέλεση (Run), η οποία δέχεται ένα σύνολο από ονόματα κόμβων που πρέπει να υπολογιστούν, καθώς επίσης και ένα προαιρετικό σύνολο τανυστών (tensors) για να τροφοδοτήσουν το γράφο σε εξόδους συγκεκριμένων κόμβων. Συνήθως ένας γράφος εκτελείται πολλαπλές φορές κατά τη διάρκεια ενός προγράμματος. Οι περισσότεροι tensors δεν επιβιώνουν πάνω από μία εκτελέσεις του γράφου. Υπάρχει όμως ένα operation ειδικού τύπου, ονομαζόμενο μεταβλητή (variable), που επιστρέφει μία αναφορά σε tensor του οποίου η τιμή μπορεί να ανανεωθεί και να διατηρηθεί ανάμεσα σε εκτελέσεις του γράφου. Οι μεταβλητές αυτές χρησιμοποιούνται συνήθως, στο πλαίσιο εφαρμογών μηχανικής μάθησης, ως οι παράμετροι του μοντέλου.

Βασικές Συνιστώσες

Οι βασικές συνιστώσες (components) στο TensorFlow είναι ο client, που επικοινωνεί με τον αφέντη (master), και μία ή παραπάνω διεργασίες-εργάτες (workers), οι οποίες είναι υπεύθυνες για τον διαμοιρασμό του υπολογιστικού φορτίου στις διαθέσιμες υπολογιστικές συσκευές (όπως πυρήνες επεξεργαστή ή κάρτες γραφικών) και την εκτέλεση των κόμβων πάνω σε αυτές, σύμφωνα με τις

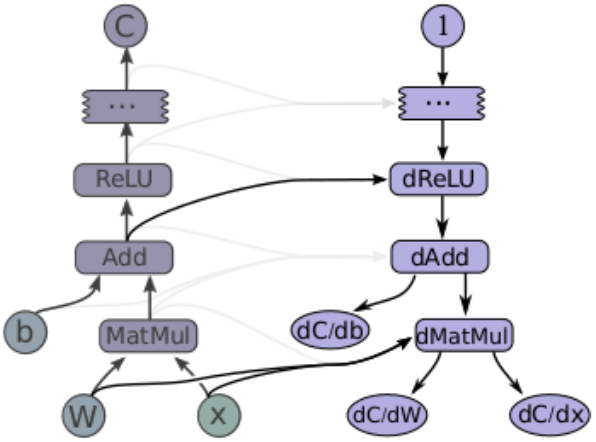
οδηγίες του master. Υπάρχει, μάλιστα, τόσο τοπική όσο και κατανεμημένη υλοποίηση του TensorFlow.

3.1.2 Υπολογισμός κλίσης

Πολλοί αλγόριθμοι βελτιστοποίησης, συμπεριλαμβανομένων κλασικών αλγορίθμων εκπαίδευσης μηχανικής μάθησης όπως η στοχαστική κατάβαση κλίσης (stochastic gradient descent, SGD), υπολογίζουν την κλίση μίας συνάρτησης κόστους ως προς ένα σύνολο εισόδων. Επειδή αυτό αποτελεί επιτακτική ανάγκη στο πεδίο της μηχανικής μάθησης, το TensorFlow υποστηρίζει αυτόματο υπολογισμό κλίσης. Αν ένας tensor C στον γράφο του TensorFlow εξαρτάται, πιθανώς μέσω ενός πολύπλοκου υπογράφου από operations, από κάποιο σύνολο από tensors X_k, τότε υπάρχει έτοιμη συνάρτηση του TensorFlow που θα επιστρέψει τους tensors των επιμέρους κλίσεων $\frac{dC}{dX_k}$. Οι tensors κλίσης υπολογίζονται, όπως και οι υπόλοιποι tensors, με την επέκταση του γράφου του TensorFlow, ως εξής: Για τον υπολογισμό της κλίσης ενός tensor C ως προς κάποιον tensor I από τον οποίο ο C εξαρτάται, το TensorFlow βρίσκει αρχικά το μονοπάτι στον υπολογιστικό γράφο από τον I στον C. Ύστερα οπισθοδρομεί από τον C στον I, και για κάθε operation στο αντίστροφο (backward) μονοπάτι προσθέτει έναν κόμβο στον γράφο του TensorFlow, συνθέτοντας έτσι τις επιμέρους κλίσεις ενώ διασχίζει το αντίστροφο μονοπάτι χρησιμοποιώντας τον κανόνα της αλυσίδας. Ο κάθε κόμβος που προστίθεται υπολογίζει τη συνάρτηση κλίσης για το αντίστοιχο operation του κανονικού (forward) μονοπατιού. Αυτή η συνάρτηση παίρνει ως είσοδο όχι μόνο τις επιμέρους κλίσεις που έχουν ήδη υπολογιστεί κατά το backward μονοπάτι, αλλά επίσης προαιρετικά και τις εισόδους και εξόδους του αντίστοιχου operation κατά το forward μονοπάτι. Γενικότερα ένα oparation μπορεί να έχει πολλαπλές τιμές εξόδου και το αποτέλεσμα της συνάρτησης κόστους να εξαρτάται μόνο από κάποιες από αυτές. Οι τιμές εξόδου από τις οποίες δεν εξαρτάται το κόστος θα έχουν και αμελητέα συνεισφορά στον υπολογισμό της επιμέρους κλίσης του συγκεκριμένου κόμβου. Στην εικόνα 3.2 φαίνεται ο γράφος υπολογισμού κλίσεων για τον γράφο της εικόνας 3.1.

3.1.3 TensorFlow 2

Πρόσφατα εκδόθηκε μία νέα έκδοση του TensorFlow (TensorFlow 2)[32], η οποία περιέχει σημαντικές αλλαγές σε σχέση με το TensorFlow 1. Στο TensorFlow 2 έχει γίνει μία ανακατασκευή και αφαίρεση κάποιων προγραμματιστικών διεπαφών (API), καθώς επίσης έχει προστεθεί ανυπόμονη εκτέλεση (μοντέλο προστακτικού προγραμματισμού) ώστε να γίνεται καλύτερη ενσωμάτωση με την Python. Οι κύριες αλλαγές συνοψίζονται παρακάτω.



Εικόνα 3.2: Γράφος υπολογισμού κλίσης για τον γράφο της εικόνας 3.1

Καθαρισμός API

Πολλά API έχουν είτε μετακινηθεί είτε αφαιρεθεί τελείως στο TensorFlow 2. Για παράδειγμα, διάφορες συναρτήσεις που υπήρχαν κατευθείαν στο πακέτο tf.* έχουν μετακινηθεί σε υπό-πακέτα όπως το tf.math.

Ανυπόμονη εκτέλεση

Το TensorFlow 1 απαιτούσε από τους χρήστες να δημιουργήσουν χειρωνακτικά ένα αφαιρετικό συντακτικό δέντρο (τον υπολογιστικό γράφο). Ύστερα απαιτούσε από τους χρήστες να τρέξουν χειρωνακτικά το γράφο περνώντας ένα σύνολο από τανυστές εξόδου και εισόδου σε μία κλήση run() του Session. Το TensorFlow 2 εκτελείται ανυπόμονα (όπως δουλεύει κανονικά και η Python), αλλά η δυνατότητα για στατική εκτέλεση σε λειτουργία γράφου έχει παραμείνει για λόγους απόδοσης.

Function αντί για Session

Μία κλήση session.run() είναι σχεδόν σαν μία κλήση συνάρτησης (function): Προσδιορίζεις τις εισόδους και τη συνάρτηση που πρέπει να κληθεί, και επιστρέφει ένα σύνολο από εξόδους. Στο TensorFlow 2, μία συνάρτηση της Python μπορεί να διακοσμηθεί χρησιμοποιώντας tf.function() ώστε να μη μαρκαριστεί για μεταγλώττιση της στιγμής (just-in-time, JIT) και το TensorFlow να την τρέξει ως γράφο. Αυτός ο μηχανισμός επιτρέπει στο TensorFlow 2 να κερδίσει πολλά από τα οφέλη της λειτουργίας γράφου, όπως βελτιστοποίηση της απόδοσης (μέσω κλαδέματος του γράφου, ένωση των kernel κλπ).

3.1.4 Keras

Το Keras είναι ένα υψηλού επιπέδου (high-level) API του TensorFlow 2. Αποτελεί μία προσιτή και παραγωγική διεπαφή για επίλυση προβλημάτων μηχανικής μάθησης, με έμφαση στη σύγχρονη βαθιά μηχανική μάθηση. Παρέχει ουσιώδεις αφαιρέσεις και βασικά δομικά μπλοκ για ανάπτυξη και παράδοση λύσεων μηχανικής μάθησης σε γρήγορους ρυθμούς. Οι κύριες δομές του Keras είναι τα επίπεδα νευρωνικών δικτύων (layers) και τα μοντέλα (models).

3.1.5 Κατανεμημένη Εκτέλεση

Το tf.distribute.Strategy (Strategy)[33] είναι ένα API του TensorFlow που έχει ως σκοπό να κατανείμει την εκπαίδευση νευρωνικών μεταξύ πολλαπλών συσκευών (CPUs, GPUs, TPUs) και μηχανημάτων. Η σχεδίαση του έχει γίνει με γνώμονα τρεις βασικούς στόχους:

- Να είναι εύχρηστο και να μπορεί να υποστηρίζει πολλαπλές ομάδες χρηστών (π.χ. ερευνητές, μηχανικούς μηχανικής μάθησης (machine learning engineers) κλπ).
- Να παρέχει καλή απόδοση από μόνο του.
- Να γίνεται εύκολα εναλλαγή μεταξύ διαφορετικών στρατηγικών (υλοποιήσεων του Strategy API).

Το Strategy API μπορεί να χρησιμοποιηθεί με ένα API υψηλού επιπέδου όπως το Keras, αλλά και με προσαρμοζόμενους (custom) βρόχους εκπαίδευσης. Στο TensorFlow 2, μπορούν να εκτελεστούν προγράμματα είτε eagerly, είτε με γράφο. Το Strategy υποστηρίζει και τις δύο λειτουργίες, αλλά δουλεύει καλύτερα με τη δεύτερη.

Για να καλύψει διάφορες περιπτώσεις χρήσης (use cases), το Strategy παρέχει έξι διαθέσιμες στρατηγικές, ονομαστικά: MirroredStrategy, TPUStrategy, MultiWorkerMirroredStrategy, CentralStorageStrategy και ParameterServerStrategy.

MirroredStrategy

Η στρατηγική tf.distribute.MirroredStrategy υποστηρίζει σύγχρονη κατανεμημένη εκπαίδευση σε πολλαπλές συσκευές (GPUs) σε ένα μηχάνημα. Δημιουργεί ένα αντίγραφο ανά συσκευή. Κάθε μεταβλητή του μοντέλου καθρεπτίζεται μεταξύ όλων των αντιγράφων. Όλες αυτές οι μεταβλητές μαζί σχηματίζουν μία ουσιαστικά μονή καθρεπτιζόμενη μεταβλητή (MirroredVariable). Αυτές οι μεταβλητές είναι σε συγχρονισμό μεταξύ τους μέσω εφαρμογής ταυτόσημων ανανεώσεων. Για την επικοινωνία των ανανεώσεων των μεταβλητών μεταξύ των συσκευών χρησιμοποιούνται αποδοτικοί all-reduce αλγόριθμοι.

TPUStrategy

Η στρατηγική tf.distribute.TPUStrategy επιτρέπει στο χρήστη να τρέξει μοντέλα TensorFlow πάνω σε ειδικές υπολογιστικές συσκευές, ονομαζόμενες Tensor Processing Units (TPUs). Τα TPUs είναι ειδικές συσκευές ASIC (Application-specific integrated circuits) της Google που σχεδιάστηκαν για να επιταχύνουν δραματικά τη διαδικασία της μηχανικής μάθησης. Όσον αφορά την αρχιτεκτονική της κατανεμημένης εκτέλεσης, η TPUStrategy είναι ίδια με την MirroredStrategy.

MultiWorkerMirroredStrategy

Η στρατηγική tf.distribute.MultiWorkerMirroredStrategy είναι πολύ παρόμοια με την MirroredStrategy. Υλοποιεί σύγχρονη κατανεμημένη εκπαίδευση μεταξύ πολλαπλών workers (και σε διαφορετικά μηχανήματα), ο καθένας με πιθανώς πολλαπλές συσκευές. Ομοίως με την MirroredStrategy, δημιουργεί αντίγραφα όλων των μεταβλητών του μοντέλου σε κάθε συσκευή και σε κάθε worker. Αυτά τα αντίγραφα είναι σε πλήρη συγχρονισμό μεταξύ τους, αφού δέχονται ταυτόσημες ενημερώσεις. Η στρατηγική αυτή παρέχει δύο υλοποιήσεις για επικοινωνία μεταξύ συσκευών. Η πρώτη ονομάζεται RING, βασίζεται σε πρωτόκολλο RPC (συγκεκριμένα χρησιμοποιείται το gRPC[34]) και υποστηρίζει τόσο CPU όσο και GPU συσκευές. Η δεύτερη, ονομαζόμενη NCCL, χρησιμοποιεί NCCL και παρέχει state-of-the-art απόδοση σε GPU συσκευές, αλλά δεν υποστηρίζει CPU συσκευές.

ParameterServerStrategy

Η εκπαίδευση με αρχιτεκτονική parameter server είναι μία κοινή μέθοδος παραλληλοποίησης δεδομένων με στόχο την κλιμάκωση της εκπαίδευσης μοντέλων σε πολλαπλά μηχανήματα. Στην αρχιτεκτονική αυτή, ένα cluster αποτελείται από workers και parameter servers. Οι μεταβλητές δημιουργούνται στους parameter servers, ενώ διαβάζονται και ανανεώνονται από τους workers σε κάθε βήμα. Στην περίπτωση του TensorFlow 2 χρησιμοποιείται μία αρχιτεκτονική βασισμένη σε έναν κεντρικό συντονιστή (coordinator). Στην υλοποίηση αυτή, οι workers και οι parameter servers περιμένουν να τους ανατεθούν εργασίες (tasks) από τον coordinator. Οι workers διαβάζουν τις μεταβλητές από τους parameter servers ξεχωριστά ο καθένας. Γι' αυτό το λόγο, λοιπόν, η επικοινωνία στην υλοποίηση αυτή λέγεται ασύγχρονη. Ο συντονιστής δημιουργεί πόρους, αναθέτει tasks εκπαίδευσης, αποθηκεύει χρονικές στιγμές του μοντέλου (checkpoints) και αντιμετωπίζει πιθανά σφάλματα.

CentralStorageStrategy

Η στρατηγική tf.distribute.experimental.CentralStorageStrategy εκτελεί επίσης σύγχρονη εκπαίδευση. Οι μεταβλητές δεν καθρεπτίζονται, αλλά αντιθέτως

τοποθετούνται στην CPU και όλα τα operations κατανέμονται σε όλες τις διαθέσιμες υπολογιστικές συσκευές (GPUs ή CPUs). Αν υπάρχει μόνο τέτοια συσκευή, τότε όλα τα operations θα τοποθετηθούν σε αυτήν.

3.2 PyTorch

Το PyTorch δημιουργήθηκε με στόχο να κάνει τη διεπαφή με το χρήστη πιο εύκολη, χρησιμοποιώντας δυναμική ανυπόμονη (eager) εκτέλεση αντί για ορισμό ολόκληρου του υπολογιστικού γράφου από την αρχή (όπως είδαμε νωρίτερα στο TensorFlow). Επιπλέον, σκοπός ήταν πέρα από το να γίνει πιο φιλικό προς το χρήστη, να αποφύγει και τα κόστη που αυτό επέφερε στην απόδοση, όπως είχε συμβεί σε αντίστοιχες περιπτώσεις (Chainer).

Το PyTorch, με σκοπό την ισορροπία ανάμεσα σε ταχύτητα και ευχρηστία, ακολουθεί τέσσερις βασικές σχεδιαστικές αρχές:

- Να είναι pythonic: Καθότι η python (μαζί με τα εργαλεία που τη συνοδεύουν) είναι η πιο διάσημη γλώσσα προγραμματισμού για εφαρμογές μηχανικής μάθησης, το PyTorch κρατάει τη διεπαφή με το χρήστη απλή και σε συνέπεια με τους ιδιωματισμούς αυτής της γλώσσας. Επίσης ενσωματώνεται από τη φύση του με στάνταρ εργαλεία απεικόνισης, εντοπισμού σφαλμάτων (debugging), και επεξεργασίας δεδομένων.

- Να βάλει τους ερευνητές πρώτους: Το PyTorch προσπαθεί να κάνει τη διαδικασία δημιουργίας μοντέλων, φορτωτών δεδομένων (data loaders) και βελτιστοποιητών όσο το δυνατόν πιο εύκολη και παραγωγική. Παρέχει ενστικτώδεις διεπαφές προγραμματισμού εφαρμογών (Application Programming Interface, API), τα οποία προστατεύουν από παρενέργειες και απροσδόκητες πτώσεις στην απόδοση και αποφεύγουν όλη την πολυπλοκότητα που συνδέεται με τη μηχανική μάθηση στο backend του PyTorch.

- Να παρέχει ρεαλιστική απόδοση: Το PyTorch προσπαθεί να προσφέρει ελκυστική απόδοση, χωρίς να θυσιάσει την απλότητα και την ευκολία χρήσης. Κατ' επέκταση, θεωρείται αποδεκτή η ανταλλαγή ενός μικρού ποσοστού ταχύτητας για ένα σημαντικά πιο απλό στη χρήση μοντέλο. Επιπλέον, μέσω του PyTorch επιτρέπεται στο χρήστη να ελέγξει μόνος του την εκτέλεση του κώδικά του, αφήνοντας τον έτσι να βρει βελτιώσεις στην απόδοση ανεξάρτητα από αυτές που παρέχει το ίδιο αυτόματα.

- Το χειρότερο είναι καλύτερο: Το PyTorch κρατάει την εσωτερική του υλοποίηση απλή και ελαφρώς ελλιπή, με το σκεπτικό ότι θα είναι πιο εύκολο να επεκταθεί και να προσαρμοστεί στις εξελίξεις του τομέα της τεχνητής νοημοσύνης σε σχέση με άλλα πιο διεξοδικά αλλά και πιο σύνθετα και δυσκολότερο να διατηρηθούν συστήματα.

Σχεδίαση με γνώμονα την ευχρηστία

Το PyTorch, για να υποστηρίξει την αυξημένη πολυπλοκότητα που επιφέρει το πεδίο της μηχανικής μάθησης, παραβλέπει τα πιθανά οφέλη μίας μοντελοποίησης βασισμένης σε γράφους για να διατηρήσει το προστακτικό (imperative) προγραμματιστικό μοντέλο της Python. Σε αυτή τη σχεδίαση για δημιουργία μοντέλων ήταν πρωτοπόρες οι βιβλιοθήκες Chainer και Dynet[35]. Το PyTorch επεκτείνει αυτή τη σχεδίαση σε όλες τις πτυχές που περιέχει μία διαδικασία βαθιάς μηχανικής μάθησης. Ο ορισμός επιπέδων νευρωνικού (layers), η σύνθεση μοντέλων, η φόρτωση δεδομένων, η εκτέλεση βελτιστοποιητών και η παραλληλοποίηση της διεργασίας εκπαίδευσης του μοντέλου εκφράζονται όλα χρησιμοποιώντας τις οικείες έννοιες που έχουν αναπτυχθεί για προγραμματισμό γενικού σκοπού.

3.2.1 Διαλειτουργικότητα και επεκτασιμότητα

Το PyTorch επιτρέπει την αμφίδρομη ανταλλαγή δεδομένων με εξωτερικές βιβλιοθήκες, όπως για παράδειγμα τη μετατροπή PyTorch tensors σε NumPy[36] πίνακες και αντίστροφα. Οι μετατροπές αυτές γίνονται χωρίς αντιγραφή δεδομένων, αλλά τα αντικείμενα και από τις δύο μεριές περιγράφουν μόνο πως να ερμηνεύσουν μία περιοχή μνήμης την οποία μοιράζονται. Έτσι αυτές οι λειτουργίες είναι εξαιρετικά φθηνές και παίρνουν σταθερό χρόνο ανεξαρτήτως του μεγέθους των πινάκων. Ακόμα, πολλά από τα σημαντικά συστήματα του PyTorch έχουν σχεδιαστεί ειδικά για να είναι επεκτάσιμα. Για παράδειγμα, το σύστημα αυτόματης διαφόρισης επιτρέπει στους χρήστες να προσθέσουν υποστήριξη για προσαρμοσμένες διαφορίσιμες συναρτήσεις. Το πιο σημαντικό όμως είναι ότι οι χρήστες είναι ελεύθεροι να αντικαταστήσουν οποιαδήποτε συνιστώσα του PyTorch που δεν ταιριάζει στις ανάγκες τους, καθώς είναι όλα σχεδιασμένα ώστε να είναι πλήρως ανταλλάξιμα.

3.2.2 Αυτόματη διαφόριση

Καθότι η βασισμένη στην κλίση βελτιστοποίηση είναι ζωτικής σημασίας στη βαθιά μηχανική μάθηση, το PyTorch διαθέτει μηχανισμό αυτόματης διαφόρισης. Το PyTorch χρησιμοποιεί την προσέγγιση υπερφόρτωσης τελεστών, η οποία χτίζει μία αναπαράσταση της υπολογισμένης συνάρτησης κάθε φορά που αυτή εκτελείται. Στην τωρινή του υλοποίηση, το PyTorch εκτελεί αυτόματη διαφόριση σε αντίστροφη λειτουργία (reverse mode), η οποία υπολογίζει την κλίση μίας βαθμωτής εξόδου ως προς μία πολυδιάστατη είσοδο. Η διαφόριση συναρτήσεων με περισσότερες εξόδους από εισόδους εκτελείται πιο αποδοτικά χρησιμοποιώντας αυτόματη διαφόριση σε λειτουργία προώθησης (forward mode, υποστηρίζεται επίσης από το PyTorch), αλλά αυτή η περίπτωση δε συναντάται συχνά σε εφαρμογές μηχανικής μάθησης. Ένα ακόμα χαρακτηριστικό αυτού του συστήματος είναι ότι μπορεί να διαφοροποιήσει κώδικα που εκτελεί μεταλλάξεις σε tensors, το οποίο είναι ένα από τα βασικά δομικά

στοιχεία του προστακτικού προγραμματισμού.

3.2.3 Υλοποίηση

C++ core

Το μεγαλύτερο μέρος του πυρήνα (core) του PyTorch είναι γραμμένο σε C++ για να πετύχει υψηλή απόδοση. Αυτή η core βιβλιοθήκη, ονομαζόμενη libtorch, υλοποιεί τη δομή δεδομένων tensor, τους CPU και GPU τελεστές, και βασικές λειτουργίες παραλληλίας. Επίσης παρέχει το σύστημα αυτόματης διαφόρισης, συμπεριλαμβανομένων των τύπων κλίσης για τις περισσότερες ενσωματωμένες συναρτήσεις. Αυτό διασφαλίζει ότι ο υπολογισμός των παραγώγων των συναρτήσεων που αποτελείται από core PyTorch τελεστές εκτελείται εξολοκλήρου σε multithreaded περιβάλλον.

Ξεχωριστές ροές ελέγχου και δεδομένων

Το PyTorch διατηρεί έναν αυστηρό διαχωρισμό μεταξύ της ροής ελέγχων και δεδομένων του. Την ανάλυση της ροής ελέγχου τη διαχειρίζεται η Python και ο βελτιστοποιημένος C++ κώδικας που εκτελείται στη CPU του μηχανήματος και έχει ως αποτέλεσμα μία γραμμική αλληλουχία από κλήσεις τελεστών στη συσκευή (CPU ή GPU).

Το PyTorch έχει σχεδιαστεί να εκτελεί τελεστές ασύγχρονα πάνω σε GPU, εκμεταλλευόμενο τον μηχανισμό stream του CUDA για να βάλει στην ουρά του FIFO GPU hardware κλήσεις από CUDA kernels.

3.2.4 Κατανεμημένη εκτέλεση

Στο PyTorch, όλα τα modules που βοηθάνε στην κατανεμημένη εκτέλεση προγραμμάτων βρίσκονται στο πακέτο torch.distributed[37]. Τα χαρακτηριστικά που διαθέτει το πακέτο αυτό μπορούν να κατηγοριοποιηθούν σε τρεις βασικές συνιστώσες:

- Η κατανεμημένη εκπαίδευση παραλληλοποίησης δεδομένων (Distributed Data-Parallel Training, DDP)[38] είναι ένα ευρέως διαδεδομένο πρότυπο εκπαίδευσης με ένα πρόγραμμα και πολλαπλά δεδομένα. Κατά την DDP, το μοντέλο αντιγράφεται σε κάθε διεργασία και σε κάθε αντίγραφο του μοντέλου τροφοδοτείται ένα διαφορετικό σύνολο δειγμάτων των δεδομένων εισόδου. Η DDP φροντίζει για την επικοινωνία των κλίσεων που υπολογίζουν τα διαφορετικά αντίγραφα του μοντέλου ώστε να τα κρατήσει συγχρονισμένα. Η επικοινωνία αυτή μεταξύ των αντιγράφων συμβαίνει παράλληλα με τον υπολογισμό νέων κλίσεων ώστε να επιταχυνθεί η διαδικασία εκπαίδευσης. Για την επικοινωνία μεταξύ των διεργασιών υποστηρίζονται τα πρωτόκολλα mpi,

gloo, και nccl. Για τα πειράματά μας επιλέχθηκε το gloo, το οποίο αποτελεί και προεπιλογή για CPU cluster.

- Η κατανεμημένη εκπαίδευση βασιζόμενη σε RPC (κλήση απομακρυσμένης διαδικασίας, remote procedure call) έχει αναπτυχθεί για να υποστηρίζει γενικές δομές εκπαίδευσης που δεν μπορούν εφαρμοστούν στην εκπαίδευση παραλληλοποίησης δεδομένων, όπως παραλληλισμό κατανεμημένης σωλήνωσης, πρότυπο parameter server, και συνδυασμούς της DDP με άλλα πρότυπα εκπαίδευσης. Βοηθάει στη διαχείριση του κύκλου ζωής απομακρυσμένων αντικειμένων και στην επέκταση της μηχανής αυτόματης διαφόρισης σε περιβάλλοντα με πολλαπλά μηχανήματα.

- Η βιβλιοθήκη συλλογικής επικοινωνίας (Collective Communication, c10d) είναι μία βιβλιοθήκη χαμηλού επιπέδου που υποστηρίζει την αποστολή τανυστών μεταξύ διεργασιών σε ομάδες. Προσφέρει APIs τόσο για συλλογική επικοινωνία όσο και για επικοινωνία κόμβο-σε-κόμβο (peer-to-peer, P2P). Η DDP και η RPC αρχιτεκτονικές που αναφέρθηκαν παραπάνω έχουν χτιστεί με βάση την c10d, με την πρώτη να χρησιμοποιεί συλλογικές επικοινωνίες και τη δεύτερη επικοινωνίες peer-to-peer.

Στην παρούσα εργασία χρησιμοποιήθηκε η βιβλιοθήκη DDP του PyTorch για την σύγχρονη κατανεμημένη εκπαίδευση δικτύων, γι' αυτό και θα αναλυθεί πιο εκτενώς ο εσωτερικός της σχεδιασμός.

DDP λεπτομερώς

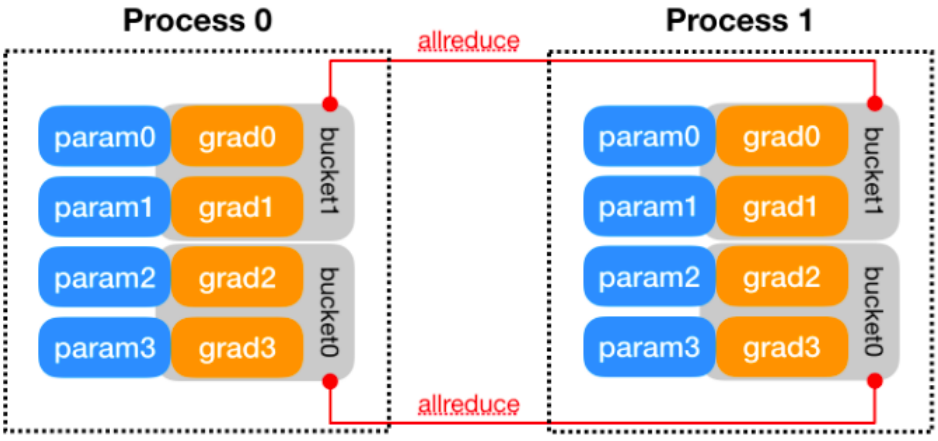
Κατά την *αρχικοποίηση*, ο worker με index 0 στέλνει σε όλους τους άλλους workers (broadcast) τις παραμέτρους του μοντέλου (state), ώστε να ξεκινήσουν όλοι οι workers από την ίδια κατάσταση. Έπειτα κάθε διεργασία DDP (μία ανά worker) δημιουργεί έναν τοπικό Συσσωρευτή (Reducer), ο οποίος θα είναι υπεύθυνος στη συνέχεια για το συγχρονισμό των gradients κατά το backward pass. Ο Reducer οργανώνει τις κλίσεις των παραμέτρων σε κουβάδες και εκτελεί υπολογισμούς σε έναν κουβά τη φορά, με σκοπό να βελτιώσει την αποτελεσματικότητα της επικοινωνίας. Το DDP περιμένει οι κλίσεις των παραμέτρων του δικτύου που βρίσκονται πιο κοντά στο επίπεδο εξόδου να γίνουν διαθέσιμες πρώτες, και έτσι τοποθετεί τις παραμέτρους σε κουβάδες ξεκινώντας από το τέλος του δικτύου και πηγαίνοντας αντίστροφα. Ένα παράδειγμα αυτού φαίνεται στην εικόνα 3.3. Σε αυτό το παράδειγμα, οι κλίσεις των δύο πρώτων παραμέτρων (grad0, grad1) βρίσκονται στον δεύτερο κουβά (bucket1), ενώ οι κλίσεις των δύο επόμενων παραμέτρων (grad2, grad3) βρίσκονται στον πρώτο κουβά (bucket0). Πέρα από το κουβάδιασμα, ο Reducer καταχωρεί έναν γάντζο αυτο-διαφόρισης ανά παράμετρο, που θα ενεργοποιηθεί κατά το ανάστροφο πέρασμα όταν η κλίση έχει υπολογιστεί.

Κατά την *εμπρόσθια τροφοδότηση*, το DDP παίρνει την είσοδο και την περνάει στο τοπικό μοντέλο. Σε αυτή τη φάση είναι διαθέσιμη μία λειτουργία που επιτρέπει στο DDP να βρει ποιες παράμετροι εμπλέκονται στην ανάστροφη τροφοδότηση μέσω

ανάλυσης της εξόδου του τοπικού μοντέλου. Αυτή η λειτουργία θα αποτρέψει αργότερα (κατά την ανάστροφη τροφοδότηση) το DDP από το να περιμένει για κλίσεις παραμέτρων που δεν συμμετέχουν στον υπολογισμό.

Κατά την *ανάστροφη τροφοδότηση*, το DDP χρησιμοποιεί τους γάντζους αυτο-διαφόρισης που είχε καταχωρήσει κατά τη φάση της αρχικοποίησης ώστε να συγχρονίσει τον υπολογισμό των κλίσεων. Όταν μία κλίση γίνει διαθέσιμη, ο γάντζος που έχει καταχωρηθεί στην παράμετρο αυτή θα πυροδοτηθεί και στη συνέχεια το DDP θα σημειώσει την κλίση της παραμέτρου ως έτοιμη για συσσώρευση. Όταν όλες οι κλίσεις ενός κουβά είναι έτοιμες, ο Reducer ξεκινά ένα ασύγχρονο all-reduce σε αυτόν τον κουβά ώστε να υπολογίσει τη μέση κλίση όλων των διεργασιών του cluster. Όταν όλοι οι κουβάδες είναι έτοιμοι, ο Reducer θα περιμένει για όλα τα all-reduce να ολοκληρωθούν. Στη συνέχεια, η μέση υπολογισμένη κλίση για κάθε παράμετρο τροφοδοτείται ως η τελική κλίση της παραμέτρου σε κάθε τοπικό μοντέλο. Επομένως, μετά την ανάστροφη τροφοδότηση η κλίση για αντίστοιχες παραμέτρους μεταξύ διαφορετικών διεργασιών (workers) θα πρέπει να είναι η ίδια.

Το DDP δε λαμβάνει μέρος στη *φάση της βελτιστοποίησης*. Η βελτιστοποίηση συμβαίνει τοπικά, αλλά όλα τα μοντέλα είναι σε συγχρονισμό μεταξύ τους και περιέχουν τελικά τις ίδιες παραμέτρους, αφού ξεκινούν αρχικά από την ίδια κατάσταση και δέχονται κάθε φορά τις ίδιες μέσες υπολογισμένες κλίσεις.



Εικόνα 3.3: Παράδειγμα κουβαδιάσματος κλίσεων στο DDP

3.3 MXNet

Στο πεδίο της μηχανικής μάθησης υπάρχουν δύο κύρια προγραμματιστικά μοντέλα. Ο δηλωτικός προγραμματισμός, που θέτει ξεκάθαρα όρια στον υπολογιστικό γράφο (βλ. TensorFlow version 1), έχει την προοπτική για μεγαλύτερη βελτιστοποίηση. Από

την άλλη ο προστακτικός προγραμματισμός (βλ. PyTorch) προσφέρει μεγαλύτερη ευελιξία. Το MXNet δημιουργήθηκε με την προοπτική να αναμίξει τα πλεονεκτήματα και από τα δύο αυτά προγραμματιστικά μοντέλα, εξού και το όνομα (MXNet or "mix-net"). Το backend του MXNet έχει γραφεί σε C++ και διέθετε εξαρχής διεπαφές με το χρήστη σε αρκετές γλώσσες προγραμματισμού (C++, Python, R, Go και Julia).

3.3.1 Προγραμματιστική διεπαφή

Symbol: Δηλωτικές συμβολικές εκφράσεις

Το MXNet χρησιμοποιεί συμβολικές εκφράσεις πολλαπλών εξόδων (Symbol) για τον ορισμό του υπολογιστικού γράφου. Τα Symbols συντίθενται από τελεστές, όπως απλές λειτουργίες πινάκων (π.χ. πρόσθεση) ή πολύπλοκα επίπεδα νευρωνικών δικτύων (π.χ. συνελικτικό επίπεδο, convolutional layer). Ένας τελεστής μπορεί να δεχτεί πολλές μεταβλητές εισόδου, να παραγάγει πάνω από μία μεταβλητές εξόδου και να διατηρήσει εσωτερική κατάσταση (state). Μία μεταβλητή μπορεί να είναι είτε ελεύθερη, η οποία μπορεί να δεθεί με μία τιμή αργότερα, ή η έξοδος ενός άλλου symbol. Για την αποτίμηση ενός symbol χρειάζεται να δεθούν οι ελεύθερες μεταβλητές με δεδομένα και να δηλωθούν οι απαιτούμενες έξοδοι. Πέρα από την αποτίμηση, ένα symbol υποστηρίζει επίσης αυτόματη διαφόριση.

NDArray: Προστακτικός υπολογισμός τανυστών

Το MXNet προσφέρει τη δομή δεδομένων NDArray με προστακτικό υπολογισμό τανυστών ώστε να καλύψει το κενό μεταξύ των δηλωτικών συμβολικών εκφράσεων και της frontend γλώσσας. Τα NDArray δουλεύουν άψογα σε συνεργασία με τις εκτελέσεις που έχουν οριστεί με Symbol, και είναι έτσι δυνατόν να αναμίξει κανείς τον προστακτικό υπολογισμό τανυστών του πρώτου με το δεύτερο. Για παράδειγμα, μπορεί κανείς να χρησιμοποιήσει το Symbol για τον ορισμό του νευρωνικού δικτύου και στη συνέχει να ανανεώσει τα βάρη του δικτύου χρησιμοποιώντας το προστακτικό API του NDArray. Για να γίνει ο παραπάνω υπολογισμός όσο αποδοτικός θα ήταν χρησιμοποιώντας μόνο το Symbol API, το MXNet χρησιμοποιεί οκνηρή αποτίμηση (lazy evaluation) του NDArray και το backend μπορεί να επιλύσει σωστά την εξάρτηση στα δεδομένα που υπάρχει μεταξύ τους.

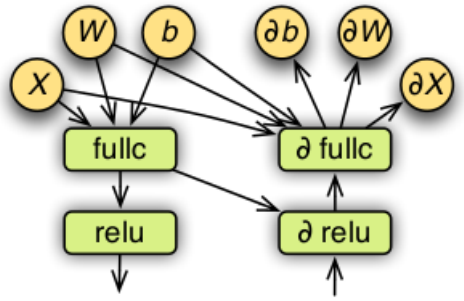
KVStore: Συγχρονισμός δεδομένων μεταξύ συσκευών

Το KVStore είναι μία κατανεμημένη αποθήκη κλειδιού-τιμής για συγχρονισμό δεδομένων μεταξύ πολλαπλών συσκευών. Υποστηρίζει δύο βασικές λειτουργίες: push (σπρώξε) ένα ζεύγος κλειδιού-τιμής από μία συσκευή στην αποθήκη, και pull (τράβα) την τιμή για ένα συγκεκριμένο κλειδί από την αποθήκη. Βασική χρήση αυτού του module είναι η κατανεμημένη εκτέλεση με παραλληλισμό δεδομένων, όπου κάθε worker τραβάει από το KVStore τα ανανεωμένα βάρη του νευρωνικού και στέλνει πίσω στο KVStore την υπολογισμένη κλίση.

3.3.2 Υλοποίηση

Γράφος υπολογισμού

Μία έκφραση από σύμβολα αναπαριστάται από έναν γράφο προς αποτίμηση. Στην εικόνα 3.4 φαίνεται ένα μέρος του υπολογιστικού γράφου του εμπρόσθιου και του ανάστροφου περάσματος ενός πολυεπίπεδου νευρωνικού δικτύου (multi-layer perceptron, MLP). Πριν την αποτίμηση του γράφου, το MXNet εφαρμόζει ορισμένες μετατροπές στο γράφο ώστε να βελτιστοποιήσει την απόδοση και να διανείμει μνήμη σε μεταβλητές. Για την βελτιστοποίηση του γράφου, αρχικά αφαιρούνται οι κόμβοι που δε χρειάζονται για τον υπολογισμό των καθορισμένων εξόδων και κρατείται ουσιαστικά ένας υπογράφος του αρχικού. Επίσης γίνεται ομαδοποίηση κάποιων τελεστών που μπορούν να αντικατασταθούν από έναν μόνο τελεστή (π.χ. το 'a x b + 1' αντικαθίσταται από μία κλήση BLAS ή GPU, BLAS: Basic Linear Algebra Subprogram[39]). Ακόμα, στο MXNet υπάρχουν υλοποιημένες καλά βελτιστοποιημένες "μεγάλες" λειτουργίες, όπως ένα layer νευρωνικού δικτύου.



Εικόνα 3.4: Υπολογιστικός γράφος εμπρόσθιου και οπίσθιου περάσματος

Μηχανή εξαρτήσεων

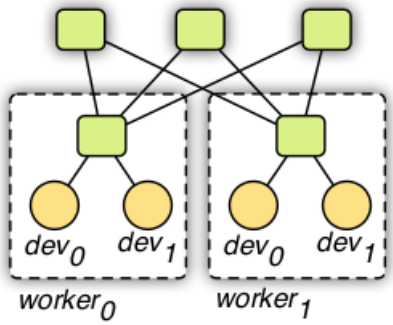
Στο MXNet, κάθε πηγαία μονάδα, συμπεριλαμβανομένων των NDArray, γεννήτρια τυχαίων αριθμών και πρόσκαιρου χώρου (temporal space), εγγράφεται στη μηχανή εξαρτήσεων με μία μοναδική ετικέτα. Κάθε λειτουργία, όπως λειτουργίες πινάκων ή επικοινωνία δεδομένων, γίνεται ύστερα push μες στη μηχανή αυτή με προσδιορισμένες τις ετικέτες των απαιτούμενων πόρων. Η μηχανή εξάρτησης δρομολογεί συνεχώς τις λειτουργίες που έχουν γίνει push για εκτέλεση εφόσον έχουν επιλυθεί οι εξαρτήσεις τους. Καθώς υπάρχουν συνήθως πολλαπλοί υπολογιστικοί πόροι (CPUs, GPUs), η μηχανή χρησιμοποιεί πολλαπλά νήματα για τη δρομολόγηση των λειτουργιών για καλύτερη αξιοποίηση των πόρων και παραλληλισμό. Σε αντίθεση με τις περισσότερες μηχανές ροής δεδομένων, η μηχανή εξαρτήσεων του MXNet καταγράφει τις λειτουργίες μετάλλαξης ως μία υπάρχουσα

πηγαία μονάδα. Αυτό κάνει εφικτή τη δρομολόγηση μεταλλάξεων πινάκων καθώς επίσης και την ευκολότερη επαναχρησιμοποίηση μνήμης των παραμέτρων.

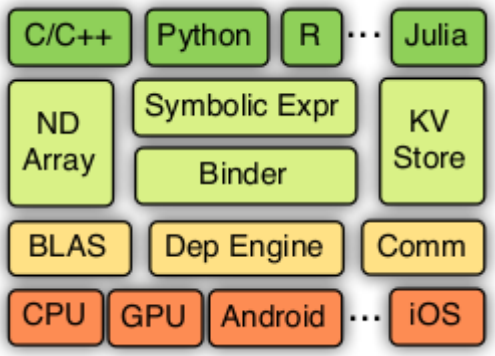
Επικοινωνία δεδομένων

Η υλοποίηση του KVStore βασίζεται σε αρχιτεκτονική parameter-server. Δύο είναι τα πράγματα που το διαφοροποιούν από προηγούμενες δουλειές. Πρώτον, χρησιμοποιείται η μηχανή εξάρτησης για τη δρομολόγηση λειτουργιών του KVStore και για διαχείριση της συνέπειας των δεδομένων (data consistency). Δεύτερον, υιοθετείται μία δομή δύο επιπέδων. Ένας server επιπέδου-1 διαχειρίζεται το συγχρονισμό των δεδομένων μεταξύ των συσκευών μέσα σε ένα μηχάνημα, ενώ ένας server επιπέδου-2 διαχειρίζεται το συγχρονισμό μεταξύ μηχανημάτων. Τα εξερχόμενα δεδομένα από τον server επιπέδου-1 μπορούν να συναθροιστούν, μειώνοντας έτσι τις απαιτήσεις σε εύρος ζώνης. Μία απεικόνιση της αρχιτεκτονικής αυτής φαίνεται στην εικόνα 3.5.

Στην εικόνα 3.6 φαίνεται μία σύνοψη των συνιστωσών που περιέχει το MXNet.



Εικόνα 3.5: Επικοινωνία δεδομένων στο MXNet



Εικόνα 3.6: Σύνοψη συστήματος MXNet

3.3.3 Gluon

Αντίστοιχα με το Keras για το TensorFlow, υπάρχει και το Gluon[40] για το MXNet, ώστε να διευκολύνει ακόμα παραπάνω τη διεπαφή του χρήστη με το σύστημα. Η βιβλιοθήκη Gluon στο MXNet παρέχει ένα καθαρό, ακριβές και απλό API για βαθιά μηχανική μάθηση. Διευκολύνει την προτυποποίηση, το χτίσιμο και την εκπαίδευση βαθιών νευρωνικών δικτύων χωρίς να θυσιάσει ταχύτητα. Τα οφέλη που προσφέρει η χρήση του Gluon συνοψίζονται παρακάτω:

- Απλός, εύκολος στην κατανόηση κώδικας: Το Gluon προσφέρει ένα πλήρες σύνολο ετοιμοπαράδοτων δομικών μπλοκ νευρωνικών δικτύων, συμπεριλαμβανομένων προορισμένων layers, βελτιστοποιητών και αρχικοποιητών.

- Ευέλικτη, προστακτική δομή: Το Gluon δεν απαιτεί να οριστεί αυστηρά το νευρωνικό δίκτυο, αλλά φέρνει τον αλγόριθμο εκπαίδευσης και το μοντέλο πιο κοντά για να παρέχει ευελιξία στην διαδικασία ανάπτυξης.

- Δυναμικοί γράφοι: Το Gluon δίνει τη δυνατότητα στους προγραμματιστές να ορίσουν μοντέλα νευρωνικών δικτύων που είναι δυναμικά, δηλαδή να μπορούν να χτιστούν στη στιγμή, με οποιαδήποτε δομή, και χρησιμοποιώντας οποιαδήποτε ροή ελέγχου της Python.

- Υψηλή απόδοση: Το Gluon παρέχει όλα τα παραπάνω οφέλη χωρίς να έχει κάποια επίπτωση στην ταχύτητα εκπαίδευσης που παρέχει η μηχανή (το MXNet backend) που τρέχει από κάτω.

3.3.4 Κατανεμημένη εκτέλεση

Κατά την κατανεμημένη εκτέλεση, το MXNet διαθέτει τρεις τύπους διεργασιών που επικοινωνούν μεταξύ τους για να πραγματοποιήσουν την εκπαίδευση ενός μοντέλου[41].

- Εργάτης (worker): Ένας κόμβος worker εκτελεί την εκπαίδευση πάνω σε ένα batch δειγμάτων του συνόλου δεδομένων εκπαίδευσης. Προτού επεξεργαστούν το κάθε batch, οι workers τραβάνε τα βάρη του δικτύου από τους servers. Οι workers επίσης στέλνουν τις υπολογισμένες κλίσεις στους servers μετά από κάθε batch.

- Εξυπηρετητής (server): Μπορούν να υπάρξουν πολλαπλοί servers που αποθηκεύουν τις παραμέτρους του μοντέλου και επικοινωνούν με τους workers. Ένας server μπορεί είτε να τοποθετηθεί στην ίδια διεργασία με τον worker είτε ξεχωριστά.

- Δρομολογητής (scheduler): Μπορεί να υπάρξει μόνο ένας scheduler. Ο ρόλος αυτού είναι να στήσει το cluster. Αυτό συμπεριλαμβάνει την αναμονή

για μηνύματα από τον κάθε κόμβο που να δηλώνουν ότι έχει σηκωθεί και σε ποια θύρα ακούει. Ο scheduler τότε ενημερώνει όλες τις διεργασίες σχετικά με όλους τους άλλους κόμβους που υπάρχουν στο cluster, ώστε να μπορούν να επικοινωνήσουν μεταξύ τους.

Η κατανεμημένη εκπαίδευση στο MXNet ενεργοποιείται αυτόματα με την δημιουργία ενός kvstore (όπως αναφέρθηκε παραπάνω) που παίρνει ως όρισμα μία συμβολοσειρά που περιέχει τη λέξη "dist". Διαφορετικοί τύποι κατανεμημένης εκτέλεσης μπορούν να ενεργοποιηθούν με χρήση διαφορετικών τύπων kvstore, οι οποίοι περιγράφονται παρακάτω:

- *dist_ sync*

  Στη σύγχρονη κατανεμημένη εκτέλεση, όλοι οι workers χρησιμοποιούν το ίδιο συγχρονισμένο σύνολο παραμέτρων στην αρχή του κάθε batch. Στο τέλος του κάθε batch ο server περιμένει να λάβει τις υπολογισμένες κλίσεις από όλους τους workers προτού ανανεώσει τις παραμέτρους του μοντέλου. Αυτή η διαδικασία βέβαια προσθέτει ένα κόστος, αφού ο worker θα πρέπει να περιμένει έως ότου οι νέες παράμετροι να είναι έτοιμες και να μπορεί να τις τραβήξει και να συνεχίσει με το επόμενο batch. Σε αυτή τη λειτουργία, εάν υπάρξει σφάλμα σε έναν από τους workers τότε θα διακοπεί η εκπαίδευση όλου του cluster.

- *dist_ async*

  Στην ασύγχρονη κατανεμημένη εκτέλεση, ο server λαμβάνει τις υπολογισμένες κλίσεις από έναν worker και ανανεώνει άμεσα τις παραμέτρους που κρατάει, τις οποίες χρησιμοποιεί μετά για να απαντήσει σε μελλοντικές αιτήσεις για pull παραμέτρων. Αυτό σημαίνει ότι ένας worker που τελειώνει την επεξεργασία του πάνω σε ένα batch μπορεί να τραβήξει τις τρέχουσες παραμέτρους από τον server και να ξεκινήσει το επόμενο batch, ακόμα και αν οι άλλοι workers δεν έχουν προλάβει να τελειώσουν την επεξεργασία τους στο προηγούμενο batch. Αυτό είναι γρηγορότερο από το dist_sync επειδή δεν υπάρχει κόστος για συγχρονισμό, αλλά μπορεί να χρειαστεί περισσότερες εποχές για να συγκλίνει το δίκτυο. Η ανανέωση των παραμέτρων είναι ατομική, που σημαίνει ότι δεν μπορούν να συμβούν δύο ανανεώσεις στην ίδια παράμετρο την ίδια στιγμή. Όμως, η σειρά των ανανεώσεων δεν είναι εγγυημένη.

- *dist_ sync_ device*

  Αυτή η λειτουργία είναι όμοια με την dist_sync με εξαίρεση ότι υπάρχουν πολλαπλές GPUs που χρησιμοποιούνται σε κάθε κόμβο. Σε αυτή τη λειτουργία οι κλίσεις συσσωρεύονται και οι παράμετροι ανανεώνονται πάνω σε GPU, ενώ στην dist_sync αυτό γίνεται στη μνήμη της CPU. Αυτή η λειτουργία είναι γρηγορότερη της dist_sync, καθώς μειώνει την ακριβή επικοινωνία μεταξύ GPU και CPU, αλλά αυξάνει τη χρήση μνήμης στη GPU.

- *dist_ async_ device*

  Σε αυτή τη λειτουργία συμβαίνει το ανάλογο της dist_sync_device αλλά σε ασύγχρονη λειτουργία.

Ανανέωση παραμέτρων

Ο KVStore εξυπηρετητής υποστηρίζει δύο λειτουργίες για την ανανέωση των παραμέτρων του μοντέλου. Κατά την πρώτη, συσσωρεύει τις κλίσεις από όλους τους workers, υπολογίζει τη μέση κλίση και τη χρησιμοποιεί για να ανανεώσει τα βάρη του νευρωνικού. Κατά τη δεύτερη, υπολογίζει και πάλι τη μέση κλίση αλλά δίχως να εφαρμόσει την ανανέωση των παραμέτρων. Σε αυτήν την περίπτωση, ο κάθε worker τραβάει τη μέση υπολογισμένη κλίση και εφαρμόζει την ενημέρωση των παραμέτρων τοπικά.

Επικοινωνία διεργασιών

Για την επικοινωνία μεταξύ διεργασιών που βρίσκονται σε διαφορετικά μηχανήματα σε κοινό δίκτυο παρέχεται ένα launch.py module, το οποίο αποτελεί ένα προσαρμοσμένο RPC εργαλείο και υποστηρίζει τα παρακάτω:

- *ssh*

  Εφόσον τα μηχανήματα μπορούν να επικοινωνήσουν με ssh χωρίς κωδικούς. Αυτή είναι η προεπιλεγμένη λειτουργία.

- *mpi*

  Αν το Open MPI είναι διαθέσιμο.

- *sge*

  Για υποστήριξη Sun Grid Engine.

- *yarn*

  Για υποστήριξη Apache Yarn.

3.4 Μέθοδοι Επικοινωνίας Διεργασιών

Παρακάτω περιγράφονται εν συντομία οι προαναφερθείσες μέθοδοι με τις οποίες επιτυγχάνεται η επικοινωνία μεταξύ των διεργασιών, όταν τα παραπάνω τρία συστήματα λειτουργούν σε κατανεμημένο περιβάλλον.

3.4.1 RPC

Μία κλήση απομακρυσμένης διαδικασίας (remote procedure call - RPC)[42] πραγματοποιείται όταν ένα πρόγραμμα προκαλεί την εκτέλεση μίας διαδικασίας

(υπορουτίνας) σε έναν διαφορετικό χώρο διευθύνσεων. Αυτός ο χώρος συνήθως εντοπίζεται σε ένα άλλο μηχάνημα που βρίσκεται σε κοινό δίκτυο. Η κλήση αυτή σε επίπεδο κώδικα φαίνεται σαν μία κανονική τοπική κλήση διαδικασίας. Αποτελεί μία μορφή αλληλεπίδρασης πελάτη-εξυπηρετητή που υλοποιείται συνήθως με ένα σύστημα ανταλλαγής μηνυμάτων τύπου αιτήσεων-απαντήσεων.

3.4.2 MPI

Η Διεπαφή Μετάδοσης Μηνυμάτων (Message Passing Interface, MPI)[43] είναι ένα πρότυπο μετάδοσης μηνυμάτων που έχει σχεδιαστεί για να λειτουργεί σε παράλληλες αρχιτεκτονικές υπολογιστών. Το πρότυπο αυτό ορίζει τη σύνταξη και τη σημασιολογία ρουτινών βιβλιοθηκών που είναι χρήσιμες σε χρήστες που γράφουν υλοποιήσεις της διεπαφής αυτής.

3.4.3 GLOO

Το Gloo[44] είναι μία βιβλιοθήκη για επικοινωνία μεταξύ διεργασιών που έφτιαξε η ομάδα του Facebook, κυρίως για χρήση στο PyTorch. Περιέχεται ως προεπιλογή στα έτοιμα εκτελέσιμα του PyTorch. Υποστηρίζει όλες τις λειτουργίες σημείου-σε-σημείο, τις συλλογικές λειτουργίες σε CPU και όλες τις συλλογικές λειτουργίες σε GPU. Ωστόσο, στην τελευταία περίπτωση δεν είναι όσο βελτιστοποιημένο όσο το NCCL.

3.4.4 NCCL

Η NCCL[45] βιβλιοθήκη παρέχει μία βελτιστοποιημένη υλοποίηση συλλογικών λειτουργιών για τανυστές CUDA. Αποτελεί την καλύτερη επιλογή για κατανεμημένη εκπαίδευση νευρωνικών σε GPU συσκευές.

3.4.5 SSH

Το SSH (Secure Shell)[46] είναι ένα ασφαλές δικτυακό πρωτόκολλο που επιτρέπει τη μεταφορά δεδομένων μεταξύ δύο υπολογιστών. Το SSH όχι μόνο κρυπτογραφεί τα δεδομένα που ανταλλάσσονται κατά τη συνεδρία, αλλά προσφέρει κι ένα ασφαλές σύστημα αναγνώρισης, καθώς και άλλα χαρακτηριστικά όπως ασφαλή μεταφορά αρχείων.

3.4.6 YARN

Η θεμελιώδης ιδέα του Apache Yarn[47] είναι να διαχωρίσει τις λειτουργίες της διαχείρισης πόρων και της δρομολόγησης εργασιών σε ξεχωριστές διεργασίες. Η ιδέα είναι να υπάρχει ένας ενιαίος Διαχειριστής Πόρων (Resource Manager, RM) και ένα Αφεντικό ανά εφαρμογή (Application Master, AM).

3.4.7 SGE

Το (Sun) Grid Engine[48] χρησιμοποιείται συνήθως σε μία φάρμα υπολογιστών ή σε cluster υπολογισμού υψηλής επίδοσης (high-performance computing, HPC) κι είναι υπεύθυνο για την αποδοχή, τη δρομολόγηση, το διαμοιρασμό και τη διαχείριση της απομακρυσμένης και κατανεμημένης εκτέλεσης ενός μεγάλου αριθμού εργασιών.

Κεφάλαιο 4

Μεθοδολογία

Στην παρούσα διπλωματική εργασία γίνεται ανάλυση και σύγκριση τριών εκ των πιο διάσημων βιβλιοθηκών βαθιάς μηχανικής μάθησης, δηλαδή TensorFlow, PyTorch και MXNet, τόσο σε τοπική εκτέλεση όσο και σε κατανεμημένο περιβάλλον. Κύρια μετρική της σύγκρισης αυτής αποτελεί ο χρόνος εκτέλεσης. Σκοπός είναι μέσα από την ανάλυση αυτή να εντοπιστούν σημεία στα οποία η υλοποίηση των τριών συστημάτων διαφέρει, και να εξηγηθεί ποια υλοποίηση φαίνεται να υπερτερεί σε κάθε περίπτωση για τη δεδομένη αρχιτεκτονική.

Όλα τα πειράματα αφορούν κατηγοριοποίηση εικόνων, οπότε και επιλέχθηκαν συνελικτικά νευρωνικά δίκτυα ως μοντέλα. Τα σύνολα δεδομένων πάνω στα οποία αυτά θα εκπαιδευτούν αποτελούνται από εικόνες μαζί με τις αντίστοιχες ετικέτες τους (που ορίζουν σε ποια κλάση/κατηγορία ανήκει η κάθε εικόνα).

4.1 Πειραματική Διάταξη

4.1.1 Πόροι

Για την διεξαγωγή των πειραμάτων χρησιμοποιήθηκαν τρία απομακρυσμένα μηχανήματα που βρίσκονται σε κοινό δίκτυο στο υπολογιστικό νέφος. Τα χαρακτηριστικά τους φαίνονται στον πίνακα 4.1.

4.1.2 Βιβλιοθήκες

Όπως προαναφέρθηκε τα τρία συστήματα μηχανικής μάθησης που χρησιμοποιήθηκαν είναι τα TensorFlow, PyTorch και MXNet. Και για τα τρία αυτά συστήματα η client γλώσσα προγραμματισμού ήταν η Python. Ακόμα, έγινε χρήση των βιβλιοθηκών Keras και Gluon για χρήση προγραμματιστικών διεπαφών υψηλού επιπέδου πάνω από τα TensorFlow και MXNet αντίστοιχα.

Οι εκδόσεις που χρησιμοποιήθηκαν για την κάθε τεχνολογία φαίνονται στον

Χαρακτηριστικό	**Τιμή**
Μοντέλο επεξεργαστή	Intel Core Processor (Skylake) @ 2.2GHz
Πλήθος επεξεργαστών	4
Πυρήνες ανά επεξεργαστή	1
Νήματα ανά πυρήνα	1
Μνήμη RAM	16GB

Πίνακας 4.1: Χαρακτηριστικά μηχανημάτων

Τεχνολογία	**Έκδοση**
Python	3.6.9
TensorFlow	2.3.1
PyTorch	1.5.1
MXNet	1.7.0

Πίνακας 4.2: Εκδόσεις ανά τεχνολογία

πίνακα 4.2.

Οι λειτουργίες συγχρονισμού που υποστηρίζονται από τα τρία εν λόγω συστήματα παρατίθενται στον πίνακα 4.3. Σημειώνεται ότι ενώ το PyTorch ξεκίνησε να υποστηρίζει την ασύγχρονη λειτουργία κατά τη διάρκεια εκπόνησης της παρούσας εργασίας, η υποστήριξη αυτή ήταν σε πολύ πρώιμο-πειραματικό στάδιο και χωρίς την παροχή κάποιου υψηλού επιπέδου API. Η κατανεμημένη αρχιτεκτονική με την οποία έχουν υλοποιηθεί οι λειτουργίες συγχρονισμού σε κάθε σύστημα φαίνονται στον πίνακα 4.4.

Για την υλοποίηση των τελεστών μηχανικής μάθησης το TensorFlow χρησιμοποιεί την βιβλιοθήκη Eigen[49], η οποία παρέχει λειτουργικότητα για διάφορες μαθηματικές πράξεις, ενώ τα PyTorch και MXNet κάνουν χρήση της βιβλιοθήκης oneDNN (oneAPI Deep Neural Network Library, γνωστή και ως πρώην MKL-DNN)[50], η οποία παρέχει τελεστές ειδικά για βαθιά νευρωνική μάθηση και παρέχει βελτιστοποιήσεις ειδικά για επεξεργαστές τύπου Intel. Οι βιβλιοθήκες που χρησιμοποιούνται για το backend ανά σύστημα συνοψίζονται στον πίνακα 4.5.

Σύστημα	**Σύγχρονη λειτουργία**	**Ασύγχρονη λειτουργία**
TensorFlow	✓	✓
PyTorch	✓	
MXNet	✓	✓

Πίνακας 4.3: Υποστήριξη λειτουργιών συγχρονισμού ανά σύστημα

Σύστημα	Σύγχρονη αρχιτεκτονική	Ασύγχρονη αρχιτεκτονική
TensorFlow	All-Reduce	Parameter Server
PyTorch	All-Reduce	-
MXNet	Parameter Server	Parameter Server

Πίνακας 4.4: Κατανεμημένη αρχιτεκτονική ανά λειτουργία συγχρονισμού ανά σύστημα

Σύστημα	Backend βιβλιοθήκη
TensorFlow	Eigen
PyTorch	oneDNN
MXNet	oneDNN

Πίνακας 4.5: Backend βιβλιοθήκη ανά σύστημα

4.1.3 Μοντέλα

Όλα τα μοντέλα που εξετάστηκαν αποτελούν συνελικτικά νευρωνικά δίκτυα. Τα LeNet-5 και AlexNet έχουν απλή δομή, ενώ τα ResNet-18 και ResNet-50 περιέχουν πιο σύνθετες μονάδες. Περιγραφή των νευρωνικών αυτών μπορεί να βρεθεί στην υπό-ενότητα 2.2.3.

4.1.4 Σύνολα Δεδομένων

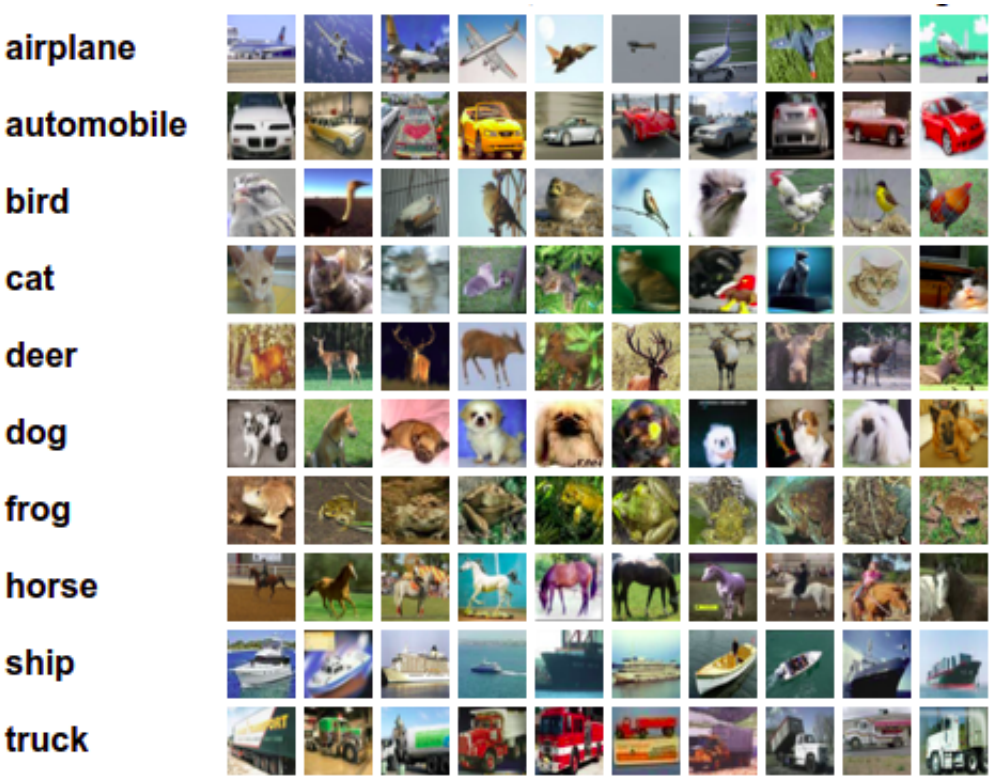
Τα σύνολα δεδομένων πάνω στα οποία εκπαιδεύτηκαν τα νευρωνικά κατά το πειραματικό στάδιο είναι τα CIFAR-10 και CIFAR-100, και περιέχουν αμφότερα εικόνες με ετικέτες (labeled images).

CIFAR-10

Το σύνολο δεδομένων CIFAR-10[51] αποτελείται από 60.000 εικόνες διαστάσεων 32x32, οι οποίες είναι χωρισμένες σε 10 ισομεγέθεις κατηγορίες[52]. Οι κατηγορίες αυτές είναι αμοιβαία αποκλειόμενες και φαίνονται στην εικόνα 4.1 μαζί με κάποια δείγματα ανά κατηγορία.

CIFAR-100

Το σύνολο δεδομένων CIFAR-100 μοιάζει με το CIFAR-10, με τη διαφορά ότι οι εικόνες έχουν διαμοιραστεί σε 100 κατηγορίες. Οι 100 αυτές κατηγορίες-κλάσεις έχουν ομαδοποιηθεί σε 20 υπέρ-κλάσεις, οι οποίες όμως δε λαμβάνονται υπόψιν κατά τη διαδικασία μάθησης. Οι κλάσεις και οι υπέρ-κλάσεις που περιέχονται στο σύνολο αυτό φαίνονται στον πίνακα 4.6.



Εικόνα 4.1: Κατηγορίες εικόνων του CIFAR10

Υπέρ-κλάση	**Κλάσεις**
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Πίνακας 4.6: Κλάσεις ανά υπέρ-κλάση στο CIFAR-100

Τα χαρακτηριστικά των CIFAR-10 και CIFAR-100 συνοψίζονται επίσης στον πίνακα 4.7.

4.2 Πειράματα

Τα πειράματα πραγματοποιήθηκαν σε τρεις βασικές φάσεις και μία προκαταρκτική. Κατά την προκαταρκτική φάση επικυρώθηκε ότι τα νευρωνικά δίκτυα μεταξύ των διαφορετικών συστημάτων παράγουν το ίδιο αποτέλεσμα κατά την εκπαίδευσή τους, ώστε οι μετρήσεις στη συνέχεια να έχουν νόημα και να είναι συγκρίσιμες.

Χαρακτηριστικό	**CIFAR-10**	**CIFAR-100**
Πλήθος εικόνων	60,000	60,000
Διαστάσεις εικόνας	32x32	32x32
Κανάλια εικόνας	3	3
Πλήθος κλάσεων	10	100
Εικόνες ανά κλάση	6000	600

Πίνακας 4.7: Χαρακτηριστικά των συνόλων δεδομένων CIFAR-10 και CIFAR-100

Οι επόμενες τρεις βασικές φάσεις αποτελούν τα πραγματικά πειράματα από τα οποία προέκυψαν οι μετρήσεις προς σύγκριση. Οι μετρήσεις αυτές περιέχουν τον συνολικό χρόνο εκπαίδευσης σε κατανεμημένο περιβάλλον, το χρόνο που ξοδεύεται σε επικοινωνία μεταξύ των κόμβων του cluster, το χρόνο που ο κάθε κόμβος ξοδεύει στην εκπαίδευση του μοντέλου τοπικά, το χρόνο που αντιστοιχεί στο φόρτωμα των δεδομένων στη μνήμη, το χρόνο που παίρνει το forward pass, το χρόνο που παίρνει το backward pass και τέλος το χρόνο που παίρνουν οι επιμέρους τελεστές για το κάθε σύστημα. Οι φάσεις αυτές περιγράφονται αναλυτικότερα στις επόμενες υπό-ενότητες.

4.2.1 Φάση 0 - Επικύρωση ισοδυναμίας νευρωνικών

Η φάση αυτή είναι προκαταρκτική και δεν αποτελεί μέρος των πειραμάτων. Στη φάση αυτή, ουσιαστικά, στήθηκε ένα σύστημα σύγκρισης νευρωνικών δικτύων (Συγκριτής) που έχουν υλοποιηθεί από διαφορετικές βιβλιοθήκες.

Ο Συγκριτής αρχικά επαληθεύει ότι τα βάρη των τριών δικτύων (π.χ. AlexNet σε TensorFlow, AlexNet σε PyTorch και AlexNet σε MXNet) έχουν ίσες διαστάσεις (shape) και τιμές, άρα ότι έχουν αρχικοποιηθεί με τον ίδιο τρόπο. Στη συνέχεια ο Συγκριτής επαληθεύει επίσης ότι για ίδια είσοδο τα τρία δίκτυα θα παράγουν την ίδια έξοδο (τιμή συνάρτησης κόστους). Η τελευταία επαλήθευση που κάνει ο Συγκριτής είναι να ελέγξει ότι μετά από εκπαίδευση για ορισμένα βήματα/εποχές τα βάρη των τριών δικτύων παραμένουν κοντά μεταξύ τους. Κάθε τιμή προς σύγκριση που παράγεται από τα διαφορετικά συστήματα (TensorFlow, PyTorch, MXNet) μετατρέπεται πριν την τελική σύγκριση σε δομή δεδομένων της βιβλιοθήκης NumPy, ώστε όλες οι τιμές/διανύσματα να έχουν κοινό τύπο.

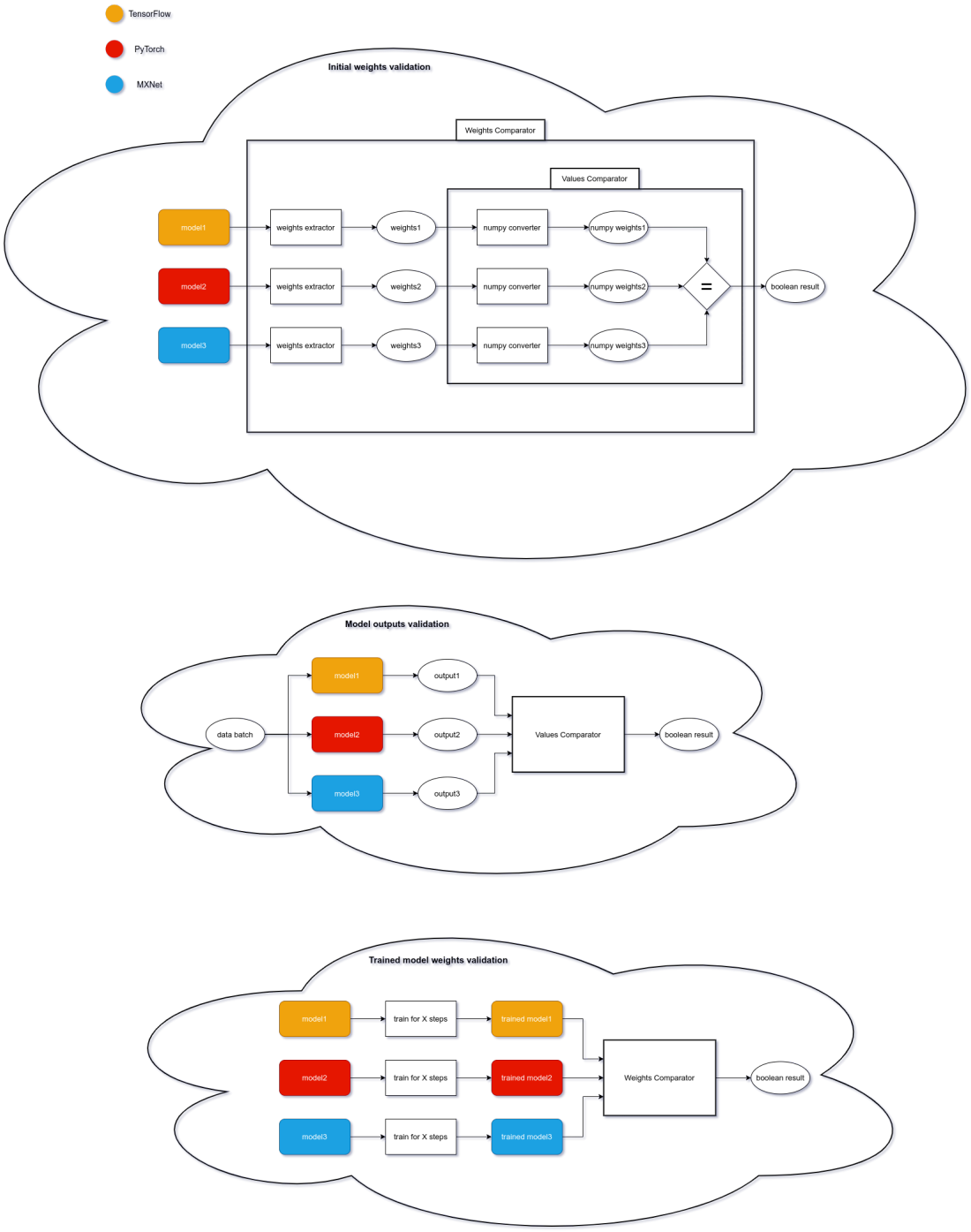
Το σύστημα αυτό (Συγκριτής), παρότι δεν αποτελεί μέρος των πειραμάτων, παίζει σημαντικό ρόλο στην επικύρωση της ισοδυναμίας των υλοποιήσεων των νευρωνικών μεταξύ διαφορετικών βιβλιοθηκών, άρα και στην εγκυρότητα των αποτελεσμάτων. Μία σχηματική απεικόνιση των τριών επαληθεύσεων που εκτελεί φαίνεται στην εικόνα 4.2.

4.2.2 Φάση 1 - Κατανεμημένη εκπαίδευση

Η φάση αυτή αποτελεί το βασικό πειραματικό στάδιο, καθώς τα πειράματα για τις επόμενες δύο φάσεις πραγματοποιήθηκαν για περαιτέρω ανάλυση αυτής. Σε αυτή τη φάση έγινε εκπαίδευση συνελικτικών νευρωνικών δικτύων πάνω στα διαθέσιμα σύνολα δεδομένων με κατανεμημένο τρόπο. Οι συνδυασμοί δικτύων και συνόλων δεδομένων που επιλέχθηκαν φαίνονται στον πίνακα 4.8.

Για κάθε τέτοιο συνδυασμό εκτελέστηκε κατανεμημένη εκπαίδευση σε κάθε σύστημα για όλες τις λειτουργίες συγχρονισμού που φαίνονται στον πίνακα 4.3.

Οι παράμετροι που τέθηκαν για την εκπαίδευση των νευρωνικών σε κατανεμημένο περιβάλλον συνοψίζονται στον πίνακα 4.9. Όπως παρατηρείται στον πίνακα αυτόν, η διάσταση της εικόνας εισόδου είναι διαφορετική μόνο για την περίπτωση που το νευρωνικό είναι το AlexNet. Η επιλογή αυτή ήταν υποχρεωτική



Εικόνα 4.2: Αρχιτεκτονική του Συγκριτή

Νευρωνικό δίκτυο	Σύνολο δεδομένων
Lenet-5	CIFAR-10
AlexNet	CIFAR-10
ResNet-18	CIFAR-10
ResNet-18	CIFAR-100
ResNet-50	CIFAR-10
ResNet-50	CIFAR-100

Πίνακας 4.8: Συνδυασμοί νευρωνικών δικτύων με σύνολα δεδομένων για τα πειράματα

Παράμετρος	Τιμή
Πλήθος δειγμάτων	50,000
Καθολικό μέγεθος batch	126
Τοπικό μέγεθος batch	42
Βήματα ανά εποχή	396
Πλήθος εποχών	10
Διαστάσεις εικόνας	32x32 (64x64 μόνο για AlexNet)

Πίνακας 4.9: Παράμετροι εκπαίδευσης

καθώς, λόγω των πολλών υπό-δειγματοληψιών που λαμβάνουν μέρος στα επίπεδα του AlexNet, οι διαστάσεις εισόδου 32x32 δεν είναι αρκετές για να περάσει η πληροφορία από όλο το δίκτυο. Αυτή η επιλογή βοήθησε βέβαια και στην αύξηση της ποικιλίας παραμέτρων των πειραμάτων.

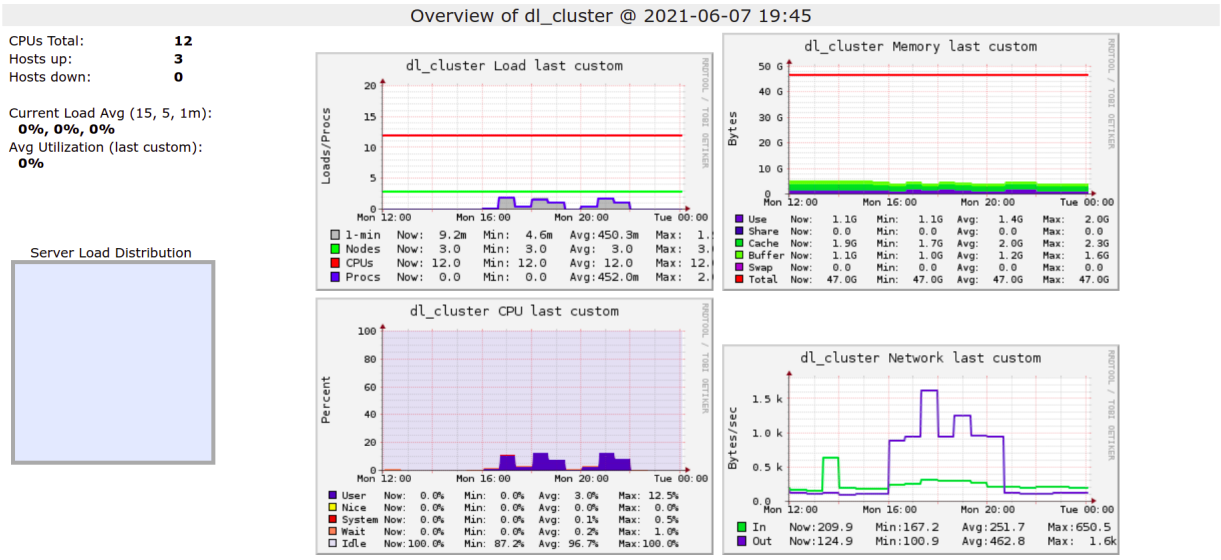
Από αυτή τη σειρά πειραμάτων κρατήθηκε ο συνολικός χρόνος εκπαίδευσης του εκάστοτε νευρωνικού για κατανεμημένη εκτέλεση όλου του cluster.

Κατά την κατανεμημένη εκτέλεση των πειραμάτων, έγινε επίσης καταγραφή της κατανάλωσης των πόρων του cluster, τόσο συνολικά όσο και ανά κόμβο ξεχωριστά. Οι μετρικές που καταγράφηκαν αφορούν την κατανάλωση σε CPU, μνήμη και δίκτυο. Το εργαλείο που χρησιμοποιήθηκε για την καταγραφή αυτή είναι το Ganglia[53], το οποίο αποτελεί ένα κλιμακούμενο σύστημα παρακολούθησης υπολογιστικών πόρων και δικτύων. Ένα παράδειγμα της διεπαφής χρήστη που παρέχει το Ganglia φαίνεται στην εικόνα 4.3.

4.2.3 Φάση 2 - Τοπική εκπαίδευση

Κατά τη δεύτερη φάση πειραμάτων, πραγματοποιήθηκαν όλα τα πειράματα που αναφέρθηκαν και στην προηγούμενη φάση, με τη διαφορά ότι η εκπαίδευση των δικτύων δεν έγινε κατανεμημένα, αλλά τοπικά σε ένα μόνο μηχάνημα του cluster.

Για να προσομοιώσουμε το φόρτο εργασίας που έχει ο κόμβος αυτός κατά την



Εικόνα 4.3: Ganglia διεπαφή χρήστη

κατανεμημένη εκτέλεση, χρησιμοποιήθηκε το τοπικό μέγεθος batch, όπως φαίνεται στον πίνακα 4.9. Ακόμα, το μέγεθος του συνόλου δεδομένων που δόθηκε για την εκπαίδευση του νευρωνικού ισούται με το ένα τρίτο (1/3) του συνολικού, όσο δηλαδή θα χρειαζόταν να επεξεργαστεί ο συγκεκριμένος κόμβος και κατά την κατανεμημένη εκτέλεση.

Από αυτή τη σειρά πειραμάτων μετρήθηκε ο χρόνος εκπαίδευσης τοπικά για έναν κόμβο, ενώ καταγράφηκαν επίσης και ξεχωριστά ο χρόνος φόρτωσης του συνόλου δεδομένων, ο χρόνος εκτέλεσης του forward pass και του backward pass. Τέλος, με μία αφαίρεση του τοπικού χρόνου εκπαίδευσης από τον χρόνο κατανεμημένης εκπαίδευσης υπολογίστηκε ο χρόνος που αντιστοιχεί στην επικοινωνία μεταξύ των κόμβων κατά την κατανεμημένη εκτέλεση.

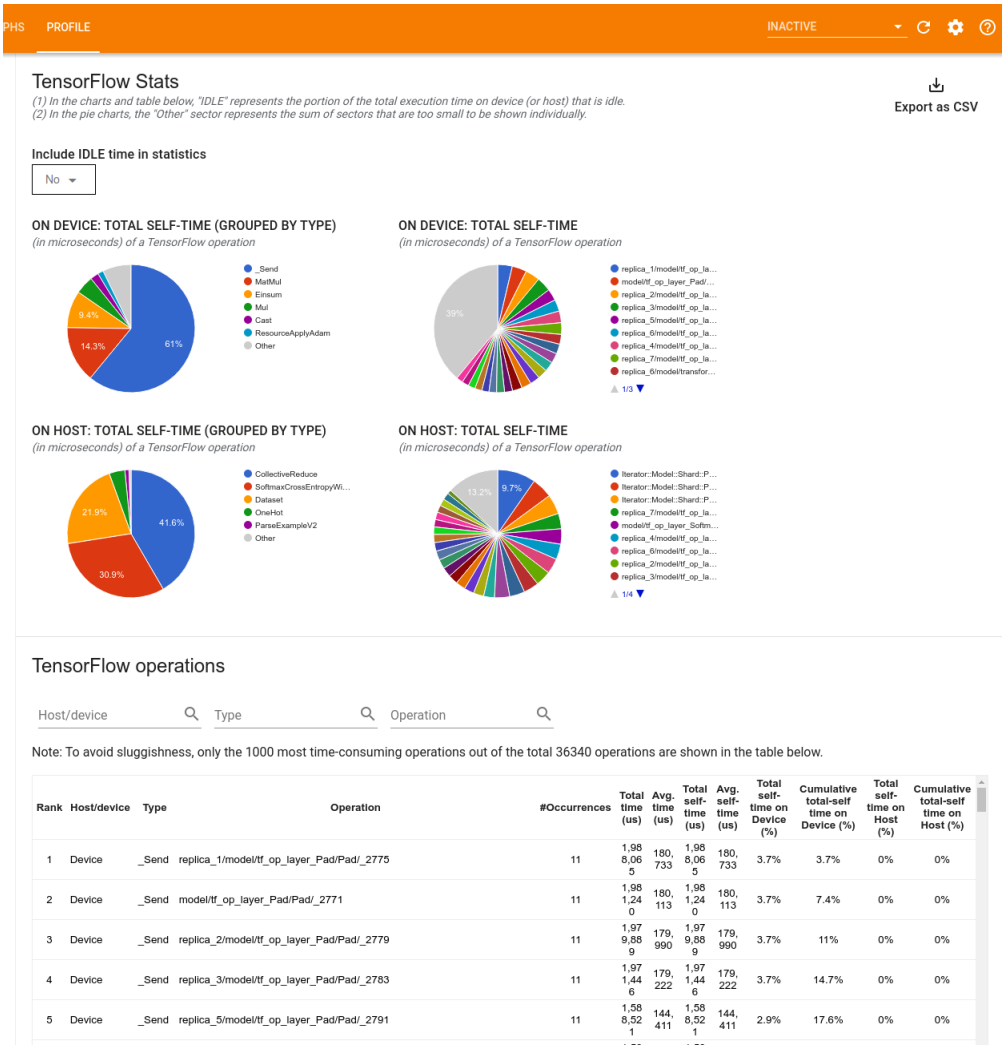
4.2.4 Φάση 3 - Ανάλυση τελεστών (profiling)

Τα τελικά πειράματα που εκτελέστηκαν πραγματοποιήθηκαν τοπικά σε ένα κόμβο, όπως και στη δεύτερη φάση δηλαδή, αλλά αυτή τη φορά με χρήση αναλυτή (profiler). Ο profiler είναι ένα εργαλείο που παρακολουθεί την εκτέλεση της εκπαίδευσης του δικτύου και καταγράφει αναλυτικά στατιστικά/μετρήσεις για όλα τα στάδιά της.

Κάθένα από τα 3 συστήματα μηχανικής μάθησης που χρησιμοποιήθηκαν παρέχει μία δική του υλοποίηση του profiler[54][55][56]. Το TensorFlow ειδικότερα, παρέχει το εργαλείο TensorBoard[57], το οποίο δημιουργεί από μόνο του πολύ χρήσιμες οπτικοποιήσεις των αποτελεσμάτων. Ένα παράδειγμα της διεπαφής χρήστη του TensorBoard φαίνεται στην εικόνα 4.4.

Από το profiling που πραγματοποιήθηκε, λοιπόν, μετρήθηκαν και καταγράφηκαν

οι χρόνοι των πιο χρονοβόρων τελεστών/πράξεων (operator) για κάθε σύστημα.



Εικόνα 4.4: TensorBoard διεπαφή χρήστη

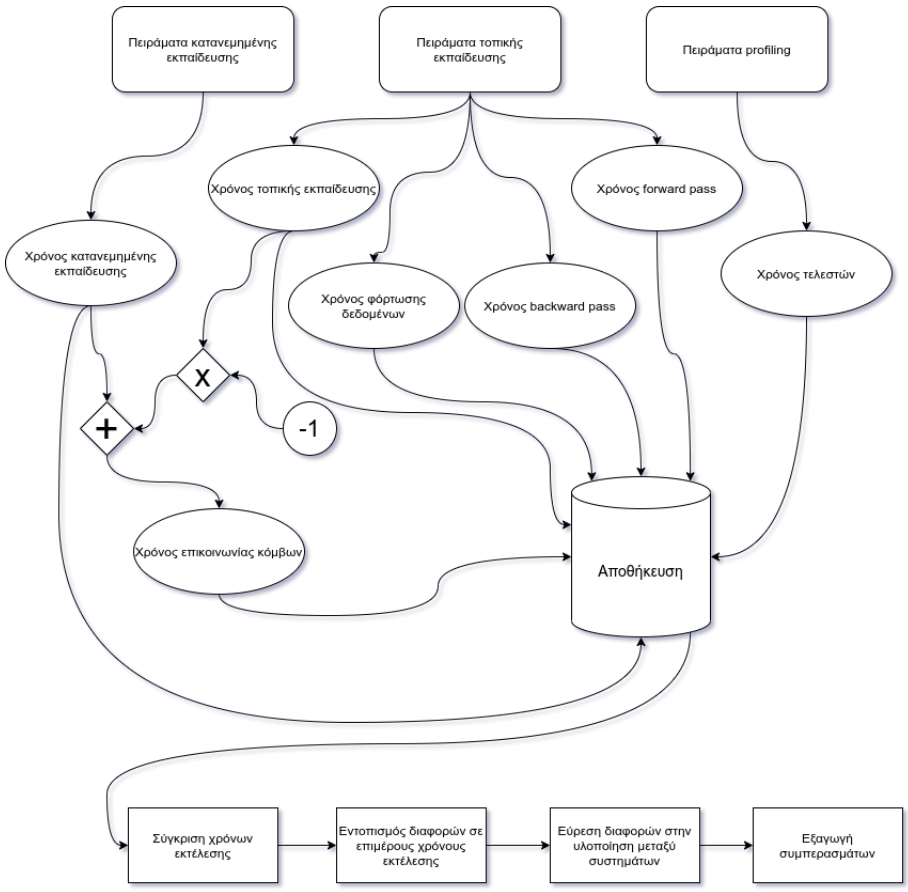
4.3 Σύγκριση αποτελεσμάτων

Το τελευταίο κομμάτι της μεθοδολογίας είναι η σύγκριση των αποτελεσμάτων. Η σύγκριση αυτή αφορά πάντα ίδιους συνδυασμούς μοντέλων/συνόλων δεδομένων και ίδια λειτουργία συγχρονισμού. Αρχικά η σύγκριση γίνεται πάνω στο συνολικό χρόνο κατανεμημένης εκπαίδευσης. Όπου παρατηρηθεί διαφορά, εξετάζεται στη συνέχεια εάν αυτή οφείλεται στο κόστος επικοινωνίας ή στην τοπική εκτέλεση ανά κόμβο. Η τοπική εκτέλεση στη συνέχεια αναλύεται σε τρία κομμάτια, τη φόρτωση των δεδομένων στη

μνήμη, το forward pass και το backward pass. Τα forward και backward pass στη συνέχεια, μπορούν να αναλυθούν στους χρόνους που κάνει ο κάθε operator που τα συνιστά.

Έτσι, η σύγκριση αυτή ξεκινάει από ένα αφαιρετικό επίπεδο και σιγά σιγά εμβαθύνει. Όταν εντοπιστεί η τελική αιτία που προκαλεί τη διαφορά στο χρόνο εκτέλεσης, διερευνάται περαιτέρω σε τι διαφέρει η υλοποίηση του συστήματος που υστερεί σε σχέση με τα υπόλοιπα ώστε να βγει ένα συμπέρασμα σχετικά με τα υπέρ και τα κατά του καθενός.

Σχηματικά η μεθοδολογία που ακολουθήθηκε συνοψίζεται στην εικόνα 4.5.



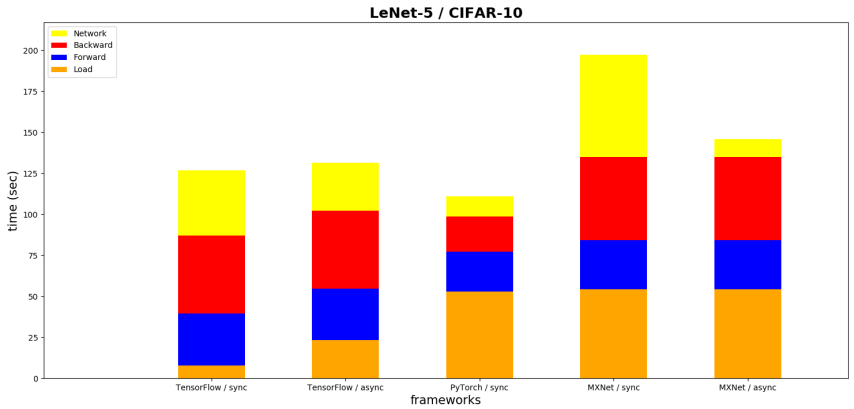
Εικόνα 4.5: Σχηματική αναπαράσταση της μεθοδολογίας που ακολουθήθηκε

Κεφάλαιο 5

Αποτελέσματα

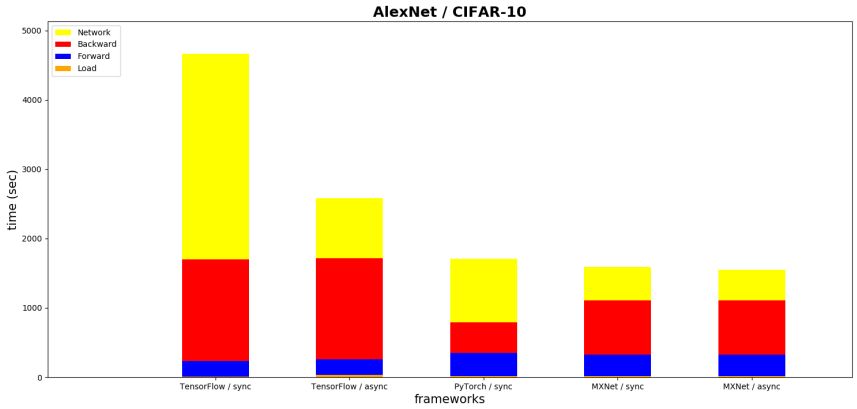
5.1 Αποτελέσματα μετρήσεων

Οι χρόνοι εκτέλεσης που καταγράφηκαν κατά τα κατανεμημένα πειράματα παρουσιάζονται στις εικόνες 5.1, 5.2, 5.3, 5.4, 5.5 και 5.6. Κάθε μπάρα στα διαγράμματα αυτά είναι χωρισμένη σε χρόνο φορτώματος δεδομένων (load), χρόνο εκτέλεσης του forward pass, χρόνο εκτέλεσης του backward pass και χρόνο που καταναλώνεται στην επικοινωνία μεταξύ των διεργασιών (network). Ακόμα, λόγω της διαφοράς σε τάξεις μεγέθους που υπάρχει μεταξύ των επιμέρους χρόνων, για να γίνουν καλύτερα αντιληπτά τα αποτελέσματα παρουσιάζονται αναλυτικά οι μετρήσεις στους πίνακες 5.1.

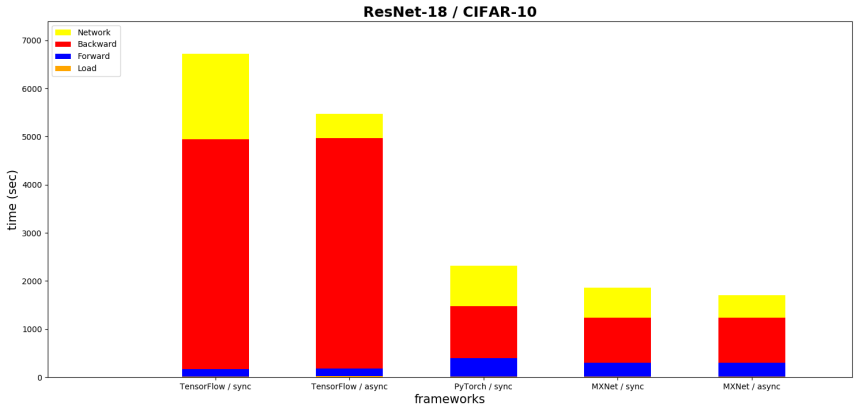


Εικόνα 5.1: Χρόνοι εκτέλεσης ανά σύστημα για LeNet-5 σε CIFAR-10

Σε όλα τα σχήματα, πλην του 5.1, φαίνεται το TensorFlow να είναι το πιο αργό από όλα, έως και ×4 φορές σε σχέση με το MXNet για ResNet-50 και έως και ×3 φορές σε σχέση με το PyTorch για ResNet-50. Παρατηρείται ότι το κομμάτι του



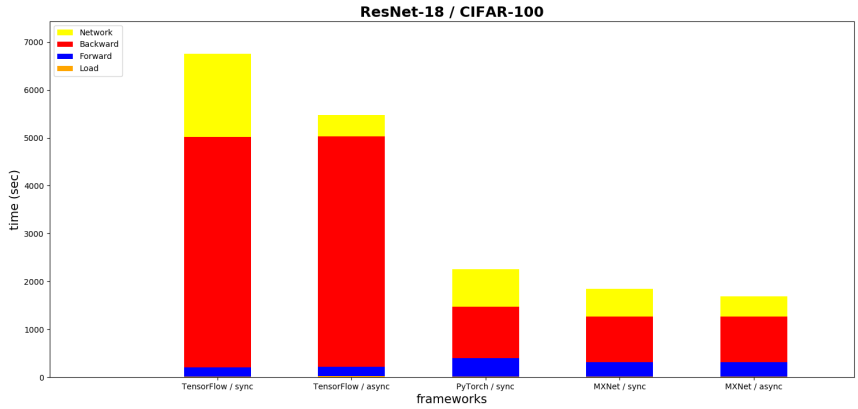
Εικόνα 5.2: Χρόνοι εκτέλεσης ανά σύστημα για AlexNet σε CIFAR-10



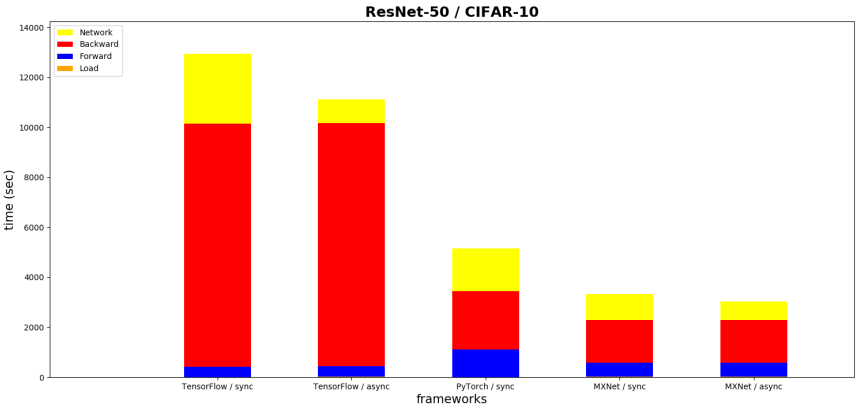
Εικόνα 5.3: Χρόνοι εκτέλεσης ανά σύστημα για ResNet-18 σε CIFAR-10

system	**total**	**load**	**forward**	**backward**	**network**
TensorFlow sync	126.81	7.9	31.56	47.64	39.7
TensorFlow async	131.57	23.1	31.56	47.64	29.26
PyTorch sync	110.91	53.02	24.21	21.29	12.39
MXNet sync	197.29	54.37	29.81	50.65	62.46
MXNet async	146.02	54.37	29.81	50.65	11.19

Πίνακας 5.1: Χρόνοι εκτέλεσης (sec) ανά σύστημα για LeNet-5 σε CIFAR-10



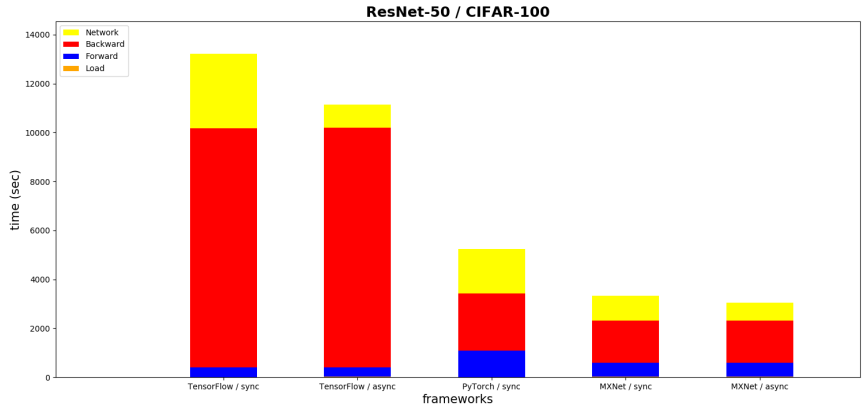
Εικόνα 5.4: Χρόνοι εκτέλεσης ανά σύστημα για ResNet-18 σε CIFAR-100



Εικόνα 5.5: Χρόνοι εκτέλεσης ανά σύστημα για ResNet-50 σε CIFAR-10

system	**total**	**load**	**forward**	**backward**	**network**
TensorFlow sync	4665.84	10.06	225.84	1459.75	2970.17
TensorFlow async	2582.86	29.26	225.84	1459.75	867.99
PyTorch sync	1711.26	15.66	337.22	438.87	919.51
MXNet sync	1590.56	17.04	306.13	787.5	479.89
MXNet async	1553.29	17.04	306.13	787.5	442.62

Πίνακας 5.2: Χρόνοι εκτέλεσης (sec) ανά σύστημα για AlexNet σε CIFAR-10



Εικόνα 5.6: Χρόνοι εκτέλεσης ανά σύστημα για ResNet-50 σε CIFAR-100

system	**total**	**load**	**forward**	**backward**	**network**
TensorFlow sync	6720.62	8.55	159.12	4782.03	1770.89
TensorFlow async	5474.09	24.81	159.12	4782.03	508.1
PyTorch sync	2321.08	11.39	388.68	1076.5	844.51
MXNet sync	1858.04	13.05	287.31	937.97	619.71
MXNet async	1700.4	13.05	287.31	937.97	462.07

Πίνακας 5.3: Χρόνοι εκτέλεσης (sec) ανά σύστημα για ResNet-18 σε CIFAR-10

system	**total**	**load**	**forward**	**backward**	**network**
TensorFlow sync	6757.06	8.63	190.98	4819.1	1738.32
TensorFlow async	5482.78	25.11	190.98	4819.1	447.56
PyTorch sync	2258.4	12.35	385.45	1079.25	781.35
MXNet sync	1846.3	11.82	296.35	959.1	579.03
MXNet async	1693.91	11.82	296.35	959.1	426.64

Πίνακας 5.4: Χρόνοι εκτέλεσης (sec) ανά σύστημα για ResNet-18 σε CIFAR-100

system	**total**	**load**	**forward**	**backward**	**network**
TensorFlow sync	12944.71	8.59	414.08	9735.55	2786.47
TensorFlow async	11115.83	24.93	414.08	9735.55	941.25
PyTorch sync	5156.35	10.78	1087.33	2356.63	1701.61
MXNet sync	3321.58	12.2	561.45	1725.25	1022.68
MXNet async	3025.18	12.2	561.45	1725.25	726.28

Πίνακας 5.5: Χρόνοι εκτέλεσης (sec) ανά σύστημα για ResNet-50 σε CIFAR-10

system	**total**	**load**	**forward**	**backward**	**network**
TensorFlow sync	13219.6	8.72	386.97	9780.24	3043.64
TensorFlow async	11138.12	25.32	386.97	9780.24	945.56
PyTorch sync	5234.54	11.46	1070.44	2330.65	1821.99
MXNet sync	3333.6	12.54	569.63	1731.95	1019.48
MXNet async	3043.24	12.54	569.63	1731.95	729.12

Πίνακας 5.6: Χρόνοι εκτέλεσης (sec) ανά σύστημα για ResNet-50 σε CIFAR-100

backward pass είναι αυτό που ευθύνεται κυρίως για τη διαφορά αυτή, όπως εξηγείται και στην υπό-ενότητα 5.2.1.

Από τα πειράματα AlexNet/CIFAR-10 στην εικόνα 5.2 παρατηρείται ότι, παρόλο που τα PyTorch και MXNet είναι κοντά σε συνολικό χρόνο, το πρώτο υπερτερεί σε τοπική εκτέλεση - και πιο συγκεκριμένα στο backward pass (βλ. υπό-ενότητα 5.2.2) - ενώ το δεύτερο σε χρόνο επικοινωνίας. Γενικότερα, στην περίπτωση του AlexNet βλέπουμε ότι το MXNet υπερτερεί κατά πολύ των άλλων δύο σε χρόνο επικοινωνίας, τόσο σε σύγχρονη όσο και σε ασύγχρονη λειτουργία (βλ. υπό-ενότητα 5.2.3). Συνολικά το MXNet είναι πολύ κοντά με το PyTorch σε αυτήν την περίπτωση, μόνο $\times 1.1$ φορές γρηγορότερο.

Από τα διαγράμματα των εικόνων 5.5 και 5.6 είναι φανερό ότι στην περίπτωση του ResNet-50 το MXNet υπερτερεί του PyTorch κατά $\times 1.6$ σε ταχύτητα. Πιο συγκεκριμένα υπάρχει σημαντική διαφορά σε όλη την τοπική εκτέλεση, τόσο στο backward pass όσο και στο forward pass, όπως εξηγείται στην υπό-ενότητα 5.2.4.

Αντίστοιχα, στα διαγράμματα των εικόνων 5.3 και 5.4 βλέπουμε ότι και για ResNet-18 πάλι το MXNet είναι γρηγορότερο του PyTorch, αυτή φορά κατά $\times 1.4$. Όπως και πριν, η διαφορά υπάρχει τόσο στο forward όσο και στο backward pass, όπως εξηγείται και στην υπό-ενότητα 5.2.5.

Η μόνη περίπτωση που το TensorFlow δεν είναι το πιο αργό από όλα είναι αυτή του LeNet-5, όπως γίνεται εμφανές στο διάγραμμα της εικόνας 5.1. Μάλιστα, σε τοπική εκτέλεση έχει τον καλύτερο χρόνο και από τα τρία. Αυτό βασικά οφείλεται στο φόρτωμα των δεδομένων στη μνήμη, όπως εξηγείται και στην υπό-ενότητα 5.2.7, και όχι στο κύριο υπολογιστικό κομμάτι στο οποίο, όπως έχει εξηγηθεί στην ενότητα 5.2.1, υστερεί των άλλων δύο. Στο συγκεκριμένο πείραμα πάντως, το TensorFlow είναι συνολικά $\times 1.6$ γρηγορότερο από το MXNet για σύγχρονη λειτουργία.

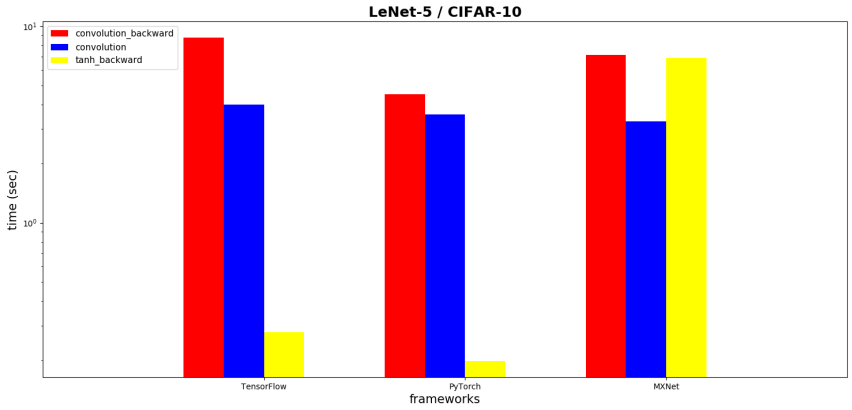
Στην προηγούμενη παράγραφο εξηγήθηκε πως το μόνο νευρωνικό στο οποίο το TensorFlow δεν κατέχει την τελευταία θέση είναι το LeNet-5. Στο ίδιο, επίσης, νευρωνικό παρατηρούμε την μόνη περίπτωση που το MXNet δεν κατέχει την πρώτη θέση, αλλά την τελευταία. Εξηγήθηκε προηγουμένως γιατί υπερτερεί το TensorFlow στην συγκεκριμένη περίπτωση, αλλά μένει να εξηγηθεί και γιατί για πρώτη φορά υπερτερεί το PyTorch έναντι του MXNet. Η κύρια διαφορά όπως φαίνεται και στο διάγραμμα της εικόνας 5.1 εντοπίζεται στο χρόνο του backward pass, όπως

αναλύεται και στην υπό-ενότητα 5.2.7. Το PyTorch στην περίπτωση αυτή είναι ×1.8 γρηγορότερο του MXNet.

Κάτι ακόμα που αξίζει να σημειωθεί είναι ότι, όπως φαίνεται και στο διάγραμμα της εικόνας 2.4.2, στην περίπτωση του LeNet-5 το TensorFlow απαιτεί οριακά λιγότερο χρόνο για τη σύγχρονη εκτέλεση παρά για την ασύγχρονη. Αυτό αρχικά μπορεί να μοιάζει παράδοξο, καθώς, όπως εξηγήθηκε στην υπό-ενότητα 2.4.2, ο συγχρονισμός προσθέτει ένα έξτρα κόστος στην επικοινωνία. Βγάζει νόημα όμως αν λάβουμε υπόψιν τον τρόπο με τον οποίο διαχωρίζεται το σύνολο δεδομένων στους κόμβους στις δύο υλοποιήσεις, σε συνδυασμό πάντα με το μικρό μέγεθος του νευρωνικού δικτύου, όπως αναλύεται και εκτενέστερα στην υπό-ενότητα 5.2.8.

5.2 Εξήγηση αποτελεσμάτων

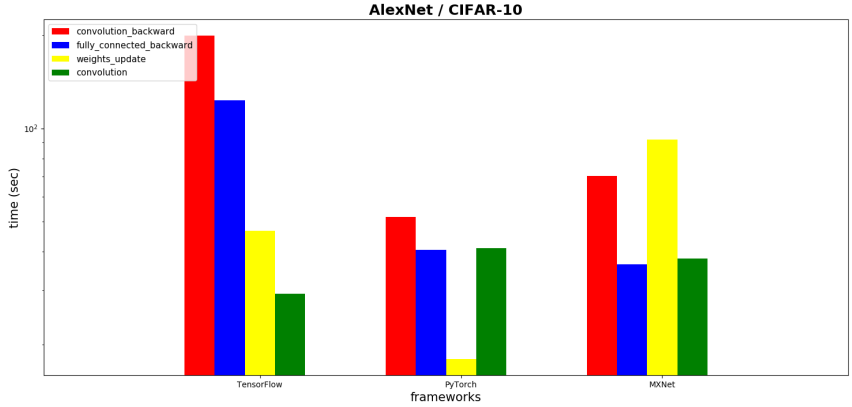
Σε αυτήν την ενότητα εξηγούνται περαιτέρω οι διαφορές σε χρόνους εκτέλεσης που παρατηρήθηκαν κατά την ενότητα 5.1. Για τον σκοπό αυτό, παρατίθενται και τα διαγράμματα με τους χρόνους εκτέλεσης των τελεστών ανά βήμα που προέκυψαν από το profiling. Τα διαγράμματα αυτά φαίνονται στις εικόνες 5.7, 5.8, 5.9 και 5.10.



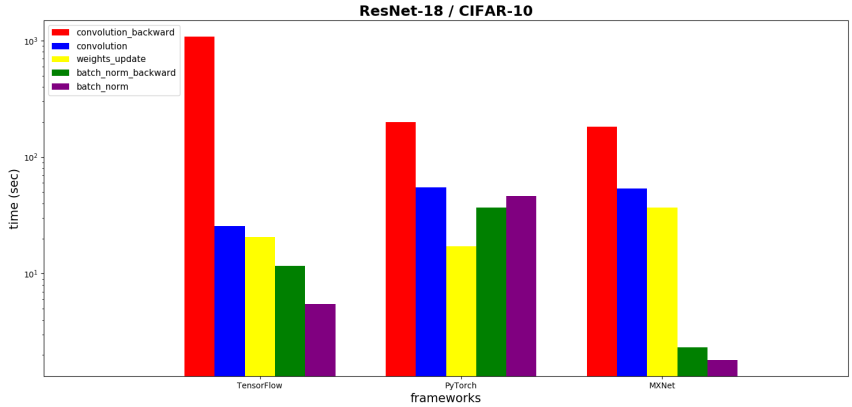
Εικόνα 5.7: Χρόνοι εκτέλεσης τελεστών ανά βήμα για LeNet-5 σε CIFAR-10

5.2.1 Αργό backward pass στο TensorFlow

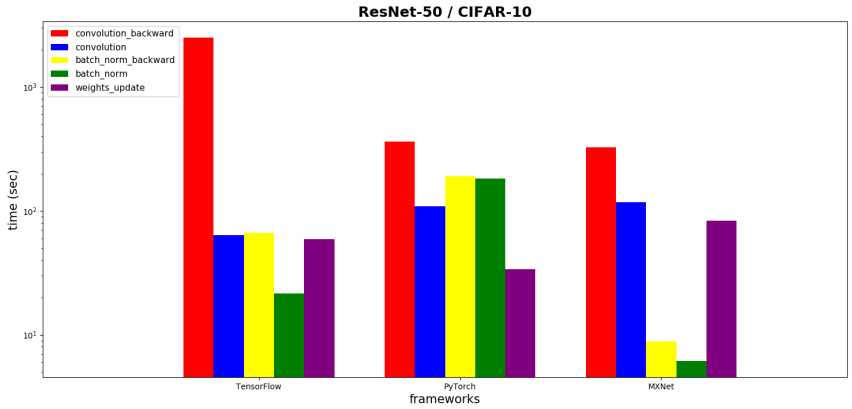
Το αργό backward pass που παρατηρήθηκε στο TensorFlow, είναι όπως αναφέρθηκε και ο κύριος λόγος που υστερεί των άλλων δύο σχεδόν σε όλα τα πειράματα. Το TensorFlow χρησιμοποιεί τη βιβλιοθήκη Eigen για την υλοποίηση των τελεστών του. Από την άλλη, τα PyTorch και MXNet βασίζονται στην oneDNN (επίσης γνωστή ως MKL-DNN). Η oneDNN παρέχει υλοποιήσεις τελεστών βαθιάς μηχανικής μάθησης, οι οποίοι είναι ειδικά βελτιστοποιημένοι για να



Εικόνα 5.8: Χρόνοι εκτέλεσης τελεστών ανά βήμα για AlexNet σε CIFAR-10



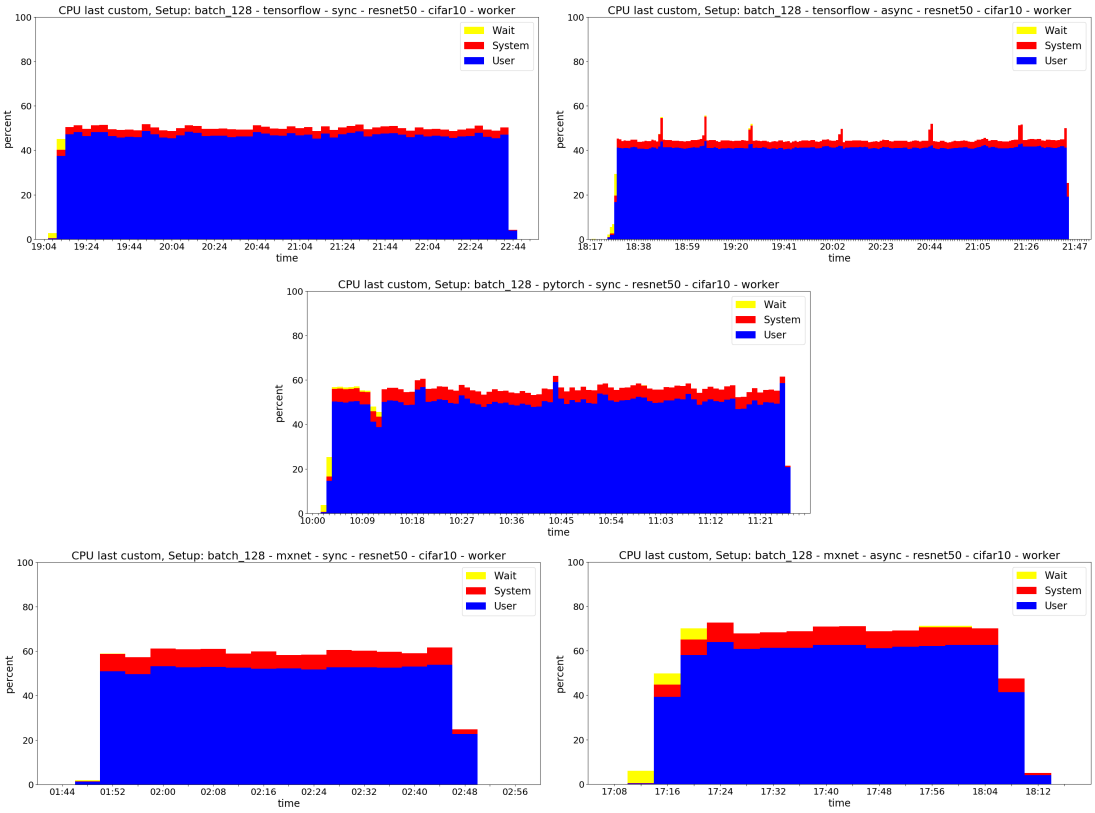
Εικόνα 5.9: Χρόνοι εκτέλεσης τελεστών ανά βήμα για ResNet-18 σε CIFAR-10



Εικόνα 5.10: Χρόνοι εκτέλεσης τελεστών ανά βήμα για ResNet-50 σε CIFAR-10

τρέχουν σε Intel επεξεργαστές. Δεδομένου ότι όλα τα πειράματα της εργασίας εκτελέστηκαν σε cluster αποτελούμενο απο Intel CPUs, σίγουρα τα συστήματα που βασίστηκαν στην oneDNN είχαν ένα ισχυρό προβάδισμα. Αυτό άλλωστε φαίνεται και από το profiling των τελεστών στα διαγράμματα των εικόνων 5.8, 5.9 και 5.10. Σε όλες τις περιπτώσεις, ο τελεστής οπισθοδρόμησης του συνελικτικού επιπέδου (convolution_backward), ο οποίος αποτελεί συνήθως και τον πιο χρονοβόρο τελεστή στα συνελικτικά νευρωνικά δίκτυα, φαίνεται να είναι πολύ πιο αργός στην υλοποίηση του TensorFlow. Ο συγκεκριμένος τελεστής του TensorFlow φτάνει να παίρνει μέχρι και $\times 7.8$ το χρόνο που χρειάζεται ο αντίστοιχος του MXNet για την περίπτωση του ResNet-50 (και $\times 7$ το χρόνο του αντίστοιχου τελεστή του PyTorch), όπως φαίνεται και στην εικόνα 5.10. Επίσης το γεγονός ότι το TensorFlow έχει πιο αργό υπολογισμό γίνεται φανερό και από τα διαγράμματα Ganglia για την κατανάλωση της CPU. Ενδεικτικά παρουσιάζεται η κατανάλωση CPU ενός worker για το πείραμα ResNet-50 σε CIFAR-10 στην εικόνα 5.11. Αναλυτικότερα, όλες οι μετρήσεις που καταγράφηκαν με το Ganglia μπορούν να βρεθούν στα παραρτήματα I, II, III και IV.

Αξίζει βέβαια να σημειωθεί ότι για όλα τα νευρωνικά πλην του LeNet-5 το TensorFlow έχει γρηγορότερο τελεστή όσον αφορά τη συνέλιξη του forward pass (convolution), που φτάνει να είναι μέχρι και $\times 2.1$ γρηγορότερος του αντίστοιχου τελεστή του PyTorch για το ResNet-18. Παρά, όμως, τη γρηγορότερη υλοποίηση της Eigen για έναν από τους σημαντικότερους τελεστές του forward pass, τη μεγαλύτερη επίπτωση την έχουν οι τελεστές του backward pass, και έτσι τα PyTorch και MXNet είναι στην πλειοψηφία των πειραμάτων πιο γρήγορα.



Εικόνα 5.11: Κατανάλωση CPU ανά worker για ResNet-50 σε CIFAR-10

5.2.2 Backward pass για AlexNet σε PyTorch και MXNet

Όπως παρατηρήθηκε παραπάνω, το MXNet υστερεί στο backward pass έναντι του PyTorch στα πειράματα με το AlexNet. Παρότι οι περισσότεροι τελεστές τους είναι κοντά σε χρόνους εκτέλεσης, πράγμα λογικό αφού αμφότερα τα συστήματα βασίζονται στη βιβλιοθήκη oneDNN, φαίνεται να υπάρχει ένας τελεστής στο MXNet που είναι σε αυτήν την περίπτωση πιο αργός ακόμα και από τον convolution_backward. Ο τελεστής αυτός είναι ο weights_update, όπως φαίνεται και στο διάγραμμα της εικόνας 5.8, και η πράξη που επιτελεί είναι η ανανέωση των βαρών του νευρωνικού (αφότου υπολογιστεί το διάνυσμα κλίσης). Συγκεκριμένα ο αντίστοιχος τελεστής του PyTorch είναι $\times 5.1$ γρηγορότερος.

5.2.3 Κόστος επικοινωνίας

Όσον αφορά το κόστος επικοινωνίας μεταξύ των διεργασιών του cluster, πρέπει να σημειωθεί αρχικά ότι ενώ στην ασύγχρονη λειτουργία ο χρόνος αυτός καταναλώνεται σε μεταφορά δεδομένων στο δίκτυο, στη σύγχρονη λειτουργία καταναλώνεται επιπλέον χρόνος για την αναμονή περάτωσης της επεξεργασίας του πιο αργού worker. Γίνεται φανερό, λοιπόν, ότι οι χρόνοι επικοινωνίας μπορούν να συγκριθούν άμεσα στην περίπτωση της ασύγχρονης λειτουργίας, ενώ για τη σύγχρονη πρέπει να ληφθεί υπόψιν ότι το κόστος συγχρονισμού θα είναι κατ' αναλογία του χρόνου της τοπικής εκτέλεσης.

Σύμφωνα με σχεδόν όλα τα πειράματα (πλην του LeNet-5 όπου τα νούμερα είναι πολύ μικρά για να υπάρξει ξεκάθαρο συμπέρασμα) τη γρηγορότερη επικοινωνία φαίνεται να την έχει το MXNet. Ιδιαίτερα στην περίπτωση του AlexNet (βλ. πίνακα 5.2), ενώ στα άλλα δύο συστήματα το κόστος επικοινωνίας φαίνεται να κλιμακώνεται με την αύξηση των παραμέτρων (βλ. πίνακα 2.1), το MXNet δε δείχνει να επηρεάζεται και έχει μάλιστα ισχυρό προβάδισμα στο συγκεκριμένο πείραμα χάρη στη μείωση του κόστους επικοινωνίας. Ειδικότερα, στο AlexNet παρότι το MXNet έχει πιο αργή τοπική εκτέλεση, έχει $\times 1.9$ γρηγορότερη επικοινωνία από το PyTorch. Βέβαια, στην Parameter Server αρχιτεκτονική δεν υπάρχει μεγάλη διαφορά μεταξύ ασύγχρονων MXNet και TensorFlow (πλην του AlexNet).

Παρότι στη σύγχρονη λειτουργία υπάρχει ένας όρος (κόστος συγχρονισμού) που αυξάνεται σε αναλογία με το χρόνο τοπικής εκτέλεσης, και άρα δεν μπορεί να γίνει ξεκάθαρη σύγκριση στις περισσότερες περιπτώσεις, φαίνεται να μην υπάρχει μεγάλη διαφορά στον καθαρό χρόνο επικοινωνίας ανάμεσα στα τρία συστήματα. Λίγο πιο γρήγορο είναι το MXNet, όπως προαναφέρθηκε, και κυρίως στο AlexNet, αλλά πέρα από αυτό οι διαφορές δεν επηρεάζουν σε μεγάλο βαθμό τον τελικό νικητή.

Σε αυτό το κομμάτι, το MXNet οφείλει την πρωτιά του στην προσαρμοσμένη υλοποίηση του RPC πρωτοκόλλου που παρέχει μέσω του launch module (βλ. υπό-ενότητα 3.3.4) έναντι του gRPC που χρησιμοποιεί το TensorFlow (βλ. υπό-ενότητα 3.1.5) και του Gloo backend στο οποίο βασίζεται το PyTorch (βλ. υπό-ενότητα

3.2.4).

5.2.4 Τοπική εκτέλεση για ResNet-50 σε PyTorch και MXNet

Όπως παρατηρήθηκε το MXNet είναι πιο γρήγορο του PyTorch στα πειράματα με ResNet-50 ως νευρωνικό. Η διαφορά εντοπίζεται κυρίως στην τοπική εκτέλεση, τόσο στο forward pass (×1.9 γρηγορότερο) όσο και στο backward pass (×1.4 γρηγορότερο), όπως φαίνεται στον πίνακα 5.5. Αναλύοντας τον υπολογισμό με τη βοήθεια του profiling (βλ. εικόνα 5.10), φαίνεται ότι ο κύριος τελεστής που ευθύνεται για τη διαφορά αυτή στο forward pass είναι ο batch_norm (δηλαδή το επίπεδο κανονικοποίησης) ο οποίος είναι ×29.7 πιο αργός για το PyTorch. Κατά το backward pass φαίνεται ότι ο αντίστοιχος τελεστής ευθύνεται για αυτή τη διαφορά (batch_norm_backward), αφού στην υλοποίηση του PyTorch είναι ×21.4 πιο αργός σε σχέση με το MXNet. Παρατηρείται, λοιπόν, ότι το MXNet παρέχει πολύ πιο γρήγορη υλοποίηση για το επίπεδο αυτό τόσο στο forward όσο και στο backward pass.

5.2.5 Τοπική εκτέλεση για ResNet-18 σε PyTorch και MXNet

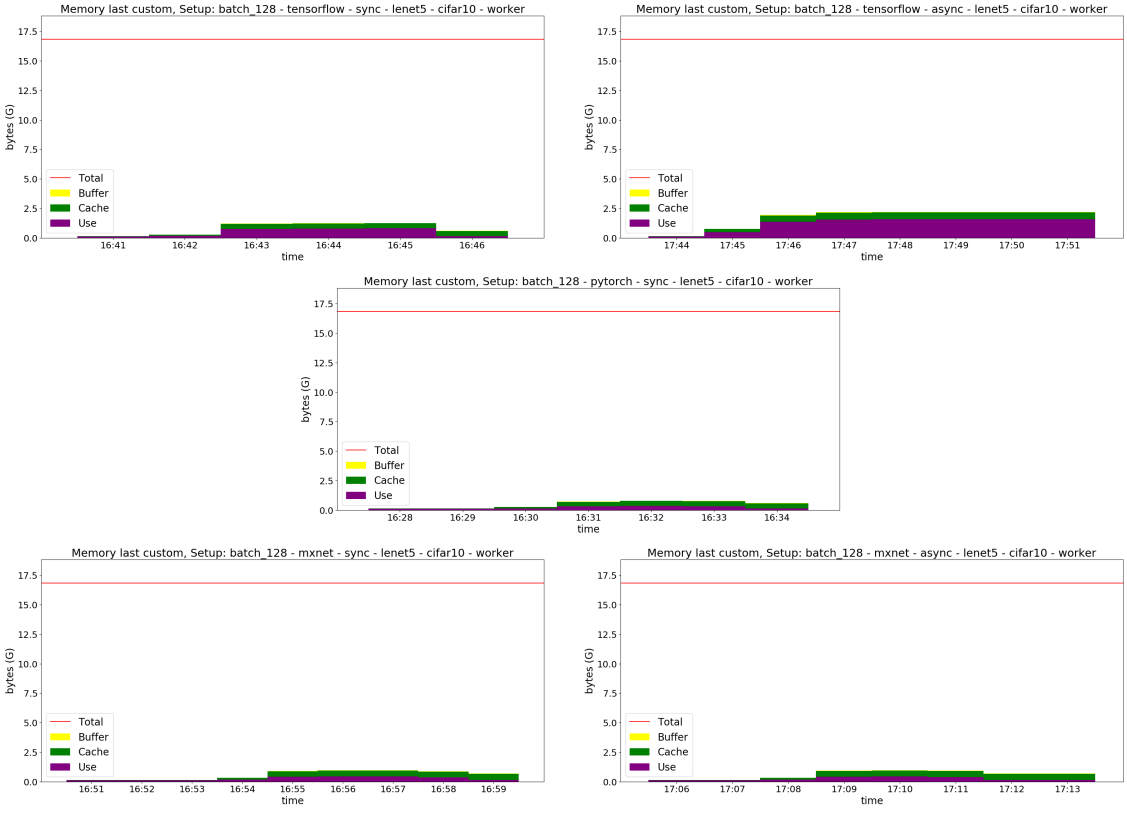
Και εδώ η διαφορά εντοπίζεται στο forward pass (×1.4 γρηγορότερο) και στο backward pass (×1.2 γρηγορότερο), όπως φαίνεται στον πίνακα 5.3. Αναλύοντας τον υπολογισμό με τη βοήθεια του profiling (βλ. εικόνα 5.9), φαίνεται ότι ο τελεστής batch_norm είναι ×25.7 πιο αργός για το PyTorch, και ο batch_norm_backward είναι ×15.9 πιο αργός σε σχέση με το MXNet.

5.2.6 Φόρτωμα δεδομένων στη μνήμη

Όσον αφορά το φόρτωμα του συνόλου δεδομένων στη μνήμη για επεξεργασία του κάθε batch, υπάρχουν δύο προσεγγίσεις που ακολουθούνται από τα τρία εν λόγω συστήματα. Κατά την πρώτη, κάθε batch δεδομένων που φορτώνεται αποθηκεύεται στη συνέχεια στην cache, ώστε την επόμενη φορά που θα χρειαστεί (στην επόμενη εποχή συνήθως) να φορτωθεί πολύ πιο γρήγορα στη μνήμη. Κατά τη δεύτερη προσέγγιση, κάθε batch που χρειάζεται φορτώνεται κατευθείαν από το δίσκο, όσες φορές και να ζητηθεί το ίδιο batch. Σε αυτήν την προσέγγιση παρατηρείται χαμηλότερη κατανάλωση των πόρων του κόμβου, αλλά πιο αργό φόρτωμα δεδομένων.

Το TensorFlow χρησιμοποιεί την αποθήκευση στην cache, ενώ τα PyTorch και MXNet ακολουθούν τη δεύτερη προσέγγιση. Ως αποτέλεσμα, το TensorFlow απαιτεί περισσότερη μνήμη, όπως φαίνεται και στο διάγραμμα της εικόνας 5.12. Στο διάγραμμα αυτό, το TensorFlow χρησιμοποιεί περισσότερη μνήμη σε ασύγχρονη

λειτουργία σε σχέση με τη σύγχρονη, όπως εξηγείται και παρακάτω στην υπό-ενότητα 5.2.8. Δεδομένου ότι και τα τρία συστήματα φορτώνουν batch δεδομένων παράλληλα με την επεξεργασία τους, η φόρτωση των batch από το δίσκο δεν προσθέτει μεγάλη καθυστέρηση για μεγάλα νευρωνικά (όπου έχουμε και μεγάλο χρόνο επεξεργασίας, δηλαδή forward και backward pass). Στην περίπτωση του LeNet-5, όμως, η φόρτωση του κάθε batch είναι ίδιας τάξης μεγέθους με την επεξεργασία του batch. Ως εκ τούτου δε γίνεται καλή εκμετάλλευση της παράλληλης φόρτωσης διαδοχικών batch και επομένως προστίθεται μία καθυστέρηση για τα συστήματα που δε χρησιμοποιούν την cache. Η καθυστέρηση αυτή, σε συνδυασμό με το γεγονός ότι για μικρά δίκτυα η φόρτωση δεδομένων αποτελεί σημαντικό κομμάτι (όσον αφορά το χρόνο) της εκπαίδευσης του δικτύου, δίνει ένα ισχυρό προβάδισμα στο TensorFlow έναντι των άλλων δύο. Πιο συγκεκριμένα, για τα πειράματα του LeNet-5 το TensorFlow είναι $\times 6.9$ γρηγορότερο από το MXNet και $\times 6.7$ γρηγορότερο από το PyTorch στο κομμάτι της φόρτωσης των δεδομένων, όπως φαίνεται και στον πίνακα 5.1.



Εικόνα 5.12: Κατανάλωση Μνήμης ανά worker για ResNet-50 σε CIFAR-10

5.2.7 Backward pass για LeNet-5

Το MXNet, όπως παρατηρήθηκε από την εικόνα 5.1, έχει την πιο αργή υλοποίηση για LeNet-5. Εξηγήθηκε παραπάνω ότι ένα μειονέκτημα που έχει αυτό και το PyTorch έναντι του TensorFlow είναι το αργό φόρτωμα δεδομένων (στην συγκεκριμένη περίπτωση). Ακόμα και έτσι, θα περιμέναμε να βρίσκεται κοντά με το PyTorch, πράγμα που δε συμβαίνει. Η διαφορά που έχουν μεταξύ τους τα PyTorch και MXNet εντοπίζεται στο backward pass. Ο τελεστής του backward pass που κάνει τη διαφορά, όπως φαίνεται και από το profiling στην εικόνα 5.7, είναι αυτός της συνάρτησης ενεργοποίησης tanh_backward. Για τον τελεστή αυτόν, το MXNet είναι μάλιστα $\times 34.8$ πιο αργό από το PyTorch και $\times 24.8$ πιο αργό από το TensorFlow στο συγκεκριμένο πείραμα.

5.2.8 Σύγχρονο και ασύγχρονο TensorFlow για LeNet-5

Όπως παρατηρήθηκε παραπάνω, το ασύγχρονο TensorFlow φαίνεται να είναι πιο αργό από το σύγχρονο στην περίπτωση του LeNet-5. Έχει ήδη εξηγηθεί (βλ. υπό-ενότητα 2.4.2) ότι η ασύγχρονη λειτουργία είναι πάντα πιο γρήγορη από τη σύγχρονη. Και αφού μιλάμε για το ίδιο σύστημα (TensorFlow), η υλοποίηση του νευρωνικού και κατ' επέκταση ο υπολογισμός του forward και του backward pass οφείλουν να είναι ίδια. Ο λόγος, λοιπόν, που παρατηρείται αυτή η περίεργη συμπεριφορά έχει να κάνει με το φόρτωμα των δεδομένων.

Στη σύγχρονη λειτουργία, κάθε worker φορτώνει στη μνήμη το μέρος του συνόλου δεδομένων που του αντιστοιχεί (το ένα τρίτο στην περίπτωσή μας) και επεξεργάζεται το ίδιο υπό-σύνολο δεδομένων καθ' όλη τη διάρκεια της εκπαίδευσης του νευρωνικού. Αντίθετα, στην ασύγχρονη λειτουργία ο κάθε worker φορτώνει ολόκληρο το σύνολο δεδομένων, και επεξεργάζεται κάθε φορά το batch που του δρομολογεί ο coordinator. Επομένως, στην ασύγχρονη λειτουργία γίνεται λίγο καλύτερο ανακάτεμα (shuffling) των δεδομένων, αφού σε οποιονδήποτε κόμβο μπορεί να τύχει οποιοδήποτε batch, υπάρχει όμως και μεγαλύτερο κόστος φόρτωσης των δεδομένων αφού ο όγκος τους για κάθε κόμβο τριπλασιάζεται. Από τον πίνακα 5.1 βλέπουμε ότι το σύγχρονο TensorFlow φορτώνει τα δεδομένα $\times 2.9$ γρηγορότερα από το ασύγχρονο και ξοδεύει επίσης περισσότερο χρόνο στην επικοινωνία μεταξύ των διεργασιών. Ακόμα, στα διαγράμματα της εικόνας 5.12 φαίνεται η αυξημένη κατανάλωση μνήμης του ασύγχρονου TensorFlow λόγω του φορτώματος τριπλάσιου μεγέθους δεδομένων σε σχέση με το σύγχρονο.

Κεφάλαιο 6

Συμπεράσματα και επεκτάσεις

Στην εργασία αυτή αξιολογήθηκαν συγκριτικά τρία από τα πιο διάσημα συστήματα βαθιάς νευρωνικής μάθησης, τα TensorFlow, PyTorch και MXNet. Βασικός στόχος ήταν μέσω πειραμάτων σε κατανεμημένη διάταξη υπολογιστών (cluster) να εντοπιστούν και να εξηγηθούν οι διαφορές στις υλοποιήσεις τους.

Από την ανάλυση αυτή προέκυψαν τα εξής κύρια συμπεράσματα:

• Τα PyTorch και MXNet είναι ταχύτερα στο υπολογιστικό κομμάτι από το TensorFlow όταν τρέχουν σε επεξεργαστές τύπου Intel, χάρη στη βιβλιοθήκη oneDNN που παρέχει ειδικά βελτιστοποιημένους τελεστές για επεξεργαστές Intel.

• Στο υπολογιστικό κομμάτι, το PyTorch φαίνεται να είναι πιο γρήγορο από το MXNet σε συνελικτικά νευρωνικά δίκτυα με απλή δομή (συγκεκριμένα εξετάστηκαν τα LeNet-5 και AlexNet), ενώ το MXNet υπερέχει σε δίκτυα με πιο σύνθετη δομή (ResNet-18, ResNet-50). Οι διαφορές αυτές προκύπτουν από μεμονωμένες υλοποιήσεις τελεστών, καθότι συνολικά οι περισσότερες βασίζονται στην βιβλιοθήκη oneDNN και για τα δύο συστήματα.

• Στο διάβασμα των δεδομένων το TensorFlow είναι πιο γρήγορο από τα άλλα δύο καθώς χρησιμοποιεί caching. Επομένως, μετά την πρώτη φορά που τα φορτώνει στη μνήμη, μπορεί να τα ξαναφέρει ταχύτατα για επεξεργασία. Το caching βέβαια αυξάνει την κατανάλωση της μνήμης. Το πλεονέκτημα αυτό γίνεται περισσότερο διακριτό σε συνελικτικά δίκτυα μικρού μεγέθους (LeNet-5), όπου η προ-φόρτωση δεδομένων δεν αξιοποιείται επαρκώς και ο χρόνος διαβάσματος είναι συγκρίσιμος με το χρόνο υπολογισμού.

• Το MXNet έχει την ταχύτερη επικοινωνία μεταξύ των διεργασιών του cluster. Η διαφορά βέβαια αυτή δεν είναι τόσο μεγάλη ώστε να επηρεάσει σημαντικά το τελικό αποτέλεσμα, με εξαίρεση την περίπτωση του AlexNet, το οποίο είναι και το νευρωνικό με τις περισσότερες παραμέτρους από όσα δοκιμάστηκαν.

Η παρούσα εργασία θα μπορούσε να επεκταθεί από διάφορες πλευρές, όπως περιγράφεται στη συνέχεια.

• Αρχικά, θα μπορούσαν να μελετηθούν οι επιπτώσεις διαφορετικών παραμέτρων εκπαίδευσης, όπως οι διαστάσεις της εικόνας, το μέγεθος του batch ή ο τύπος του βελτιστοποιητή.

• Ακόμα, ενδιαφέρον θα ήταν να εξεταστούν νευρωνικά δίκτυα με διαφορετική δομή (όπως το InceptionNet) και διαφορετικά σύνολα δεδομένων.

• Τέλος θα ήταν πολύ χρήσιμο να γίνει αντίστοιχη ανάλυση στο πεδίο της επεξεργασίας φυσικής γλώσσας με εκπαίδευση αναδρομικών νευρωνικών δικτύων (recurrent neural networks - RNN).

Αναφορές

[1] Yangqing Jia et al. «Caffe: Convolutional Architecture for Fast Feature Embedding». In: *CoRR* abs/1408.5093 (2014). arXiv: 1408.5093. URL: http://arxiv.org/abs/1408.5093.

[2] Seiya Tokui and Kenta Oono. «Chainer : a Next-Generation Open Source Framework for Deep Learning». In: 2015.

[3] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[4] Tianqi Chen et al. *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. 2015. arXiv: 1512.01274 [cs.DC].

[5] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603 . 04467 [cs.DC].

[6] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https : / / proceedings . neurips . cc / paper / 2019 / file / bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[7] Mu Li et al. «Scaling Distributed Machine Learning with the Parameter Server». In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https : / / www . usenix . org / conference / osdi14 / technical - sessions/presentation/li_mu.

[8] Pitch Patarasuk and Xin Yuan. «Bandwidth optimal all-reduce algorithms for clusters of workstations». In: *Journal of Parallel and Distributed Computing* 69.2 (2009), pp. 117–124. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2008.09.002. URL: https://www.sciencedirect.com/science/article/pii/S0743731508001767.

[9] Jianmin Chen et al. «Revisiting Distributed Synchronous SGD». In: *CoRR* abs/1604.00981 (2016). arXiv: 1604.00981. URL: http://arxiv.org/abs/1604.00981.

[10] Shanshan Zhang et al. «Asynchronous stochastic gradient descent for DNN training». In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 6660–6663. DOI: 10.1109/ICASSP.2013.6638950.

[11] Xavier Glorot and Yoshua Bengio. «Understanding the difficulty of training deep feedforward neural networks». In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

[12] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. 2nd. Chapman Hall/CRC, 2014. ISBN: 1466583282.

[13] Jun Han and Claudio Moraga. «The influence of the sigmoid function parameters on the speed of backpropagation learning». In: *From Natural to Artificial Neural Computation*. Ed. by José Mira and Francisco Sandoval. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 195–201. ISBN: 978-3-540-49288-7.

[14] B.L. Kalman and S.C. Kwasny. «Why tanh: choosing a sigmoidal function». In: *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*. Vol. 4. 1992, 578–581 vol.4. DOI: 10.1109/IJCNN.1992.227257.

[15] Abien Fred Agarap. «Deep Learning using Rectified Linear Units (ReLU)». In: *CoRR* abs/1803.08375 (2018). arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375.

[16] John Bridle. «Training Stochastic Model Recognition Algorithms as Networks can Lead to Maximum Mutual Information Estimation of Parameters». In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 2. Morgan-Kaufmann, 1990. URL:

https : / / proceedings . neurips . cc / paper / 1989 / file / 0336dcbab05b9d5ad24f4333c7658a0e-Paper.pdf.

[17] Keiron O'Shea and Ryan Nash. «An Introduction to Convolutional Neural Networks». In: *CoRR* abs/1511.08458 (2015). arXiv: 1511.08458. URL: http://arxiv.org/abs/1511.08458.

[18] Nilesh Vijayrania. *Different Normalization Layers in Deep Learning*. URL: https : / / towardsdatascience . com / different - normalization-layers-in-deep-learning-1a7214ff71d6 (visited on 06/22/2021).

[19] Y. Lecun et al. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https : / / proceedings . neurips . cc / paper / 2012 / file / c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[21] Kaiming He et al. «Deep Residual Learning for Image Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[22] Zhilu Zhang and Mert R. Sabuncu. «Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels». In: *CoRR* abs/1805.07836 (2018). arXiv: 1805 . 07836. URL: http://arxiv.org/abs/1805.07836.

[23] H. Robbins. «A Stochastic Approximation Method». In: *Annals of Mathematical Statistics* 22 (2007), pp. 400–407.

[24] Sebastian Ruder. «An overview of gradient descent optimization algorithms». In: *CoRR* abs/1609.04747 (2016). arXiv: 1609 . 04747. URL: http://arxiv.org/abs/1609.04747.

[25] Lutz Prechelt. «Early Stopping - But When?» In: *Neural Networks: Tricks of the Trade*. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69. ISBN: 978-3-540-49430-0. DOI: 10 . 1007 / 3 - 540 - 49430 - 8_3. URL: https : //doi.org/10.1007/3-540-49430-8_3.

[26] Christopher J. Shallue et al. «Measuring the Effects of Data Parallelism on Neural Network Training». In: *CoRR* abs/1811.03600 (2018). arXiv: 1811.03600. URL: http://arxiv.org/abs/1811.03600.

[27] «Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform». In: *CoRR* abs/1809.02839 (2018). Withdrawn. arXiv: 1809.02839. URL: http://arxiv.org/abs/1809.02839.

[28] Mu Li et al. «Parameter Server for Distributed Machine Learning». In: 2013.

[29] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. «Optimization of Collective Communication Operations in MPICH». In: *The International Journal of High Performance Computing Applications* 19.1 (2005), pp. 49–66. DOI: 10.1177/1094342005051521. eprint: https://doi.org/10.1177/1094342005051521. URL: https://doi.org/10.1177/1094342005051521.

[30] *TensorFlow*. URL: https://en.wikipedia.org/wiki/TensorFlow (visited on 06/22/2021).

[31] Jeffrey Dean et al. «Large Scale Distributed Deep Networks». In: *NIPS*. 2012.

[32] *Effective TensorFlow 2*. URL: https://www.tensorflow.org/guide/effective_tf2 (visited on 06/22/2021).

[33] *Distributed training with TensorFlow*. URL: https://www.tensorflow.org/guide/distributed_training (visited on 06/22/2021).

[34] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. «GRPC: A Communication Cooperation Mechanism in Distributed Systems». In: *SIGOPS Oper. Syst. Rev.* 27.3 (July 1993), pp. 75–86. ISSN: 0163-5980. DOI: 10.1145/155870.155881. URL: https://doi.org/10.1145/155870.155881.

[35] Graham Neubig et al. *DyNet: The Dynamic Neural Network Toolkit*. 2017. arXiv: 1701.03980 [stat.ML].

[36] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. «The NumPy Array: A Structure for Efficient Numerical Computation». In: *Computing in Science Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37.

[37] *PyTorch Distributed Overview*. URL: https://pytorch.org/tutorials/beginner/dist_overview.html (visited on 06/22/2021).

[38] *Distributed Data Parallel*. URL: https://pytorch.org/docs/master/notes/ddp.html (visited on 06/22/2021).

[39] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. «An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum». In: *ACM Trans. Math. Softw.* 28.2 (June 2002), pp. 239–267. ISSN: 0098-3500. DOI: 10.1145/567806.567810. URL: https://doi.org/10.1145/567806.567810.

[40] *About Gluon*. URL: https://mxnet.apache.org/versions/1.4.1/gluon/index.html (visited on 06/22/2021).

[41] *Distributed Training in MXNet*. URL: https://mxnet.apache.org/versions/1.7.0/api/faq/distributed_training (visited on 06/22/2021).

[42] *Remote procedure call*. URL: https://en.wikipedia.org/wiki/Remote_procedure_call (visited on 06/22/2021).

[43] *Message Passing Interface*. URL: https://en.wikipedia.org/wiki/Message_Passing_Interface (visited on 06/22/2021).

[44] *Distributed Communication Package - torch.distributed*. URL: https://pytorch.org/docs/stable/distributed.html (visited on 06/22/2021).

[45] *NVIDIA Collective Communications Library (NCCL)*. URL: https://developer.nvidia.com/nccl (visited on 06/22/2021).

[46] *Secure Shell Protocol*. URL: https://en.wikipedia.org/wiki/Secure_Shell_Protocol (visited on 06/22/2021).

[47] *Apache Hadoop YARN*. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html (visited on 06/22/2021).

[48] *Oracle Grid Engine*. URL: https://en.wikipedia.org/wiki/Oracle_Grid_Engine (visited on 06/22/2021).

[49] *Eigen*. URL: https://eigen.tuxfamily.org/index.php?title=Main_Page (visited on 06/22/2021).

[50] *oneAPI Deep Neural Network Library (oneDNN)*. URL: https://github.com/oneapi-src/oneDNN (visited on 06/22/2021).

[51] Alex Krizhevsky. «Learning Multiple Layers of Features from Tiny Images». In: *University of Toronto* (May 2012).

[52] *CIFAR-10 and CIFAR-100 datasets*. URL: https://www.cs.toronto.edu/~kriz/cifar.html (visited on 06/22/2021).

[53] Matthew L Massie, Brent N Chun, and David E Culler. «The ganglia distributed monitoring system: design, implementation, and experience». In: *Parallel Computing* 30.7 (2004), pp. 817–840. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2004.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0167819104000535.

[54] *TensorFlow Profiler: Profile model performance*. URL: https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras (visited on 06/22/2021).

[55] *PyTorch Profiler*. URL: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html (visited on 06/22/2021).

[56] *Profiling MXNet Models*. URL: https://mxnet.apache.org/versions/1.8.0/api/python/docs/tutorials/performance/backend/profiler.html (visited on 06/22/2021).

[57] *Optimize TensorFlow performance using the Profiler*. URL: https://www.tensorflow.org/guide/profiler (visited on 06/22/2021).

[58] Seiya Tokui et al. «Chainer: A Deep Learning Framework for Accelerating the Research Cycle». In: *CoRR* abs/1908.00213 (2019). arXiv: 1908.00213. URL: http://arxiv.org/abs/1908.00213.

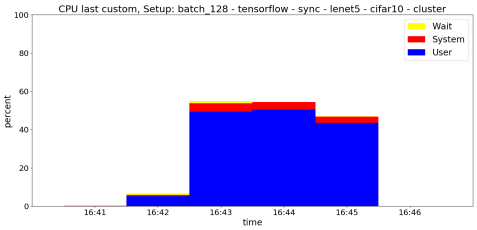
[59] Adam Paszke et al. «Automatic differentiation in PyTorch». In: 2017.

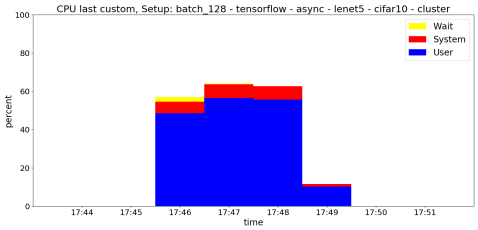
[60] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC].

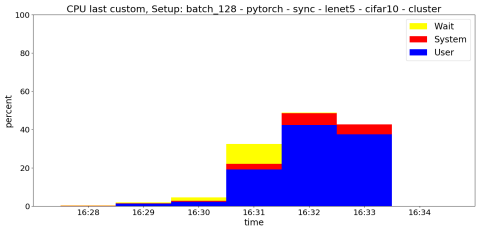
Παραρτήματα

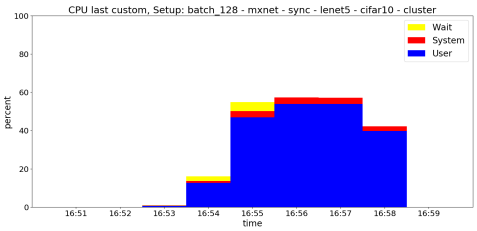
Παράρτημα Ι

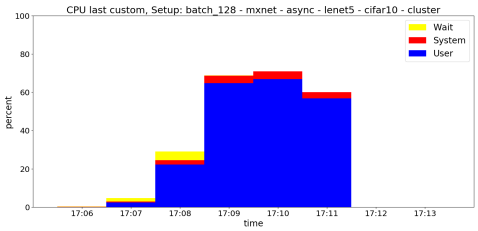
Διαγράμματα Ganglia κατανάλωσης CPU

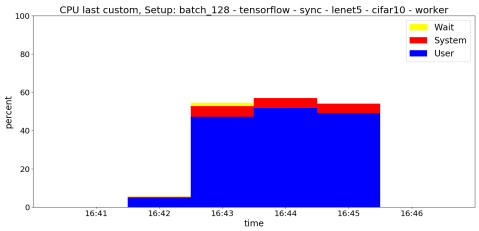


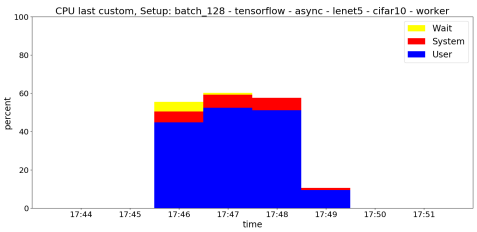


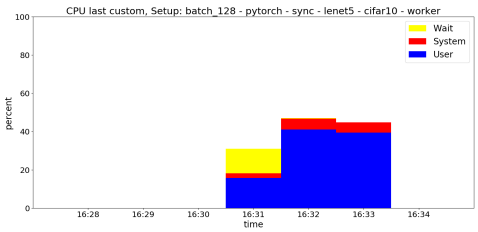


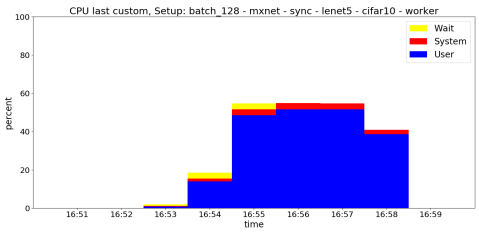


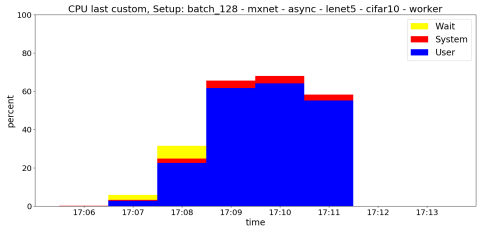


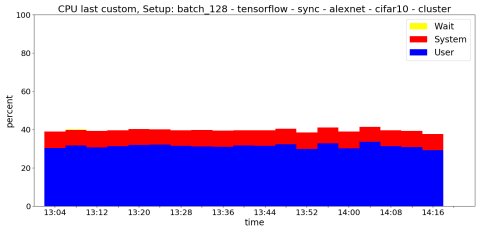


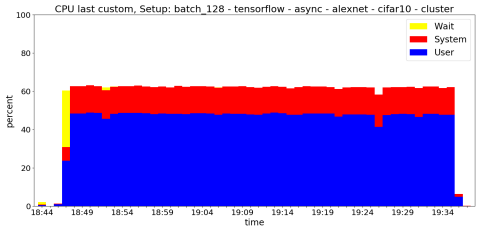


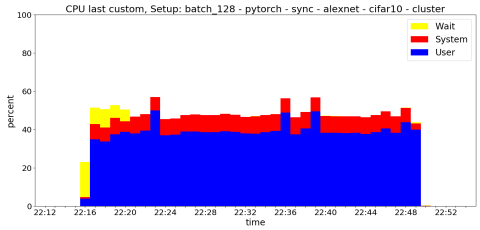


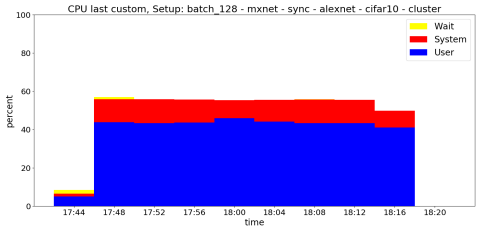


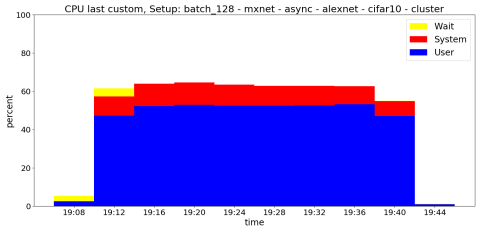

CPU last custom, Setup: batch_128 - mxnet - async - lenet5 - cifar10 - worker
Wait
System
User
percent
time

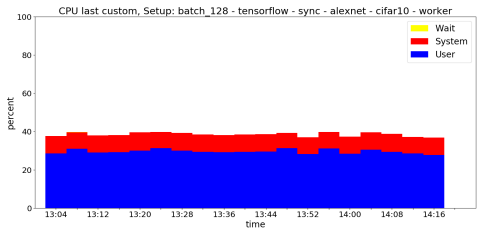

CPU last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - cluster
Wait
System
User
percent
time

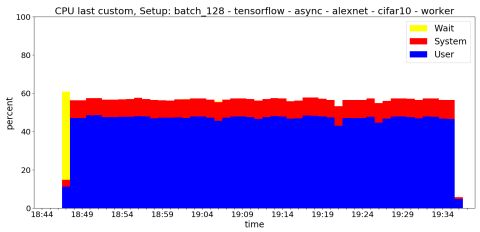

CPU last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - cluster
Wait
System
User
percent
time

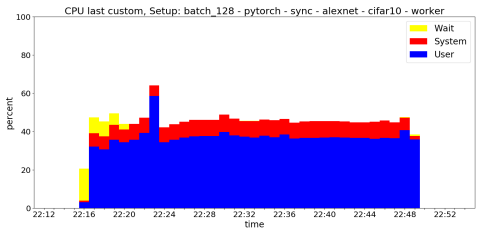

CPU last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - cluster
Wait
System
User
percent
time

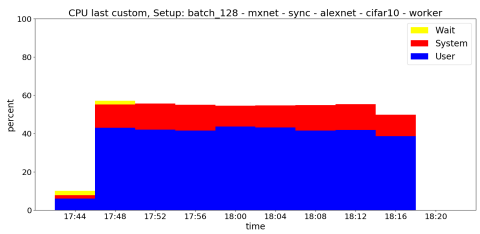

CPU last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - cluster
Wait
System
User
percent
time

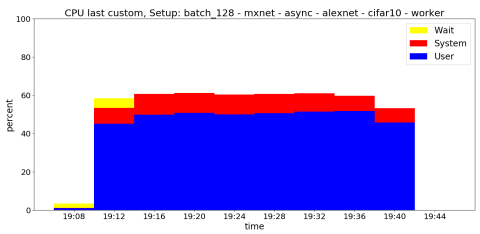

CPU last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - cluster
Wait
System
User
percent
time

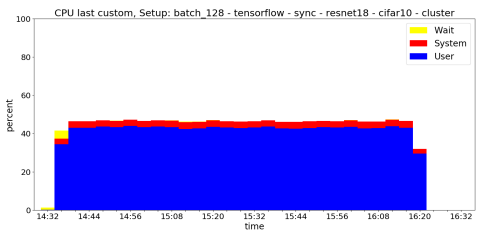

CPU last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - worker
Wait
System
User
percent
time

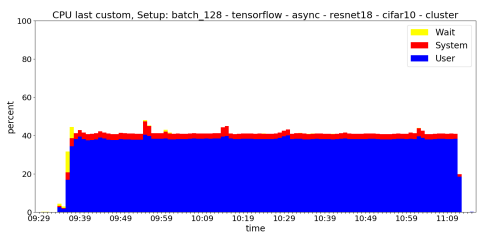

CPU last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - worker
Wait
System
User
percent
time

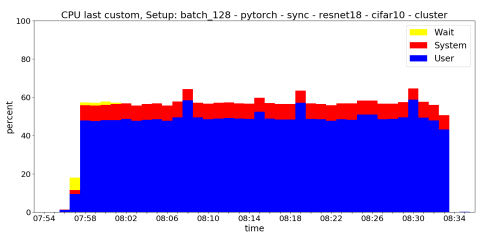

CPU last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - worker
Wait
System
User
percent
time

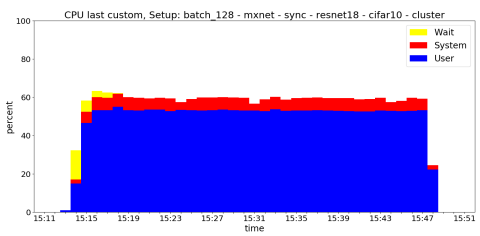

CPU last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - worker
Wait
System
User
percent
time

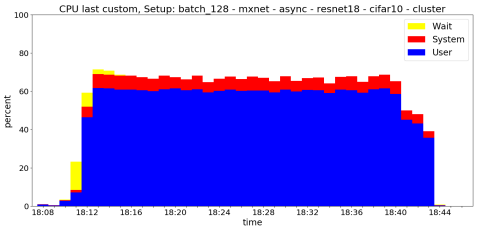

CPU last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - worker
Wait
System
User
percent
time

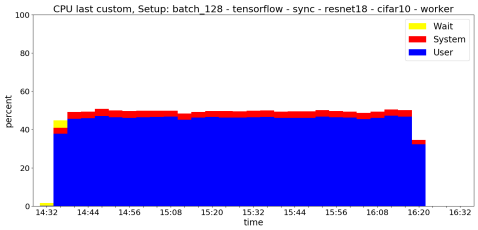

CPU last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - cluster
Wait
System
User
percent
time

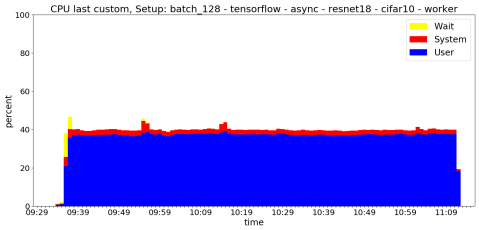

CPU last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - cluster
Wait
System
User
percent
time

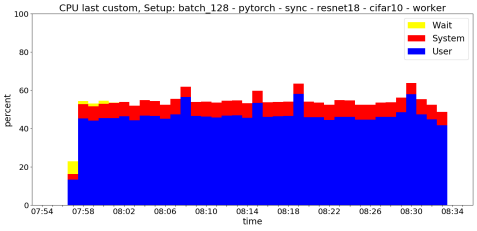

CPU last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - cluster
Wait
System
User
percent
time

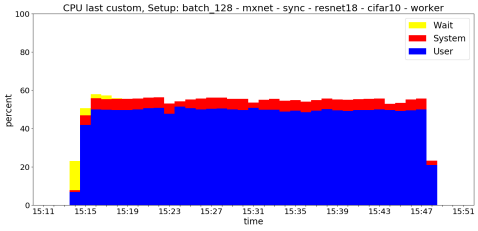

CPU last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - cluster
Wait
System
User
percent
time

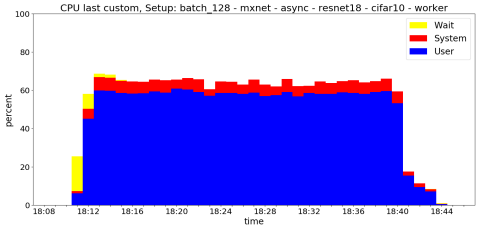

CPU last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - cluster
Wait
System
User
percent
time

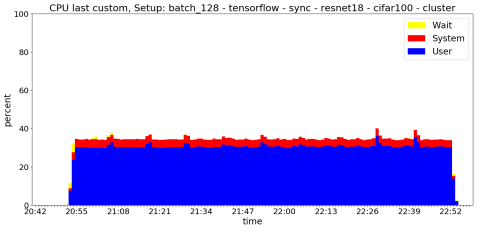

CPU last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - worker
Wait
System
User
percent
time

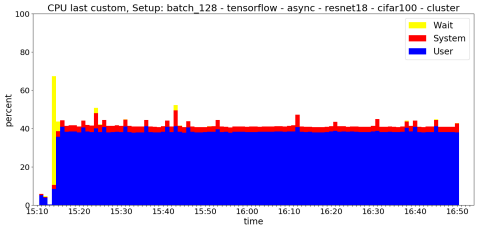

CPU last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - worker
Wait
System
User
percent
time

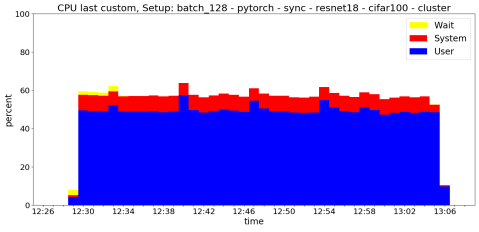

CPU last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - worker
Wait
System
User
percent
time

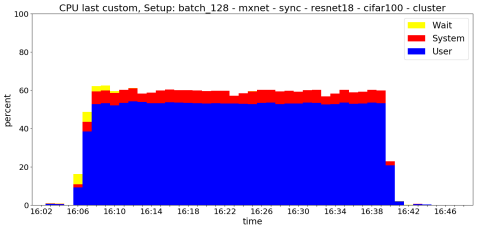

CPU last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - worker
Wait
System
User
percent
time

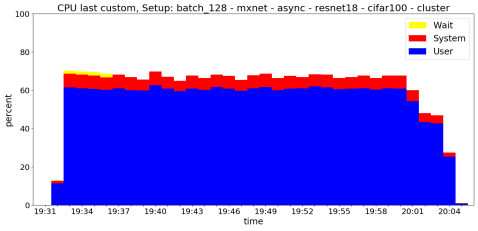

CPU last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - worker
Wait
System
User
percent
time

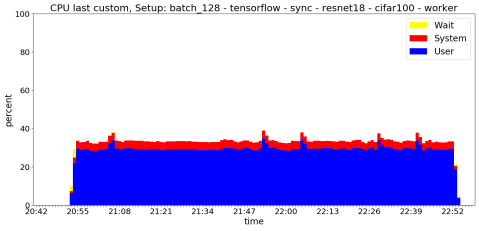

CPU last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - cluster
Wait
System
User
percent
time

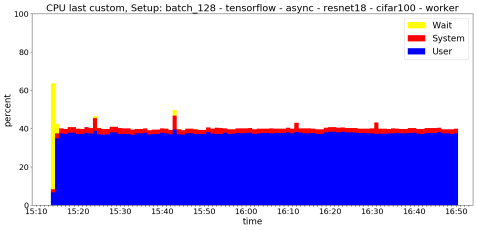

CPU last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - cluster
Wait
System
User
percent
time

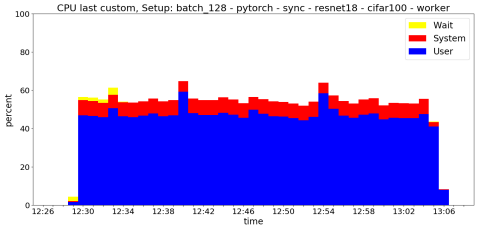

CPU last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - cluster
Wait
System
User
percent
time

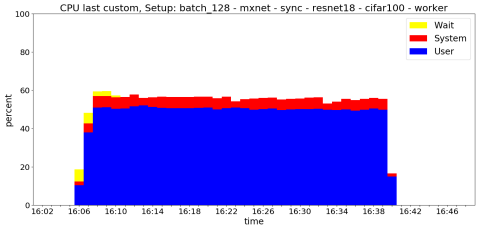

CPU last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - cluster
Wait
System
User
percent
time

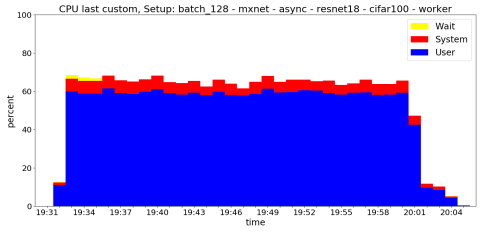

CPU last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - cluster
Wait
System
User
percent
time

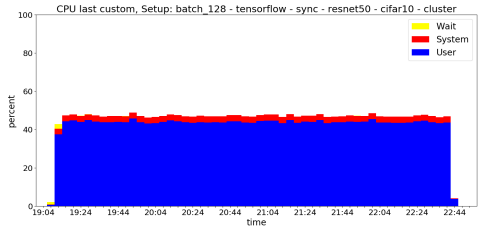

CPU last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - worker
Wait
System
User
percent
time

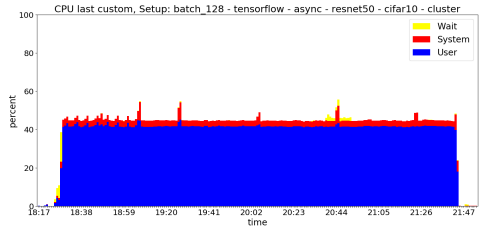

CPU last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - worker
Wait
System
User
percent
time

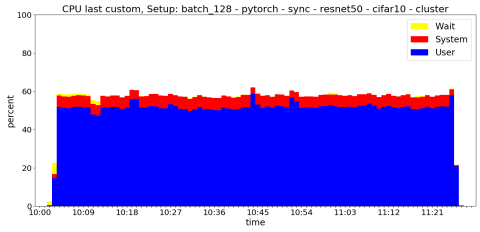

CPU last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - worker
Wait
System
User
percent
time

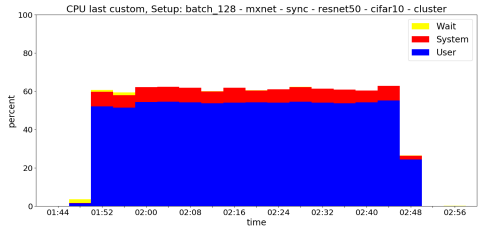

CPU last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - worker
Wait
System
User
percent
time

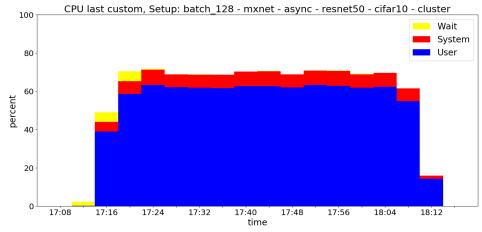

CPU last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - worker
Wait
System
User
percent
time

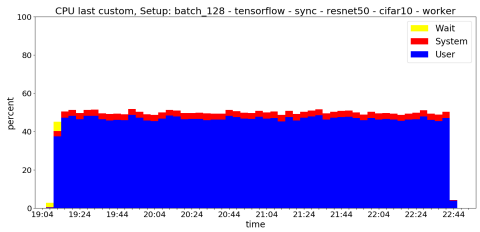

CPU last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - cluster
Wait
System
User
percent
time

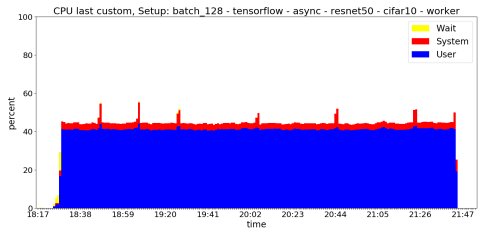

CPU last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - cluster
Wait
System
User
percent
time

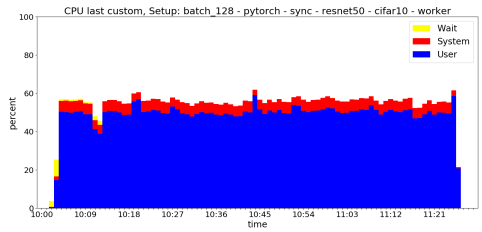

CPU last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - cluster
Wait
System
User
percent
time

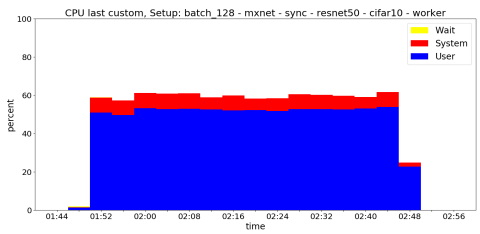

CPU last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - cluster
Wait
System
User
percent
time

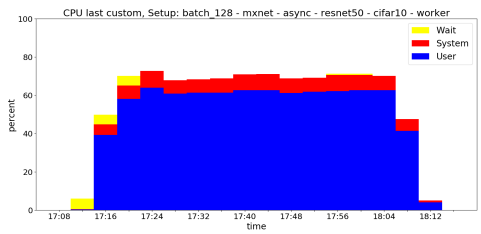

CPU last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - cluster
Wait
System
User
percent
time

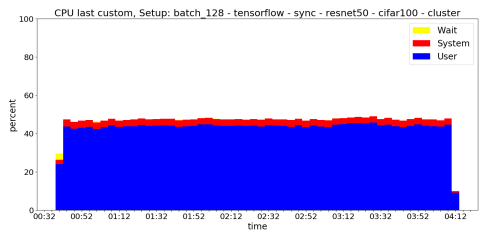

CPU last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - worker
Wait
System
User
percent
time

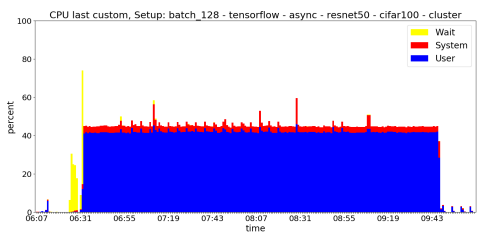

CPU last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - worker
Wait
System
User
percent
time

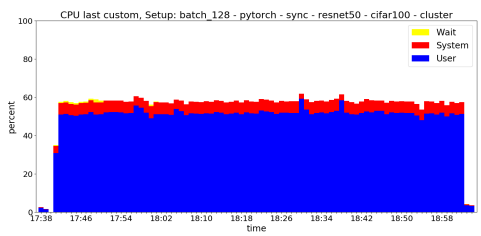

CPU last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - worker
Wait
System
User
percent
time

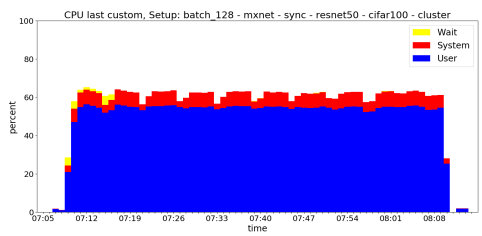

CPU last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - worker
Wait
System
User
percent
time

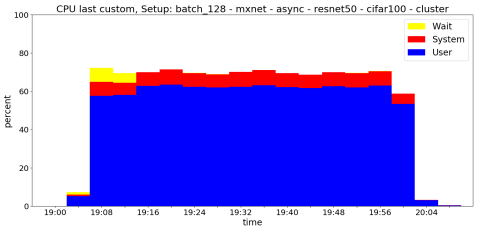

CPU last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - worker
Wait
System
User
percent
time

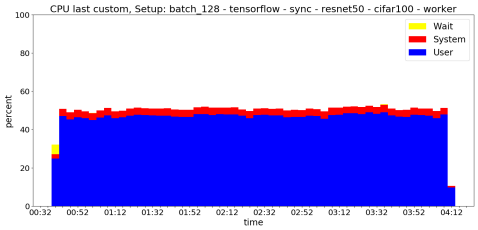

CPU last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - cluster
Wait
System
User
percent
time

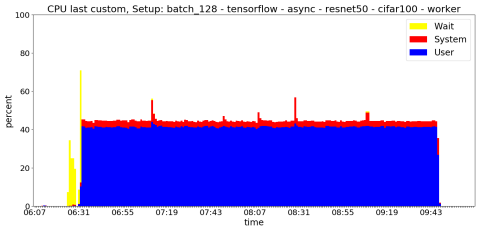

CPU last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - cluster
Wait
System
User
percent
time

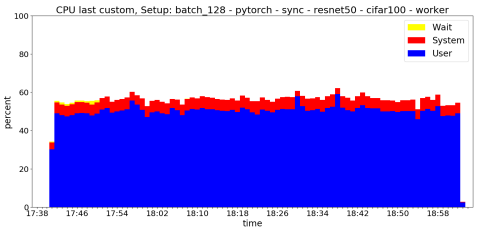

CPU last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - cluster
Wait
System
User
percent
time

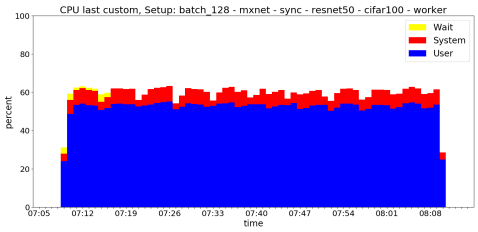

CPU last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - cluster
Wait
System
User
percent
time

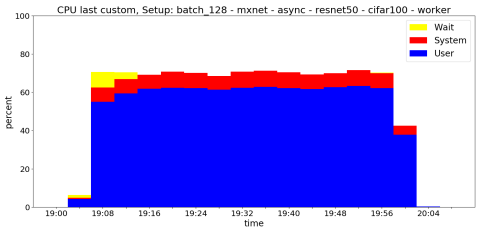

CPU last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - cluster
Wait
System
User
percent
time
19:00
19:08
19:16
19:24
19:32
19:40
19:48
19:56
20:04


CPU last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - worker
Wait
System
User
percent
time
00:32
00:52
01:12
01:32
01:52
02:12
02:32
02:52
03:12
03:32
03:52
04:12


CPU last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - worker
Wait
System
User
percent
time
06:07
06:31
06:55
07:19
07:43
08:07
08:31
08:55
09:19
09:43

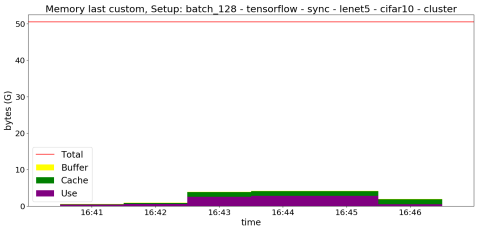

CPU last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - worker
Wait
System
User
percent
time
17:38
17:46
17:54
18:02
18:10
18:18
18:26
18:34
18:42
18:50
18:58

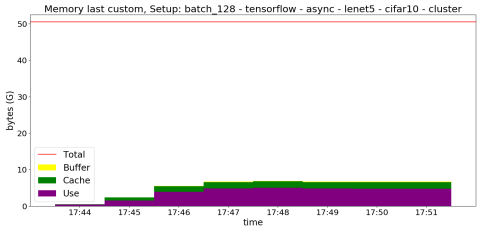

CPU last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - worker
Wait
System
User
percent
time
07:05
07:12
07:19
07:26
07:33
07:40
07:47
07:54
08:01
08:08

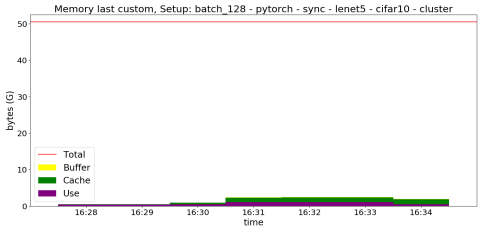

CPU last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - worker
Wait
System
User
percent
time
19:00
19:08
19:16
19:24
19:32
19:40
19:48
19:56
20:04

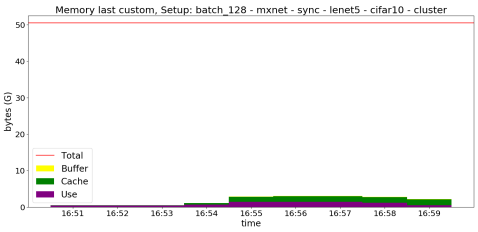
Παράρτημα ΙΙ

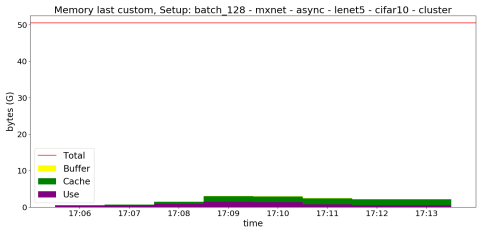
Διαγράμματα Ganglia κατανάλωσης Μνήμης

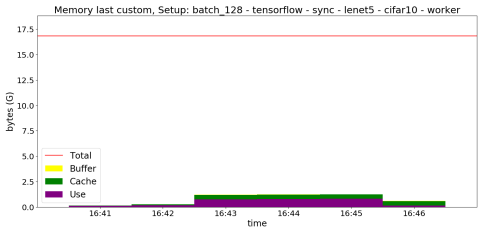


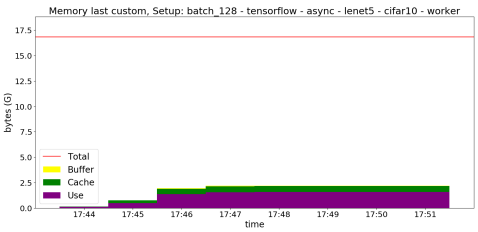


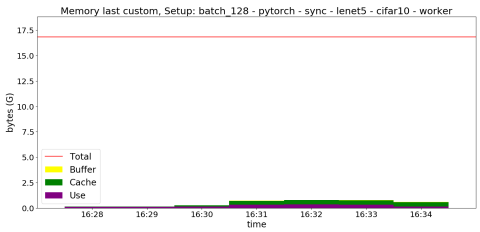


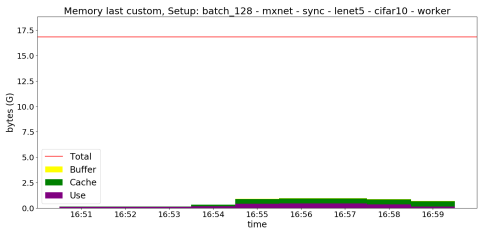


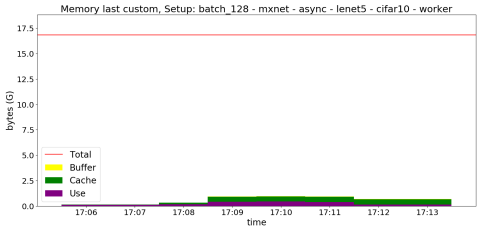


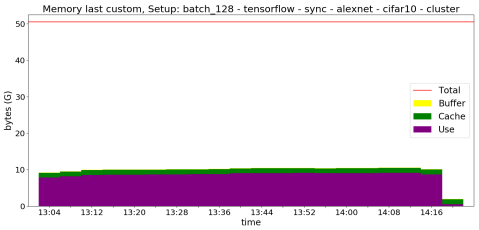


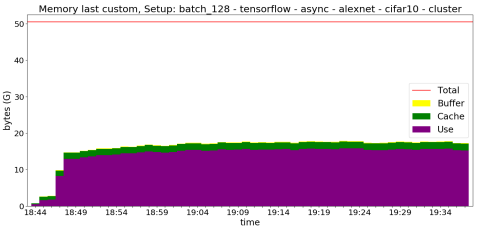


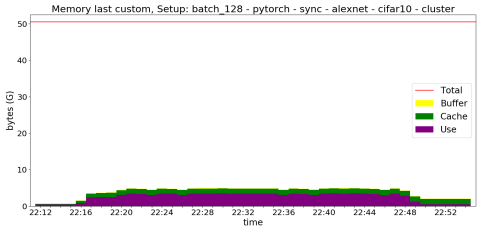


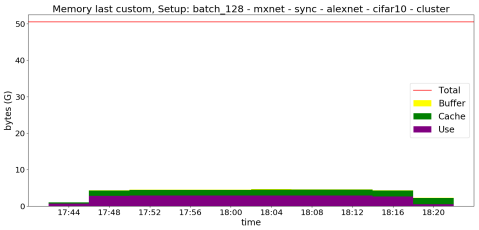


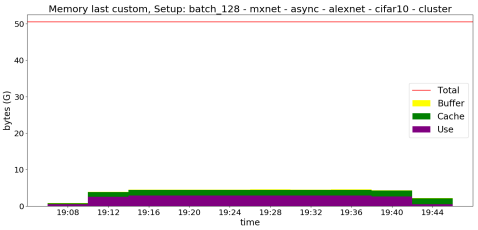

Memory last custom, Setup: batch_128 - mxnet - async - lenet5 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

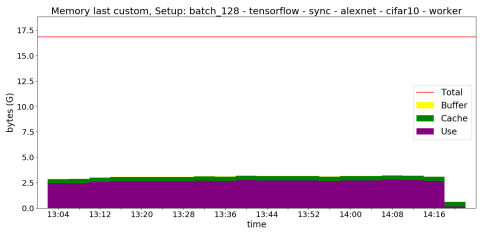

Memory last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

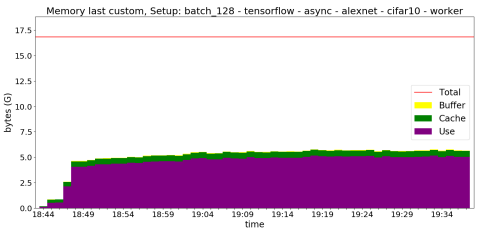

Memory last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

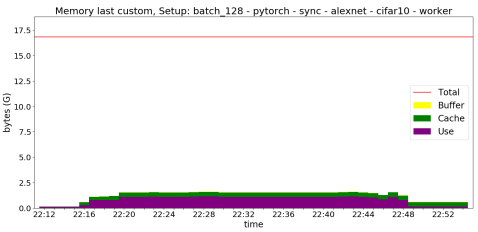

Memory last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

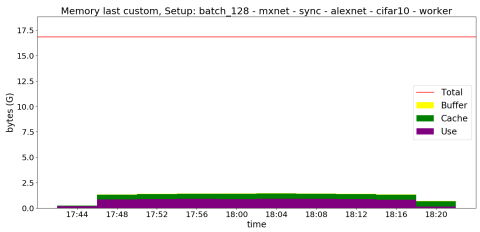

Memory last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

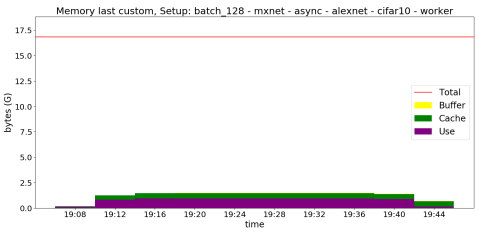

Memory last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

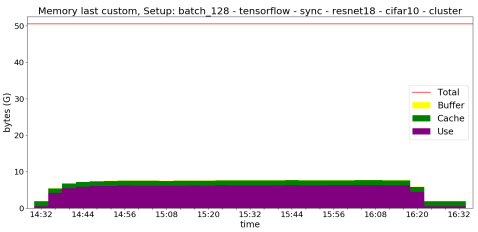

Memory last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

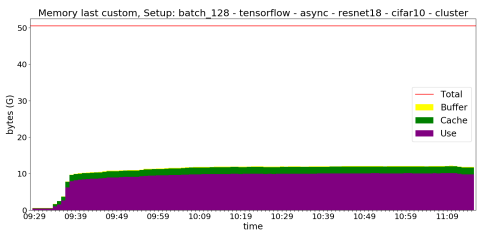

Memory last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

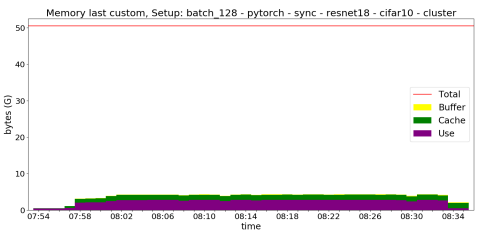

Memory last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

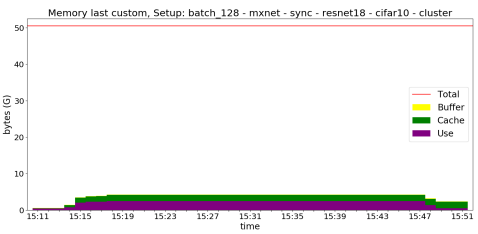

Memory last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

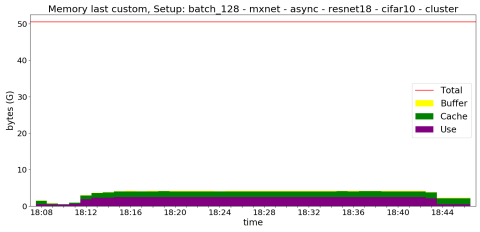

Memory last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

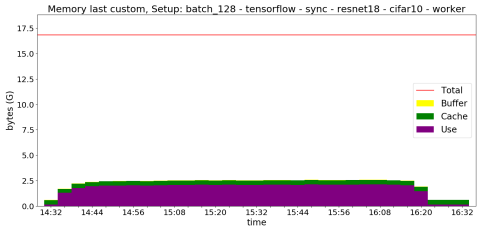

Memory last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

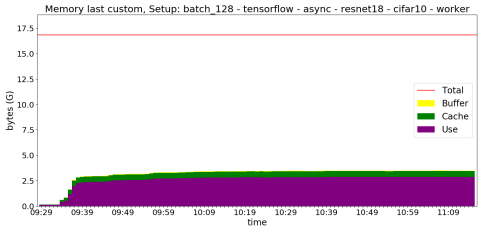

Memory last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

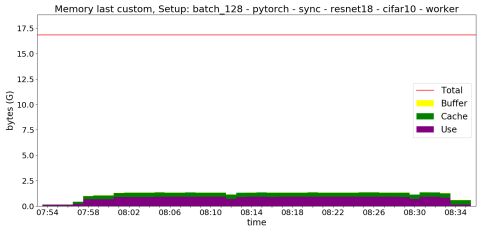

Memory last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

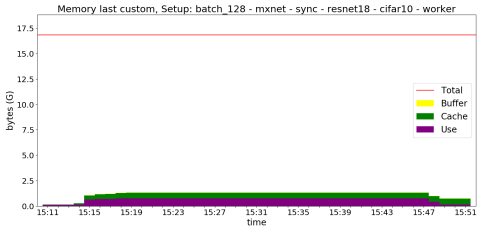

Memory last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

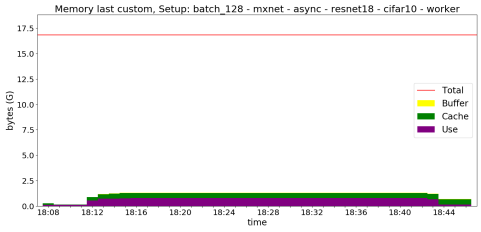

Memory last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

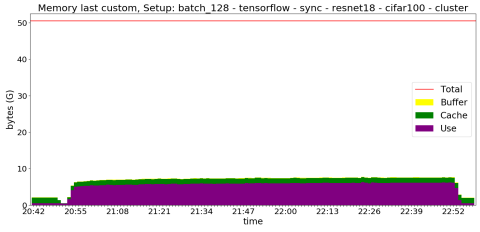

Memory last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

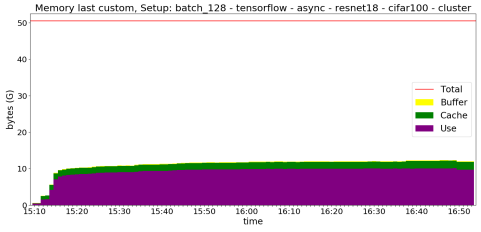

Memory last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

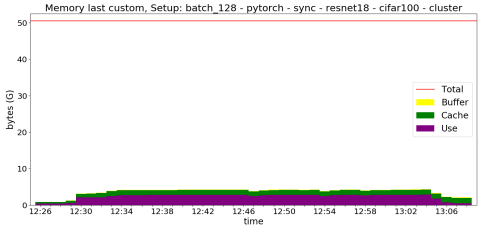

Memory last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

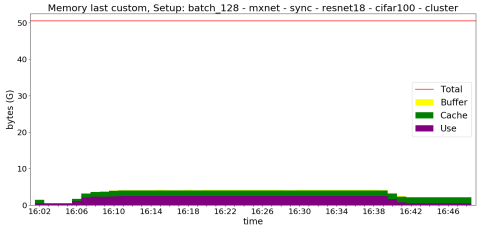

Memory last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

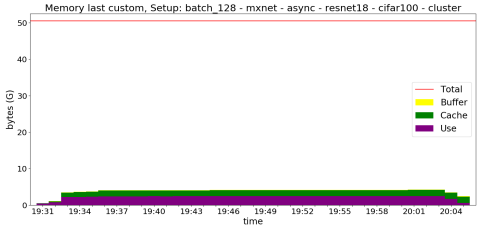

Memory last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

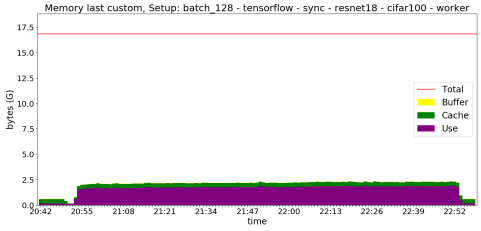

Memory last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

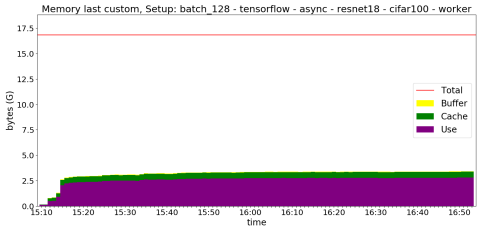

Memory last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

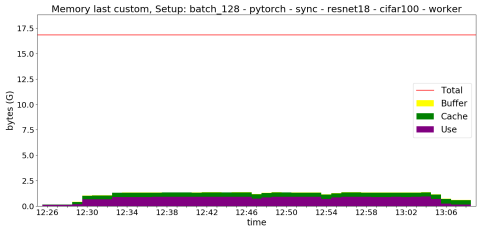

Memory last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

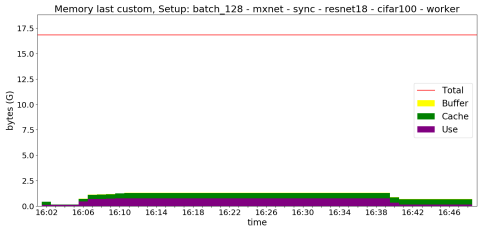

Memory last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

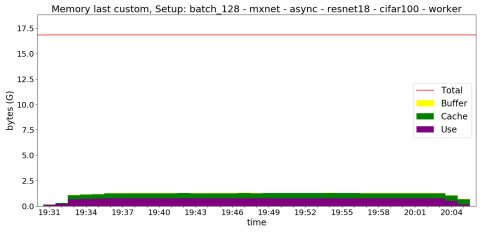

Memory last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

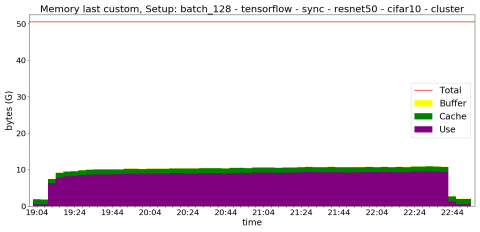

Memory last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

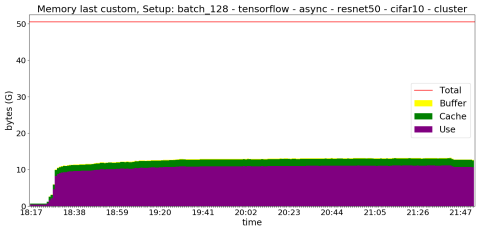

Memory last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

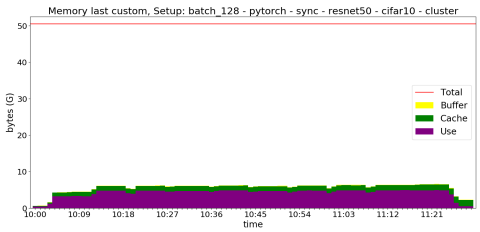

Memory last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

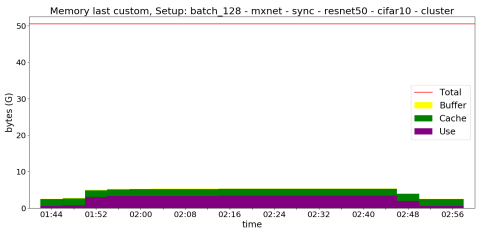

Memory last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

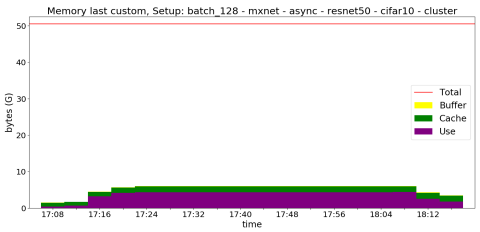

Memory last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

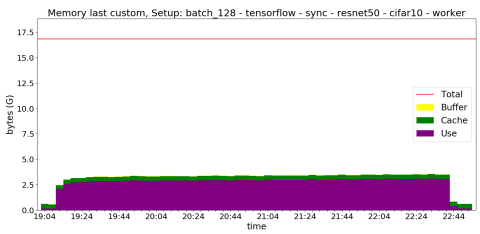

Memory last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

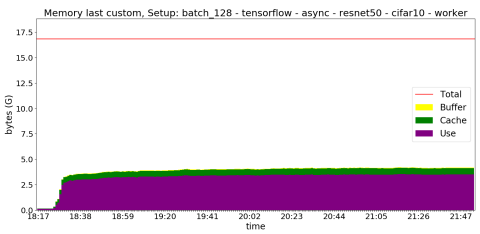

Memory last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

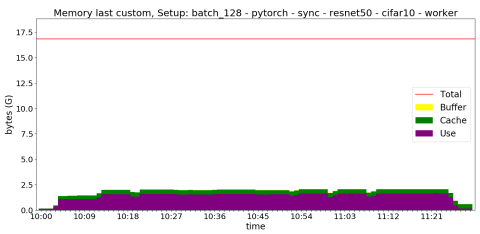

Memory last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

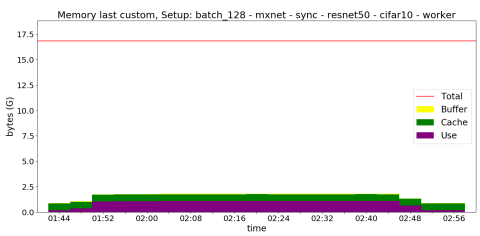

Memory last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

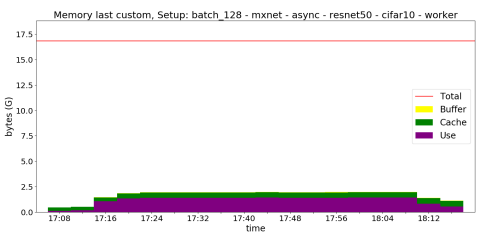

Memory last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

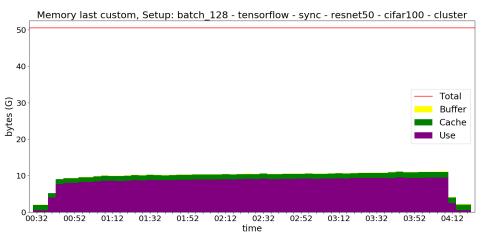

Memory last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

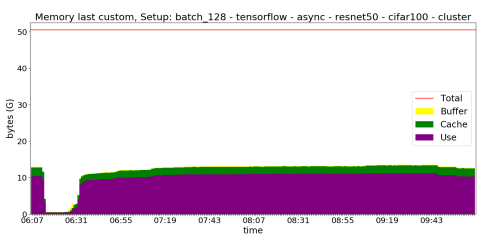

Memory last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

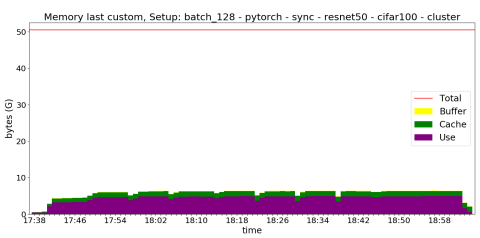

Memory last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

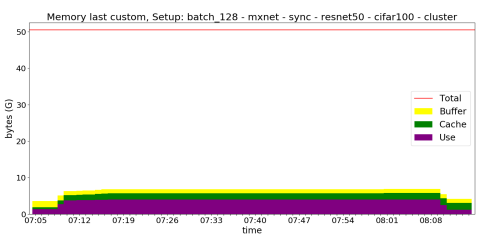

Memory last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

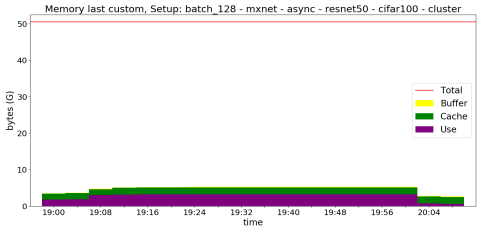

Memory last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - worker
bytes (G)
time
Total
Buffer
Cache
Use

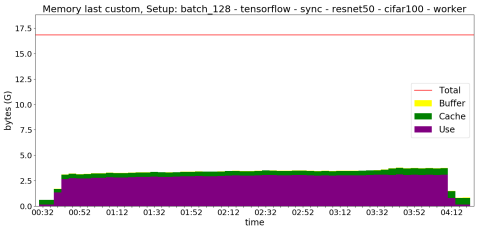

Memory last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

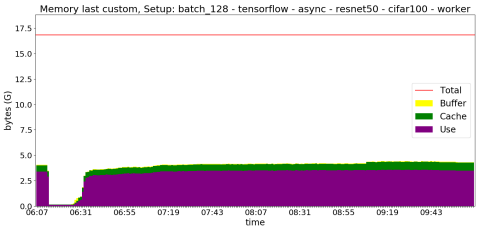

Memory last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

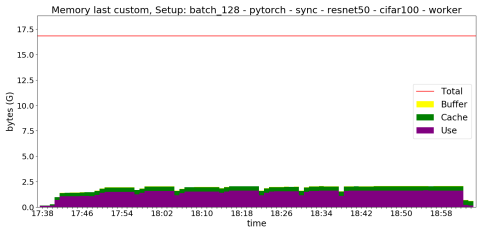

Memory last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

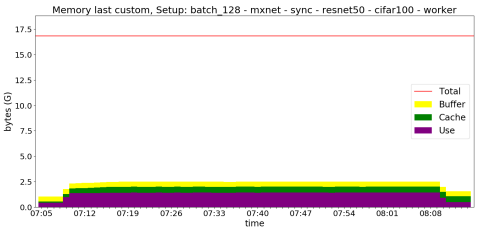

Memory last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use

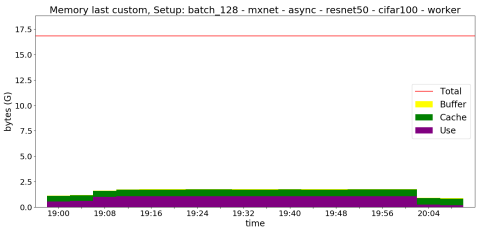

Memory last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - cluster
bytes (G)
time
Total
Buffer
Cache
Use


Memory last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use


Memory last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

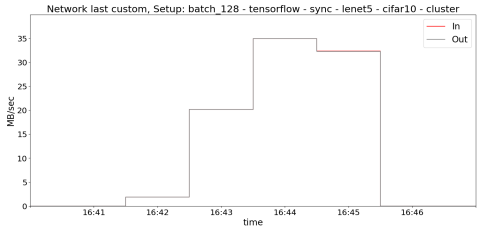

Memory last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

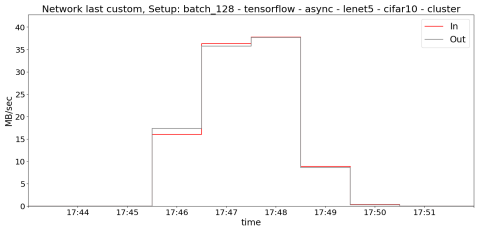

Memory last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

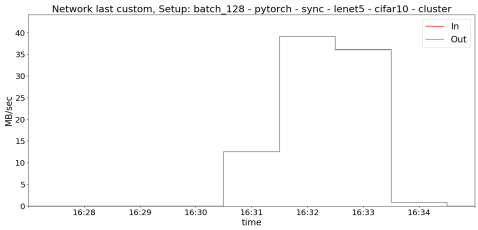

Memory last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - worker
bytes (G)
time
Total
Buffer
Cache
Use

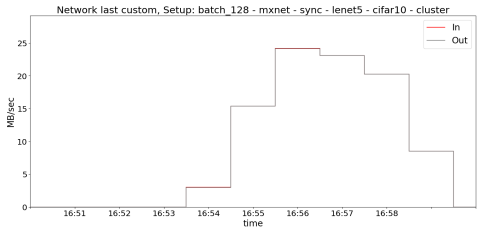
Παράρτημα III

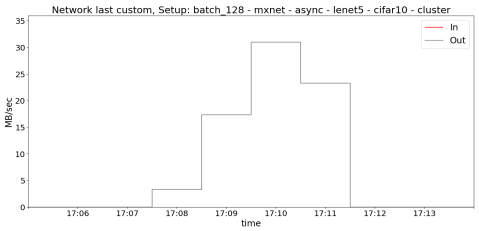
Διαγράμματα Ganglia κατανάλωσης Δικτύου

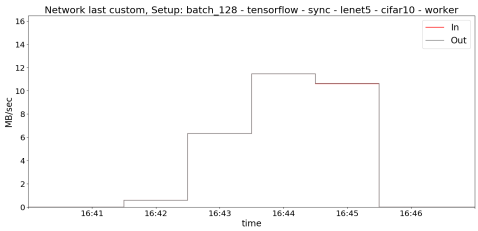


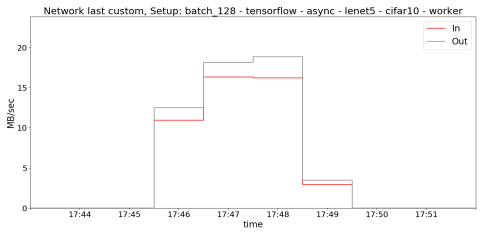


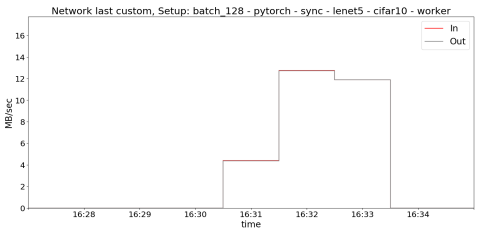


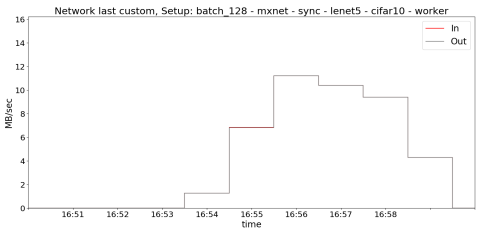


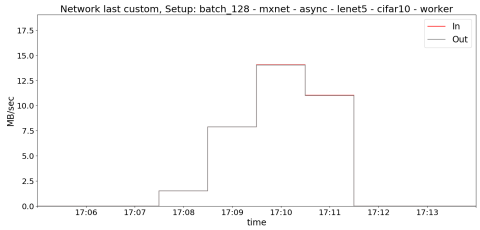


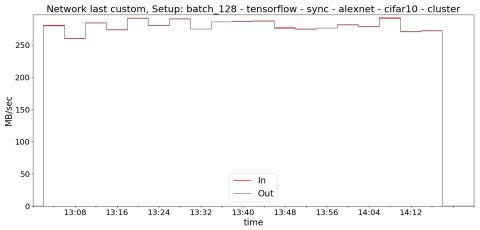


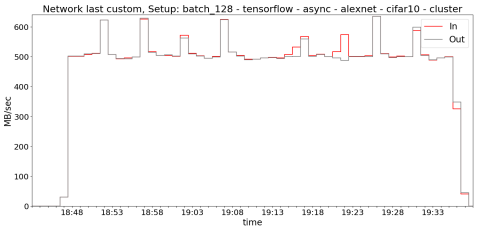


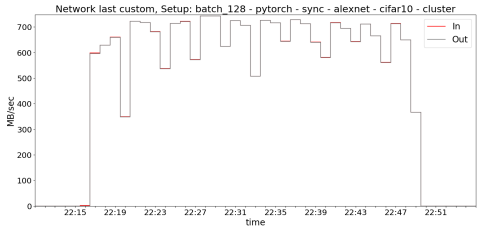


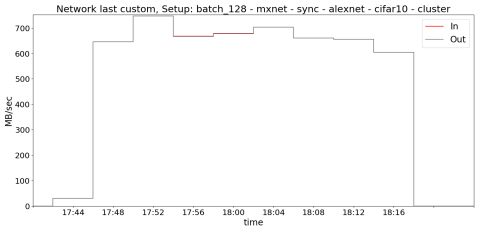


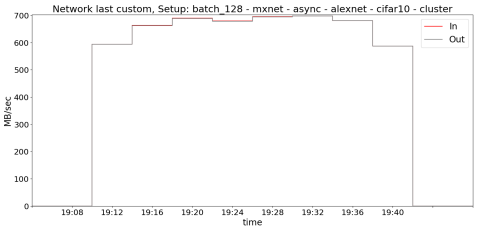

Network last custom, Setup: batch_128 - mxnet - async - lenet5 - cifar10 - worker
In
Out
MB/sec
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
17:06
17:07
17:08
17:09
17:10
17:11
17:12
17:13
time

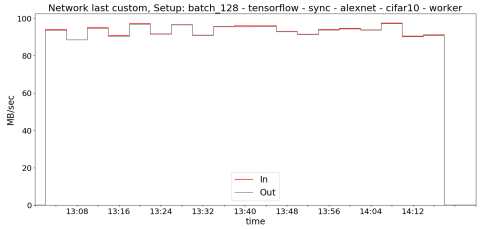

Network last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - cluster
In
Out
MB/sec
0
50
100
150
200
250
13:08
13:16
13:24
13:32
13:40
13:48
13:56
14:04
14:12
time

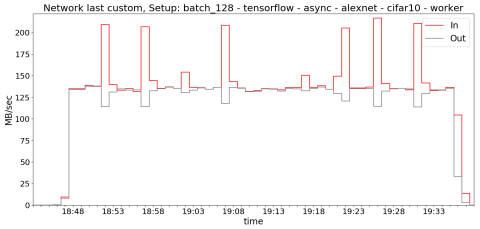

Network last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - cluster
In
Out
MB/sec
0
100
200
300
400
500
600
18:48
18:53
18:58
19:03
19:08
19:13
19:18
19:23
19:28
19:33
time

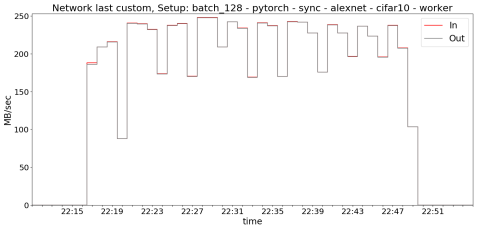

Network last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - cluster
In
Out
MB/sec
0
100
200
300
400
500
600
700
22:15
22:19
22:23
22:27
22:31
22:35
22:39
22:43
22:47
22:51
time

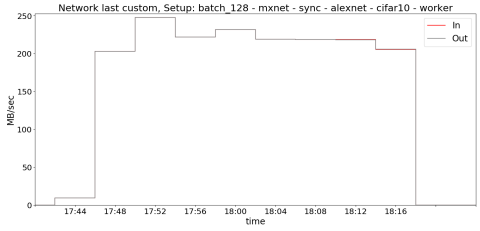

Network last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - cluster
In
Out
MB/sec
0
100
200
300
400
500
600
700
17:44
17:48
17:52
17:56
18:00
18:04
18:08
18:12
18:16
time

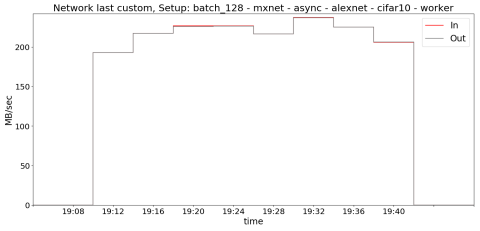

Network last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - cluster
In
Out
MB/sec
0
100
200
300
400
500
600
700
19:08
19:12
19:16
19:20
19:24
19:28
19:32
19:36
19:40
time

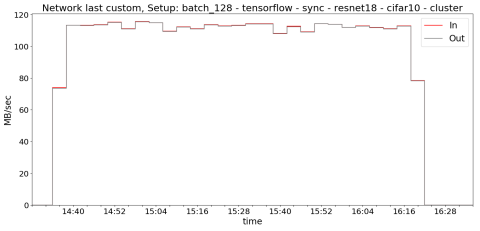

Network last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - worker
In
Out
MB/sec
0
20
40
60
80
100
13:08
13:16
13:24
13:32
13:40
13:48
13:56
14:04
14:12
time

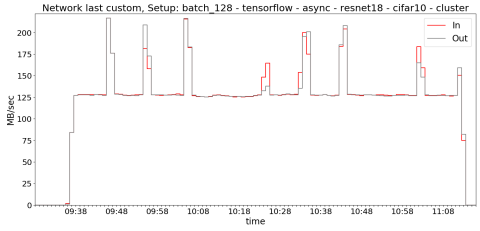

Network last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - worker
In
Out
MB/sec
0
25
50
75
100
125
150
175
200
18:48
18:53
18:58
19:03
19:08
19:13
19:18
19:23
19:28
19:33
time

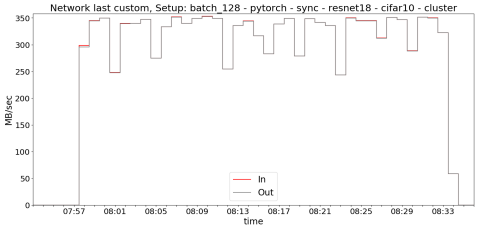

Network last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - worker
In
Out
MB/sec
0
50
100
150
200
250
22:15
22:19
22:23
22:27
22:31
22:35
22:39
22:43
22:47
22:51
time

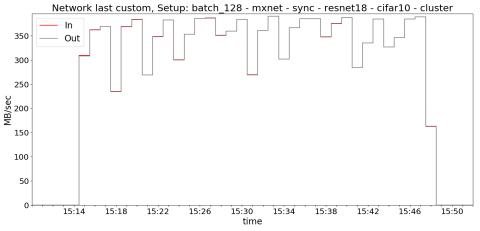

Network last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - worker
In
Out
MB/sec
0
50
100
150
200
250
17:44
17:48
17:52
17:56
18:00
18:04
18:08
18:12
18:16
time

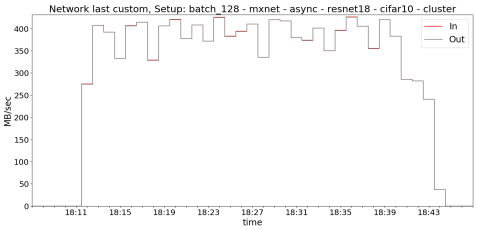

Network last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - worker
In
Out
MB/sec
0
50
100
150
200
19:08
19:12
19:16
19:20
19:24
19:28
19:32
19:36
19:40
time

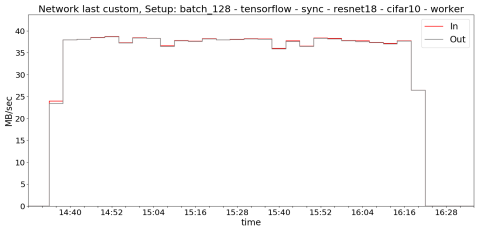

Network last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - cluster
In
Out
MB/sec
0
20
40
60
80
100
120
14:40
14:52
15:04
15:16
15:28
15:40
15:52
16:04
16:16
16:28
time

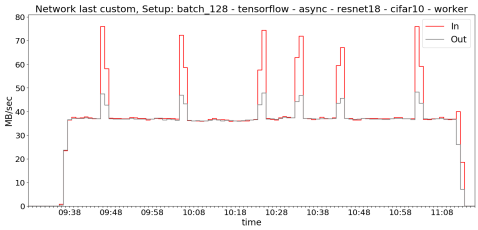

Network last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - cluster
In
Out
MB/sec
0
25
50
75
100
125
150
175
200
09:38
09:48
09:58
10:08
10:18
10:28
10:38
10:48
10:58
11:08
time

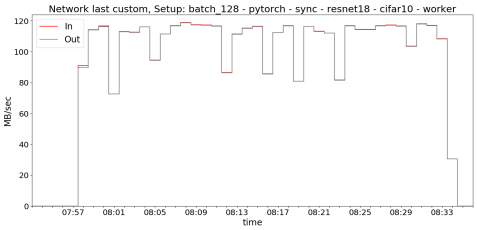

Network last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - cluster
In
Out
MB/sec
0
50
100
150
200
250
300
350
07:57
08:01
08:05
08:09
08:13
08:17
08:21
08:25
08:29
08:33
time

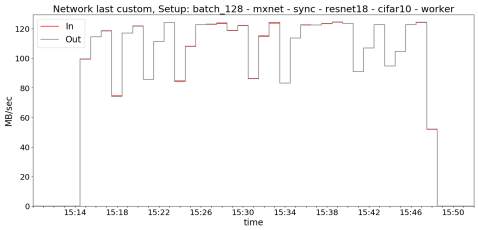

Network last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - cluster
In
Out
MB/sec
0
50
100
150
200
250
300
350
15:14
15:18
15:22
15:26
15:30
15:34
15:38
15:42
15:46
15:50
time

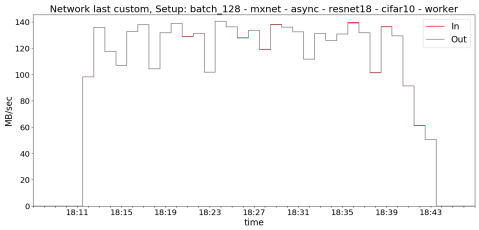

Network last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - cluster
In
Out
MB/sec
time

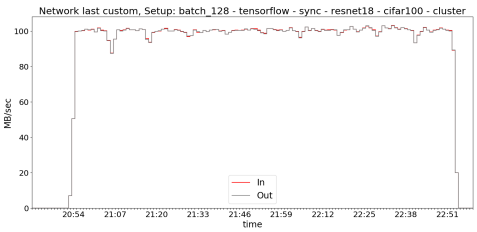

Network last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - worker
In
Out
MB/sec
time

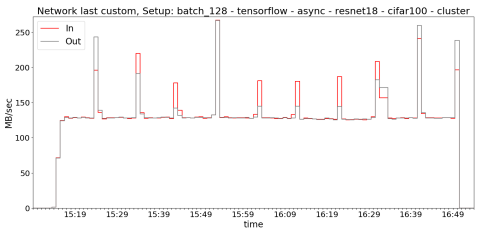

Network last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - worker
In
Out
MB/sec
time

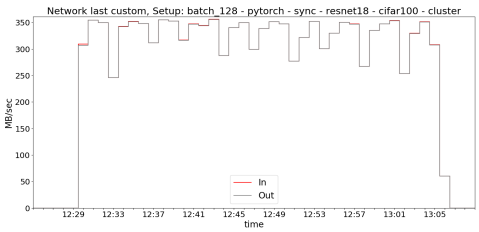

Network last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - worker
In
Out
MB/sec
time

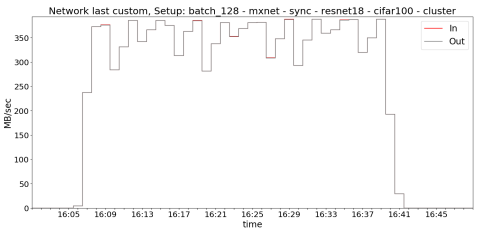

Network last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - worker
In
Out
MB/sec
time

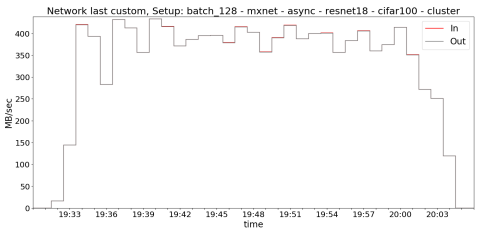

Network last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - worker
In
Out
MB/sec
time

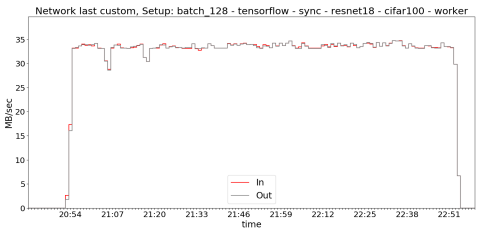

Network last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - cluster
In
Out
MB/sec
time

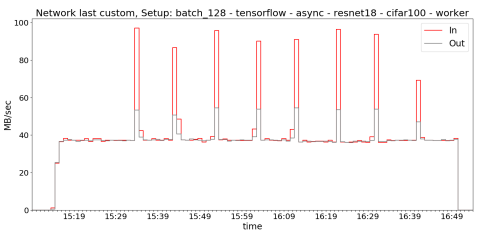

Network last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - cluster
In
Out
MB/sec
time

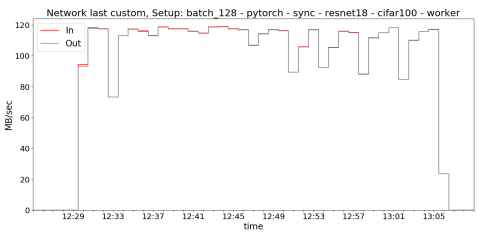

Network last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - cluster
In
Out
MB/sec
time

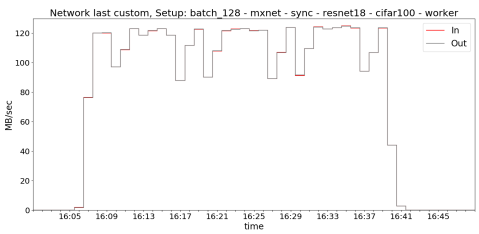

Network last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - cluster
In
Out
MB/sec
time

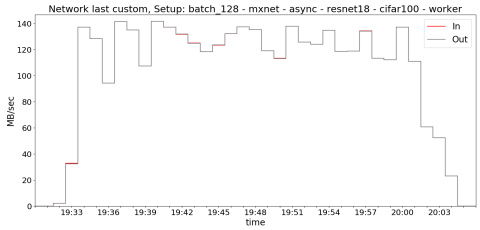

Network last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - cluster
In
Out
MB/sec
time

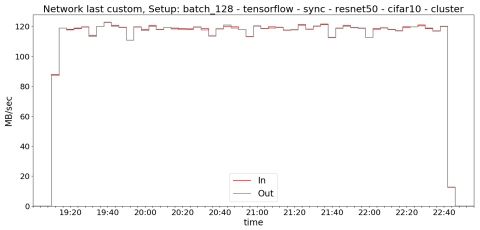

Network last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - worker
In
Out
MB/sec
time

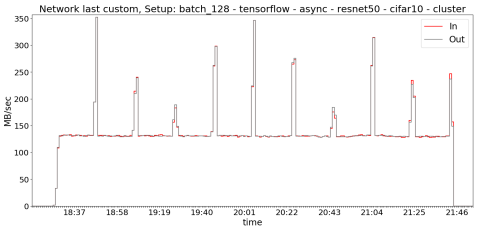

Network last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - worker
In
Out
MB/sec
time

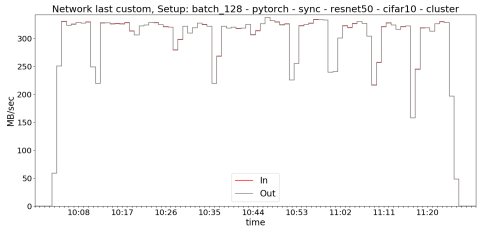

Network last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - worker
In
Out
MB/sec
time

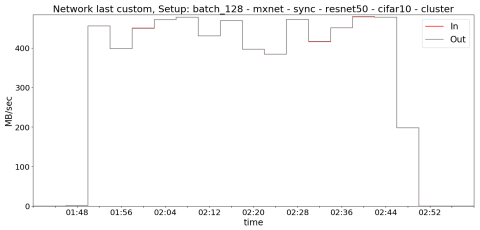

Network last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - worker
In
Out
MB/sec
time

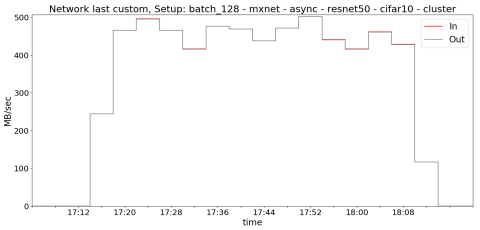

Network last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - worker
MB/sec
time
In
Out

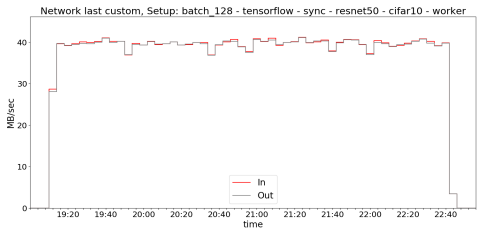

Network last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - cluster
MB/sec
time
In
Out

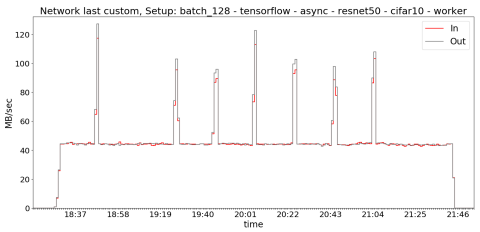

Network last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - cluster
MB/sec
time
In
Out

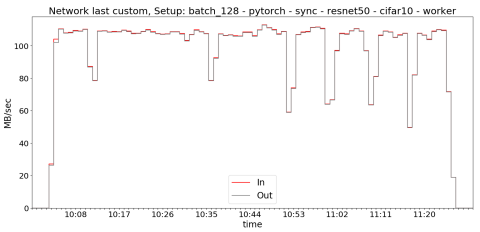

Network last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - cluster
MB/sec
time
In
Out

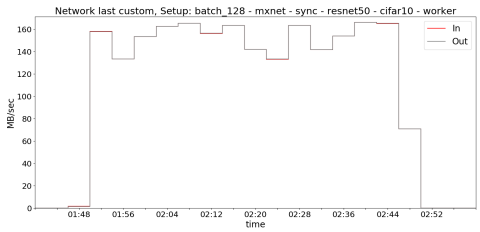

Network last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - cluster
MB/sec
time
In
Out

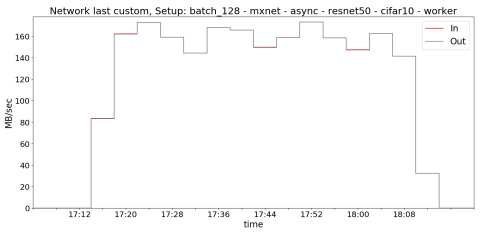

Network last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - cluster
MB/sec
time
In
Out

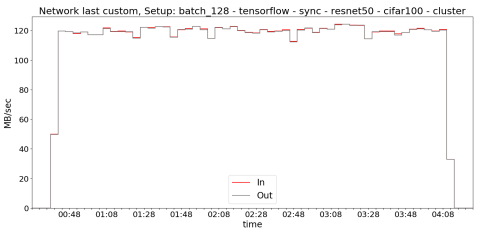

Network last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - worker
MB/sec
time
In
Out

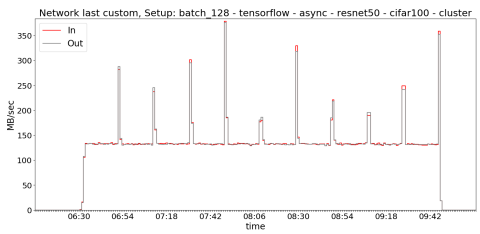

Network last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - worker
MB/sec
time
In
Out

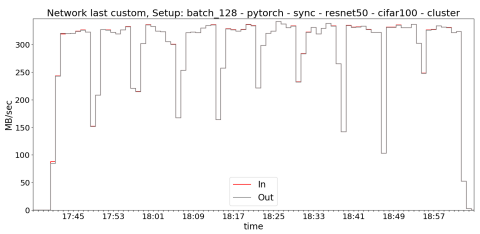

Network last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - worker
MB/sec
time
In
Out

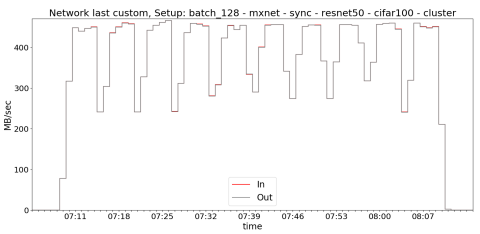

Network last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - worker
MB/sec
time
In
Out

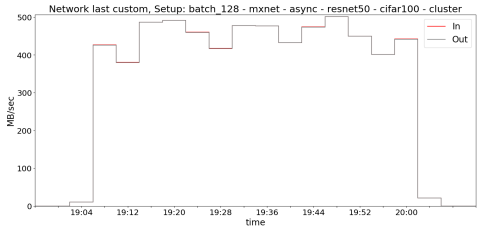

Network last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - worker
MB/sec
time
In
Out

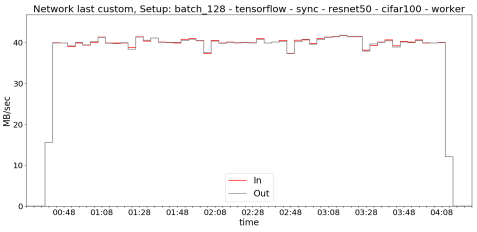

Network last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - cluster
MB/sec
time
In
Out

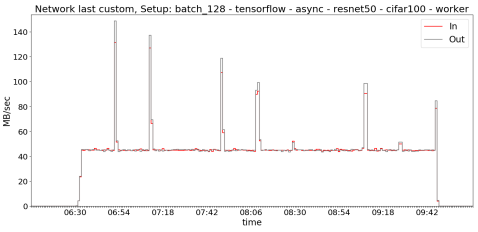

Network last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - cluster
MB/sec
time
In
Out

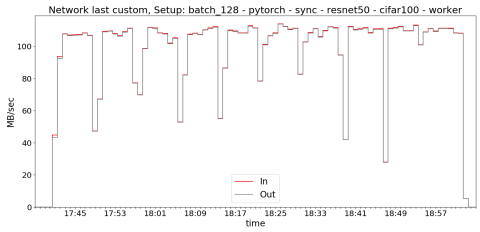

Network last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - cluster
MB/sec
time
In
Out

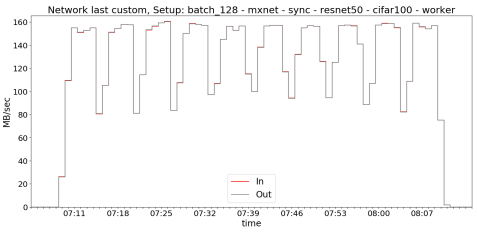

Network last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - cluster
MB/sec
time
In
Out

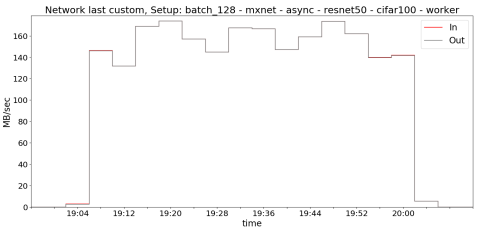

Network last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - cluster
MB/sec
time
In
Out


Network last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - worker
MB/sec
time
In
Out


Network last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - worker
MB/sec
time
In
Out

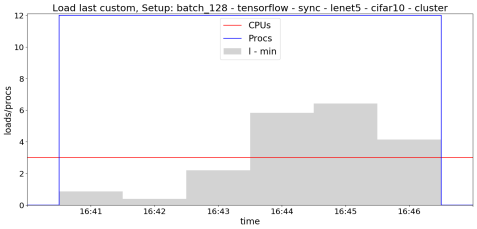

Network last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - worker
MB/sec
time
In
Out

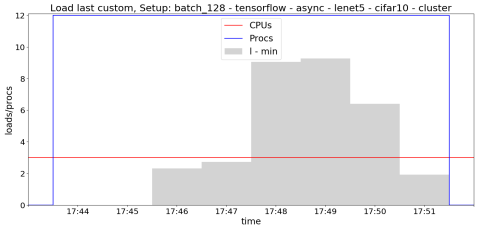

Network last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - worker
MB/sec
time
In
Out

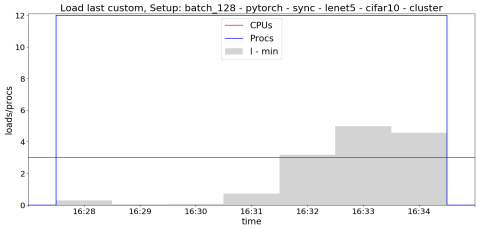

Network last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - worker
MB/sec
time
In
Out

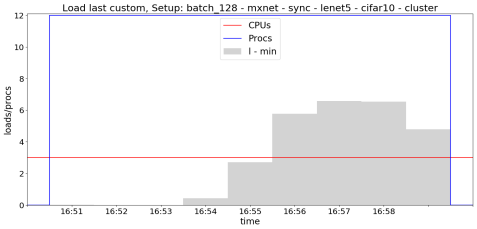
Παράρτημα IV

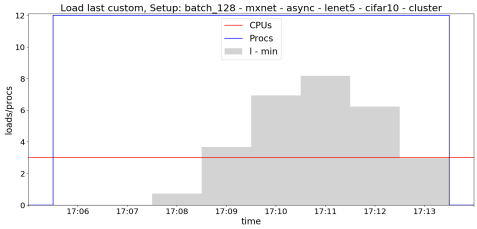
Διαγράμματα Ganglia ανοιχτών διεργασιών

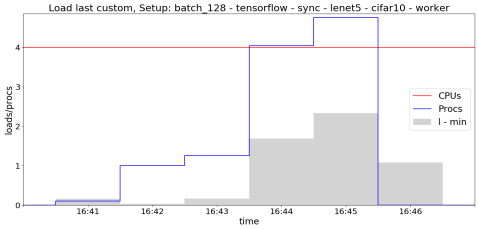


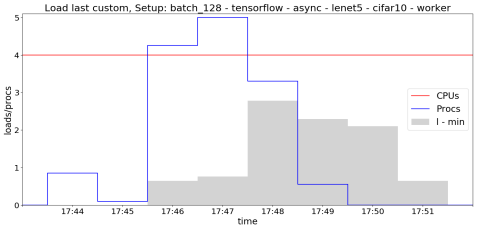


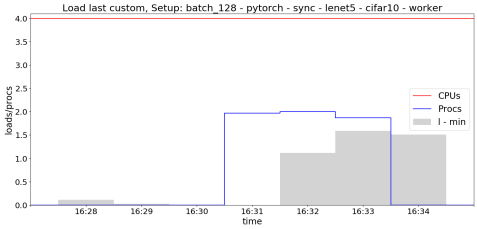


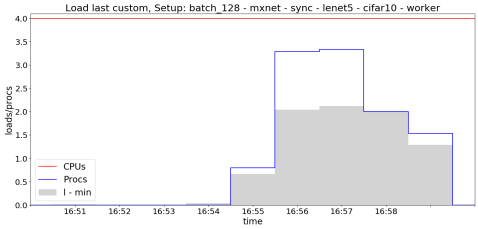


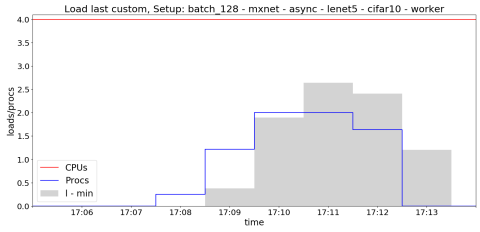


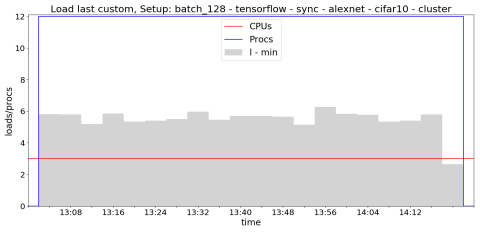


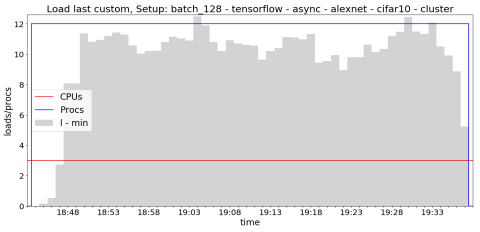


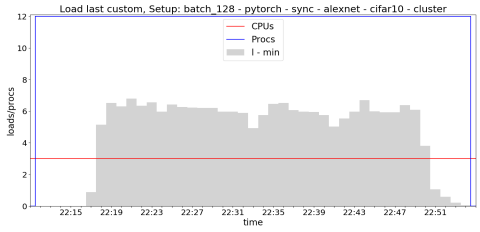


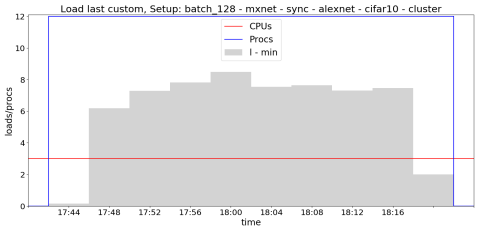


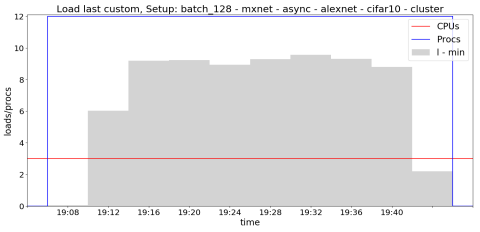

Load last custom, Setup: batch_128 - mxnet - async - lenet5 - cifar10 - worker
loads/procs
time
CPUs
Procs
l - min

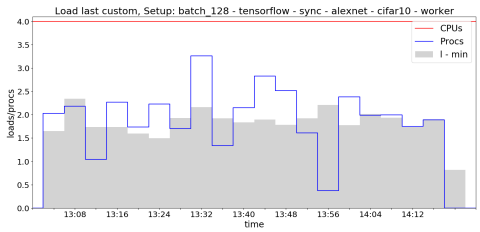

Load last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

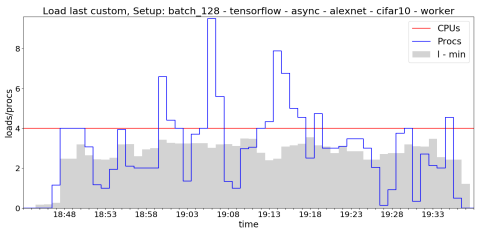

Load last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

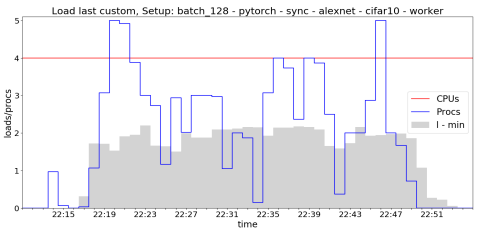

Load last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

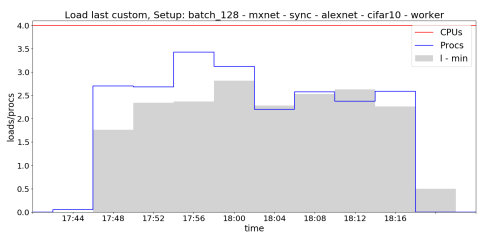

Load last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

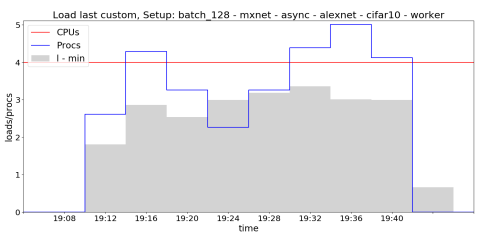

Load last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

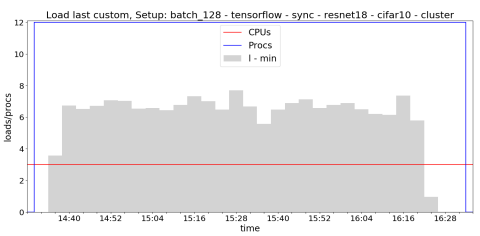

Load last custom, Setup: batch_128 - tensorflow - sync - alexnet - cifar10 - worker
loads/procs
time
CPUs
Procs
l - min

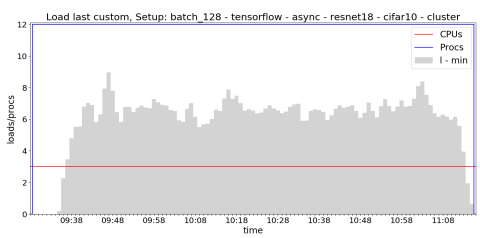

Load last custom, Setup: batch_128 - tensorflow - async - alexnet - cifar10 - worker
loads/procs
time
CPUs
Procs
l - min

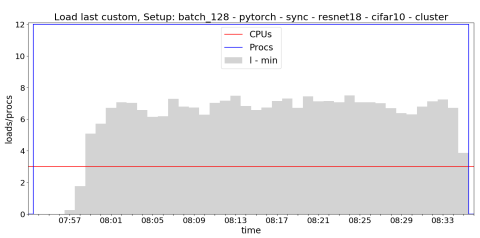

Load last custom, Setup: batch_128 - pytorch - sync - alexnet - cifar10 - worker
loads/procs
time
CPUs
Procs
l - min

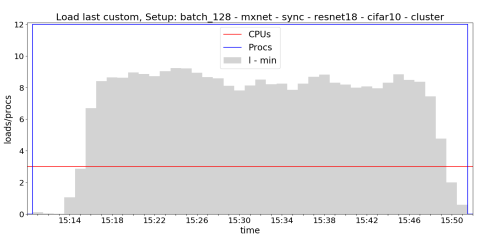

Load last custom, Setup: batch_128 - mxnet - sync - alexnet - cifar10 - worker
loads/procs
time
CPUs
Procs
l - min

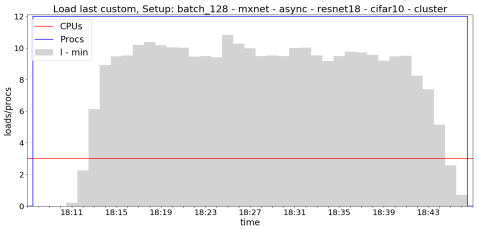

Load last custom, Setup: batch_128 - mxnet - async - alexnet - cifar10 - worker
loads/procs
time
CPUs
Procs
l - min

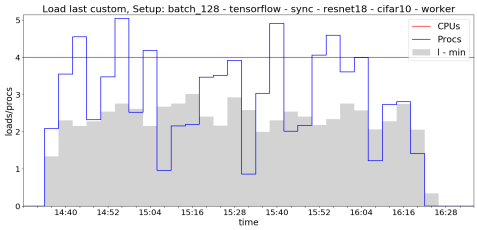

Load last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

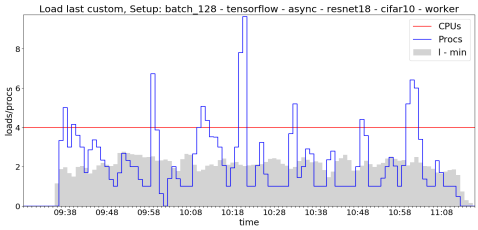

Load last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

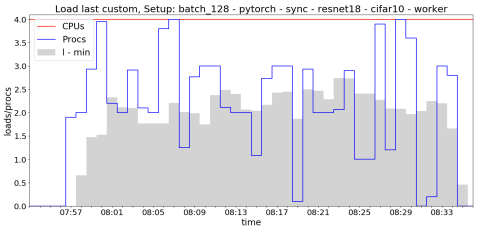

Load last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

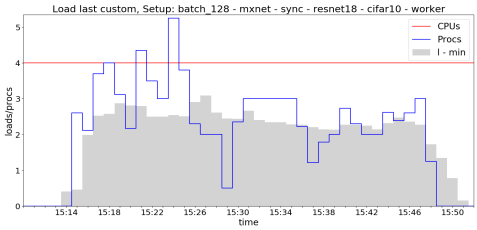

Load last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - cluster
loads/procs
time
CPUs
Procs
l - min

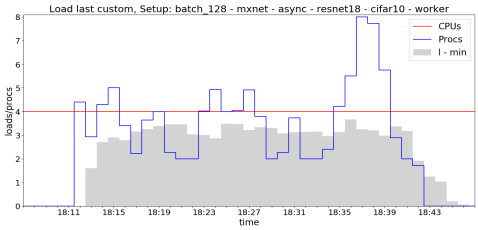

Load last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - cluster
CPUs
Procs
l - min
loads/procs
time

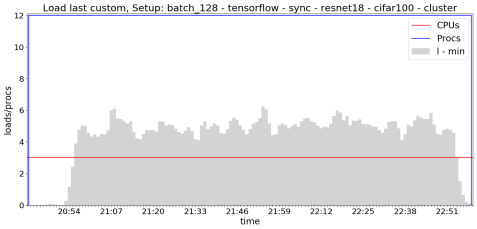

Load last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

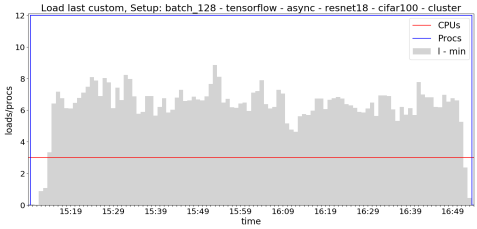

Load last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

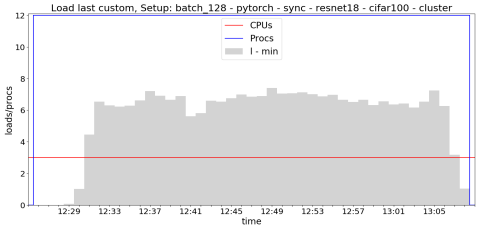

Load last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

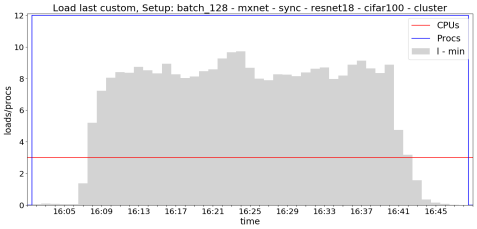

Load last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

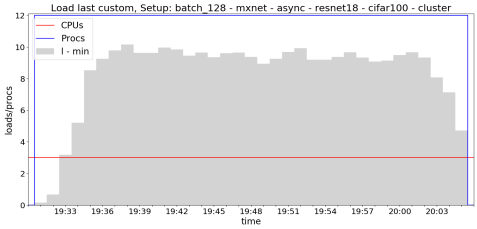

Load last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

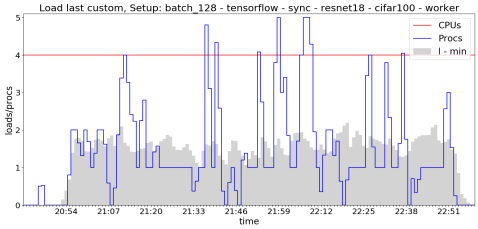

Load last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

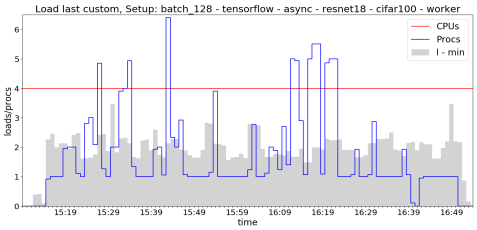

Load last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

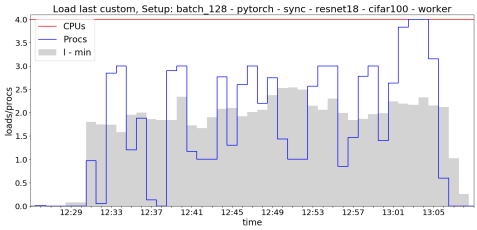

Load last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

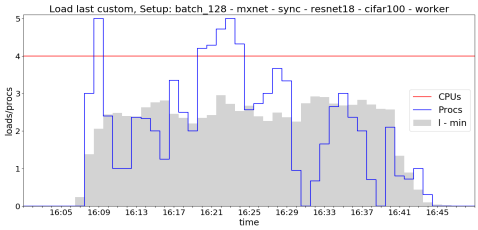

Load last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

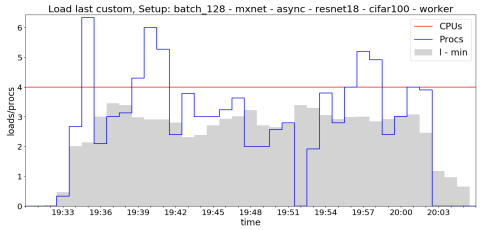

Load last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

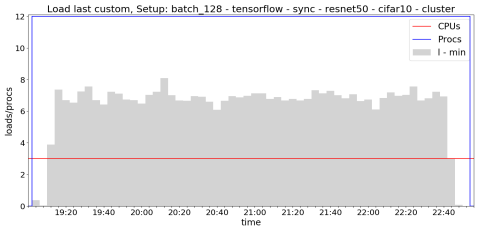

Load last custom, Setup: batch_128 - tensorflow - sync - resnet18 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time

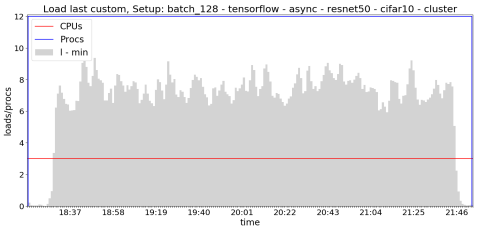

Load last custom, Setup: batch_128 - tensorflow - async - resnet18 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time

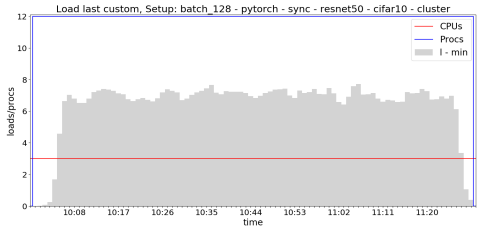

Load last custom, Setup: batch_128 - pytorch - sync - resnet18 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time

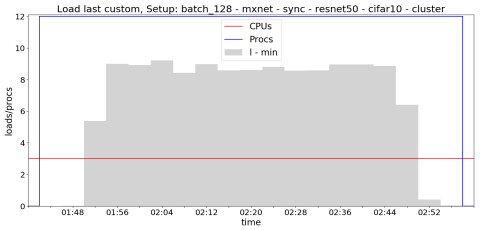

Load last custom, Setup: batch_128 - mxnet - sync - resnet18 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time

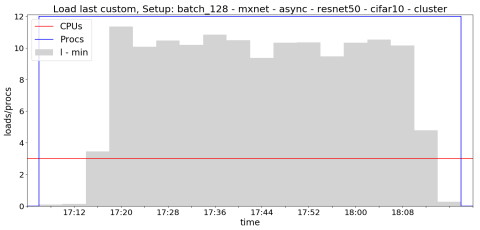

Load last custom, Setup: batch_128 - mxnet - async - resnet18 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time

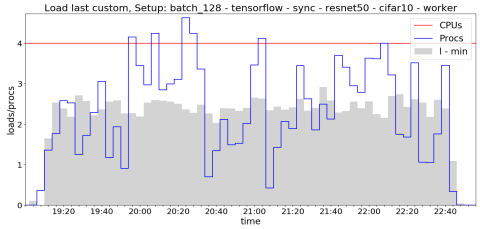

Load last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - cluster
CPUs
Procs
l - min
loads/procs
time

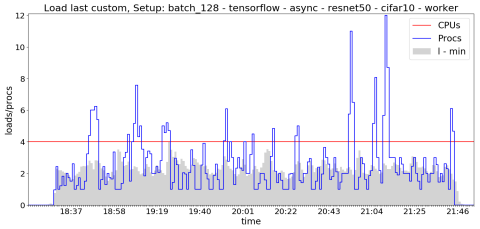

Load last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - cluster
CPUs
Procs
l - min
loads/procs
time

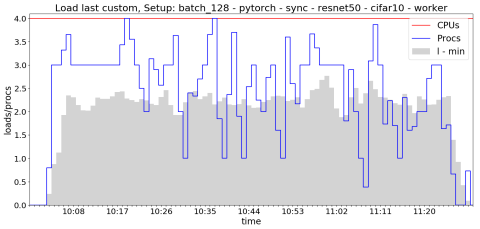

Load last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - cluster
CPUs
Procs
l - min
loads/procs
time

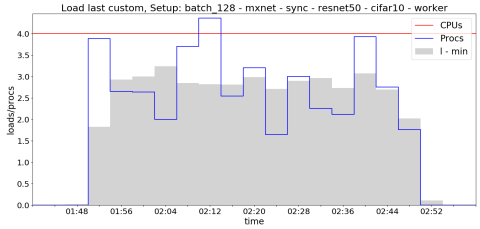

Load last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - cluster
CPUs
Procs
l - min
loads/procs
time

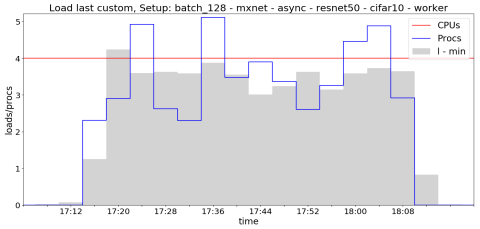

Load last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - cluster
CPUs
Procs
l - min
loads/procs
time

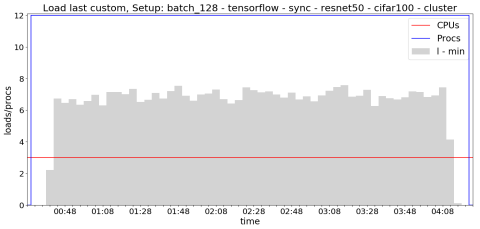

Load last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

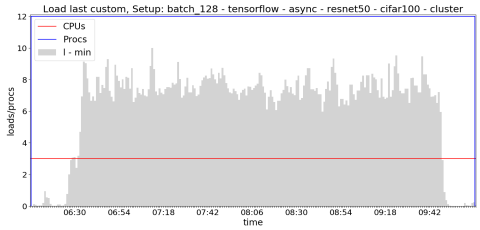

Load last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

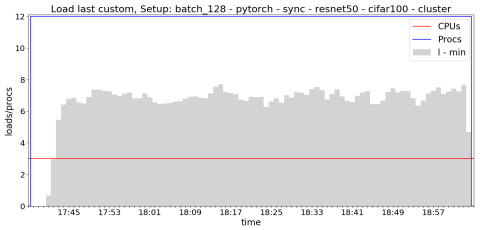

Load last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

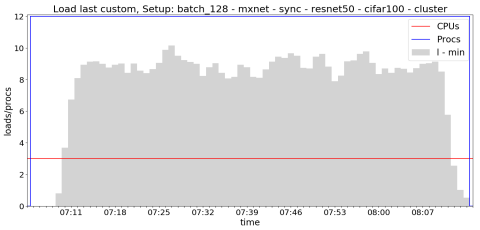

Load last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

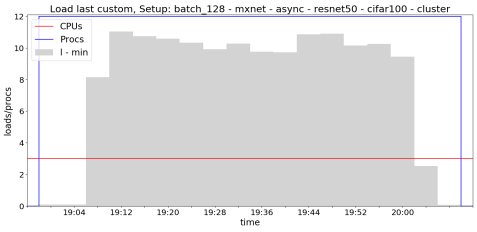

Load last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar10 - worker
CPUs
Procs
l - min
loads/procs
time

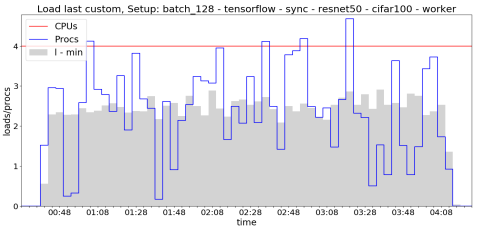

Load last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

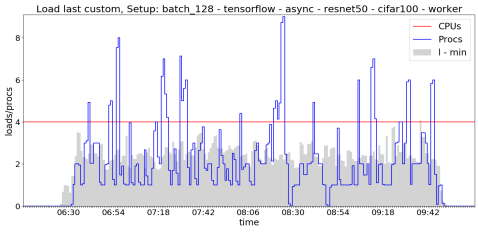

Load last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

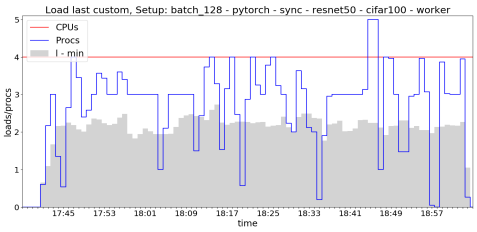

Load last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

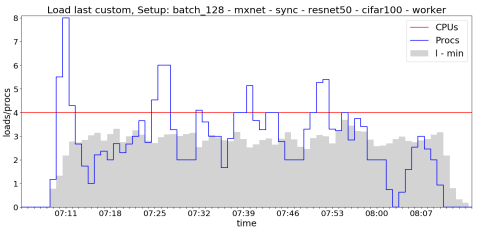

Load last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time

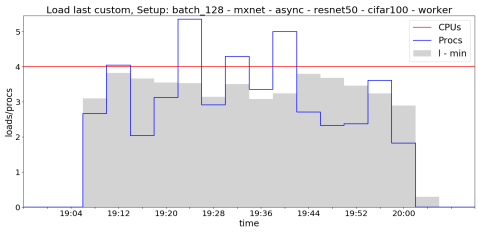

Load last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - cluster
CPUs
Procs
l - min
loads/procs
time


Load last custom, Setup: batch_128 - tensorflow - sync - resnet50 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time


Load last custom, Setup: batch_128 - tensorflow - async - resnet50 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time


Load last custom, Setup: batch_128 - pytorch - sync - resnet50 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time


Load last custom, Setup: batch_128 - mxnet - sync - resnet50 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time


Load last custom, Setup: batch_128 - mxnet - async - resnet50 - cifar100 - worker
CPUs
Procs
l - min
loads/procs
time