Università degli Studi di Modena e Reggio Emilia

DIPARTIMENTO DI INGEGNERIA ENZO FERRARI
Corso di Laurea in Ingegneria Informatica

Profiling of Distributed Training on Marconi100

Candidato:
Enrico Gherardi

Relatore:
Dott. Lorenzo Baraldi

Anno Accademico 2019-2020

Abstract in Lingua Italiana

Il regolamento del Dipartimento di Ingegneria Enzo Ferrari prevede che le tesi scritte in lingua inglese debbano contenere un'ampia sintesi dei contenuti in lingua italiana. In accordo, proponiamo un sunto dei modelli, delle tecniche e dei risultati che verranno delineati del corso del suo svolgimento. Per una descrizione più dettagliata e per una analisi rigorosa dei risultati sperimentali ottenuti si rimanda al testo in lingua inglese.
Nel corso degli anni sono stati intrapresi diversi approcci per modellare l'intelligenza umana nell'ambito dell'intelligenza artificiale. Il Machine Learning è un approccio molto popolare dell'AI che forma i sistemi ad apprendere, prendere decisioni e prevedere risultati da soli. Più nello specifico, il Deep Learning è una tecnica del Machine Learning ispirata al processo di apprendimento neurale del cervello umano. Il Deep Learning utilizza reti neurali profonde (Deep Neural Network), così chiamate per la loro struttura formata da neuroni artificiali organizzati su più livelli stratificati, che possono essere addestrati con enormi quantità di dati di input per risolvere rapidamente problemi complessi con elevata precisione. Nello specifico, attraverso le CNN (Convolutional Neural Network), un particolare tipo di DNN, il Deep Learning trova applicazione in numerosi campi quali Computer Vision, Bioinformatica e Natural Language Processing. Le CNN stanno ottenendo risultati sempre più strabilianti a fronte però di un incremento della loro complessità. Modelli sempre più complessi richiedono tempi di training sempre maggiori, che possono arrivare fino giorni o settimane.
Poiché le reti neurali sono create da un elevato numero di neuroni tutti identici, queste sono altamente paralellizzabili per natura. Tale parallelismo le associa naturalmente alle GPU, che permettono di raggiungere speedup molto significativi del training rispetto a quelli effettuati sulle tradizionali cpu. Infatti, le reti neurali fanno uso massiccio di operazioni matriciali, e complesse CNN multilivello richiedono un'enorme quantità di capacità computazionale in floating-point e di bandwidth per essere veloci ed efficienti. Le GPU hanno migliaia di processing-core ottimizzati per operazioni matriciali che permettono di raggiungere decine di centinaia di TFLOP. Le GPU, quindi, sono la piattaforma computazionale più performante per le reti neurali profonde e per le più svariate applicazioni di Machine Learning. Nonostante ciò, la complessa stratificazione delle nuove CNN e le enormi quantità di dati di input necessari rendono le operazioni di training su singola GPU decisamente lunghe in quanto possono arrivare a durare interi giorni e settimane. E' possibile ridurre i tempi di training attraverso due approcci: aumentare la precisione

del modello o diminuire il tempo di una singola epoca. Il primo approccio permette di ridurre il tempo totale di training riducendo il numero di epoche necessario per allenarlo. Tale obiettivo può essere raggiunto modificando parametri che riguardano direttamente la precisione del modello come il learning-rate e la batch-size. La seconda soluzione permette di diminuire il tempo richiesto per un training completo riducendo il tempo necessario per ogni singola epoca. Tale obiettivo può essere raggiunto aumentando il numero di GPU e di CPU utilizzate del processo di training. Siccome la precisione del modello può essere testata anche su un training tradizionale su singola GPU, si è deciso di sfruttare l'hardware a disposizione e di perseguire il secondo approccio cercando di ridurre al minimo il tempo di una singola epoca senza modificare la precisione del modello.
L'obiettivo di questa tesi, svolta in collaborazione con NVIDIA e CINECA, è l'implementazione e l'analisi delle prestazioni di Training Distribuito. In particolare, gli esperimenti verranno condotti sul nuovissimo Marconi100, il nuovo supercomputer in casa CINECA, che con i suoi 29,354 TFlop/s di Theoretical Peak si è classificato nono cluster più potente al mondo secondo la classifica stilata da top500.org. La CNN presa in considerazione è la Resnet18, che verrà allenata per l'operazione di classificazione. Il dataset usato per tutti gli esperimenti è ImageNet, che con i sui 3 milioni di immagini di training e 10000 di validation rappresenta uno dei dataset più famosi al mondo nell'ambito della Computer Vision.
Nel capitolo 2 vengono presentate le reti neurali. Si parte da una introduzione dei singoli neuroni artificiali, l'unità base attraverso la quale vengono costruite reti più complesse, descrivendone il modello matematico, la struttura e il loro funzionamento. Collegando tra loro molti neuroni è possibile ottenere una rete neurale artificiale, che ha l'obiettivo di approssimare una certa funzione. Le reti sono solitamente organizzate in layer e ogni neurone di un certo layer è connesso a tutti quelli dei layer precedente e successivo, ma non a quelli dello stesso. Affinché la rete approssimi tale funzione, viene sottoposta ad un processo di apprendimento, in cui le vengono presentati degli input di cui si conosce l'output desiderato detto ground truth. Ad ogni iterazione si confronta l'output corrente della rete con il risultato voluto e si aggiornano i suoi parametri in modo da minimizzare la differenza tra output corrente e ground truth. Vengono poi presentate le Reti Neurali Convoluzionali; delle reti particolarmente adatte all'elaborazione di immagini in cui ciascun layer trasforma un volume 3D di input in un volume 3D di output. I neuroni di un layer convoluzionale non sono connessi a tutti quelli del layer precedente, ma solo ad una porzione del volume di input, riducendo drasticamente il numero di parametri della rete e quindi la complessità computazionale. Viene poi fatta una introduzione generale al Deep Learning, campo di ricerca dell'intelligenza artificiale che utilizza un insieme di tecniche basate su reti neurali artificiali organizzate in diversi strati, dove ogni strato calcola i valori per quello successivo affinché l'informazione venga elaborata in maniera sempre più completa. Infine, viene introdotto uno dei task più famosi dell'intelligenza artificiale: la Classificazione. La classificazione è quel processo attraverso il quale si predice una classe di un oggetto dato un particolare input. In questo sotto-paragrafo vengono introdotti i

principali Classifier come k-Nearest Nieghbor e alcune reti artificiali. Viene anche fornito un insieme di tecniche utilizzate per valutare l'efficacia di un Classifier.
Nel capitolo 3 viene presentata la GPU come piattaforma computazionale nell'ambito del Deep Learning e, in particolare, l'NVIDIA Volta V100; il modello presente nei nodi di Marconi100 ed utilizzato nei nostri esperimenti. Sarà affrontato in dettaglio l'analisi della sua architettura e delle sue peculiarità che le permettono di raggiungere altissime prestazioni nel training e, in generale, nelle applicazioni di Machine Learning. Infine, viene presentato un approfondimento sul Training in Mixed Precision, una tecnica che permette di eseguire, dove è possibile senza perdere precisione, alcune operazioni floating-point a 16 bit invece che a 32. L'utilizzo di questa tecnica comporta numerosi vantaggi quali il minore utilizzo di memoria e minori tempi di computazione per le operazioni matematiche con precisione ridotta.
Nel capitolo 4 viene introdotto il framework utilizzato per l'implementazione del Training Distribuito: PyTorch. In particolare, vengono descritte le sue principali features: i tensori, la classe per modellare le reti neurali e le classi per implementare il parallelismo dei dati e del modello (DataParallel e DistributedDataParallel). DataParallel replica lo stesso modello su tutte le GPU, e ognuna di esse processa una porzione differente dei dati di input. DistributedDataParallel, invece, implementa il parallelismo dei dati a livello di modulo. Le applicazioni che utilizzano DDP generano più processi, generalmente uno per ogni GPU richiesta, e creano una singola istanza DDP per ciascuno di essi. Ogni sotto-processo, successivamente, utilizzerà comunicazioni collettive per sincronizzare i gradienti e i buffer. La nostra implementazione utilizza DistributedDataParallel.
Nel capitolo 5 viene presentato nel dettaglio Marconi100, il supercomputer utilizzato nei nostri esperimenti. In particolare, viene descritta la sua architettura composta da 980 nodi (più 8 nodi di login), ognuno dei quali composto da 2x16 core IBM POWER9 AC922, 4 GPU NVIDIA Volta V100 e 256 GB di RAM. Infine, viene introdotto SLURM, il cluster manager e job scheduler utilizzato da Marconi100.
Il sesto capitolo, finalmente, mostra i risultati sperimentali ottenuti. Nella sua prima parte c'è una breve descrizione dello script in Python, utilizzato per il training distribuito, e in bash, per il lancio del training. Successivamente, è presente una analisi dei dati raccolti e delle prestazioni del training al variare del numero di GPU, del numero di nodi, del numero di worker, della batch-size e con l'utilizzo di tecnologie che permettono training in mixed precision.
Al variare di GPU su singolo nodo si nota una decrescita, non lineare, del tempo necessario per una singola epoca che si assesta al crescere del numero di GPU. La decrescita non è lineare in quanto, come previsto da DistributedDataParallel, tutti i processi devono sincronizzarsi tra di loro e perché i worker non riescono a riempire abbastanza velocemente le batch di tutte le GPU quando il loro numero inizia ad essere elevato. Al variare del numero di nodi, invece, l'incremento di prestazioni può essere quasi considerato lineare. L'unico rallentamento riscontrato rispetto alla condizione ideale è rappresentato dal meccanismo

di sincronizzazione tra nodi. Una situazione ancora diversa si profila al variare del numero di worker. Le prestazioni crescono, non linearmente, all'aumentare del loro numero fino a che non si raggiunge un numero di worker sufficiente a riempire regolarmente le code delle batch di tutte le GPU. Infine, è stato condotto un esperimento sui benefici del Training con Mixed Precision. Questo ha portato ad una diminuzione del tempo di una singola epoca del 6.9% dovuto alla maggiore velocità delle operazioni matematiche in precisione ridotta. Attraverso il training distribuito, quindi, è possibile ridurre drasticamente il tempo necessario per un training completo. Contando che, in media, un training per addestrare una CNN alla classificazione richiede 90 epoche, siamo riusciti a passare da un tempo per singola epoca di 57 minuti e 46 secondi e totale di 3 giorni, 15 ore e 36 minuti (su singolo nodo e singola GPU) ad un tempo per singola epoca di 2 minuti e 19 secondi e totale di 3 ore e 28 minuti (su 16 nodi, 64 GPU e 512 cpu).

Ringraziamenti

Un sincero rigraziamento a Lorenzo Baraldi per avermi guidato in questo lavoro di tesi e per la passione e la professionalità che mi ha sempre trasmesso in questi mesi. Doveroso è anche ringraziare l'Ing. Matteo Tomei per il suo supporto e il mio collega, ora amico, Enrico Zelioli con cui mi sono sempre confrontato e relazionato nello svolgimento di questo lavoro.
Intendo anche ringraziare il Dott. Ing. Giuseppe Fiamenti di NVIDIA per la sua disponibilità e i consigli sempre utili e il Dott. Marco Rorro di CINECA.
Un grazie alla mia famiglia, che mi ha permesso di raggiungere questo traguardo grazie al suo solido sostegno, sia morale che economico e ai miei amici più stretti per la loro costante presenza.
Un grazie a Beatrice per avermi sempre ascoltato in questo ultimo anno, per i suoi consigli e per il suo immancabile supporto nei momenti più difficili.
Infine, un grazie a Enrico, Federico, Giuseppe, Luca, Matteo e Riccardo. Nessun pensiero scritto potrebbe rendere giustizia alle giornate di lezione, di studio, agli esami preparati insieme e a tutti i momenti divertenti trascorsi in questi tre anni.

Contents

List of Figures

Chapter 1

Introduction

Many different approaches have been used over the years to model human intelligence in the field of Artificial Intelligence. Machine Learning is a very popular approach to AI that train systems to learn how to make decisions and predict results on their own. Deep learning is a machine learning technique inspired by the neural learning process of the human brain. Deep learning uses deep neural networks (DNNs), so called because of their deep layering of many connected artificial neurons (sometimes called perceptrons), which can be trained with enormous amounts of input data to quickly solve complex problems with high accuracy.
Neural networks are created from large numbers of identical neurons, so they are highly parallel by nature. This parallelism maps naturally to GPUs, which provide a significant speedup over CPU-only training. Neural networks rely heavily on matrix math operations, and complex multi-layered networks require tremendous amounts of floating-point performance and bandwidth for both efficiency and speed. GPUs have thousands of processing cores optimized for matrix math operations, providing tens to hundreds of TFLOPS of performance. GPUs are the obvious computing platform for deep neural network-based artificial intelligence and machine learning applications. Despite this, the complex multi-layer structure of the new DNNs and the enormous amounts of input data required make the Training operation last whole days and weeks.
It is possible to reduce training times through two approaches: increase the accuracy of the model or decrease the epoch time. The first approach permits to reduce the training time by reducing the number of necessary epochs. This goal can be achieved by modifying parameters that directly concern the precision of the model such as learning-rate and batch-size. The second solution permits to decrease the time required for a complete training by reducing the time required for each single epoch. This goal can be achieved by increasing the number of GPUs and CPUs used in the training process. Since the precision of the model can also be tested on a traditional training with a single GPU, we decided to use the hardware available and to pursue the second approach by trying to minimize the epoch time without changing the precision of the model.

The aim of this thesis, carried out in collaboration with NVIDIA and CINECA, is the implementation and profiling of Distributed Training. In particular, the experiments will be conducted on the brand new Marconi100, the new supercomputer from CINECA, which with its 29.354 TFlop/s of Theoretical Peak is the ninth most powerful cluster in the world according to the ranking compiled by top500.org. The CNN used in our experiments is the Resnet18, which will be trained for the classification task. The dataset is ImageNet, which with its 3 million training and 10,000 validation images represents one of the most famous datasets in the world in the field of Computer Vision.

Chapter 2

Neural Network

The definition of a neural network, more properly referred to as an 'artificial' neural network (ANN), is provided by the inventor of one of the first neurocomputers, Dr. Robert Hecht-Nielsen. He defines a neural network as:

"...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs."

The Artificial Neural Network as computational model is inspired by the way biological neural networks work in the human brain process information. In this chapter we will introduce the mathematical model of a neuron and we will see how, connecting more of them, it is possible to build complex neural network. Then we will describe the Convolutional Neural Networks and the Resnet, the one that we used for our experiments. Finally, we will analyse a specific task of machine learning: the classification.

2.1 Neuron and neural networks

2.1.1 The model of the neuron

The basic unit of computation in a neural network is the neuron, often called a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated weight (w), which is assigned based on its relative importance to other inputs. The node applies a function to the weighted sum of its inputs. The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence and its direction: excitory (positive weight) or inhibitory (negative weight) of one neuron on another. Now we will describe more accurately his mathematical model:

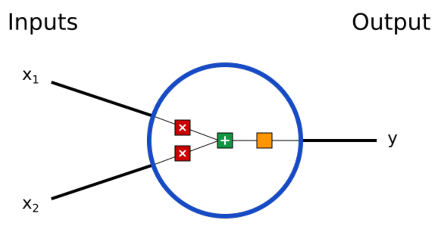


Figure 2.1: The artifical neuron

There are three important steps to analyse here. First, each input is multiplied by a weight:

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

Next, all the weighted inputs are added together with a bias b:

$$(x_1 * w_1) + (x_2 * w_2) + b$$

Finally, the sum is passed through an activation function:

$$y = f(x_1 * w_1 + x_2 * w_2 + b)$$

The activation function is used to turn an unbounded input into an output that has a predictable form. There are many activation functions, for example the Sigmoid, the Tanh and the ReLu.

2.1.2 Feedforward Neural Network

A neural network a bunch of neurons connected. One of the easiest types of neural network is the Feedforward Neural Network. A feedforward neural network is an artificial neural network where connections between the units do not form a cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network. We can distinguish two types of Feedfoward Neural networks.
The Single-layer Perceptron is the simplest feedforward neural Network and does not contain any hidden layer, which means it only consists of a single layer of output nodes. This is said to be single because when we count the layers we do not include the input layer,

the reason for that is because at the input layer no computations is done, the inputs are fed directly to the outputs via a series of weights.

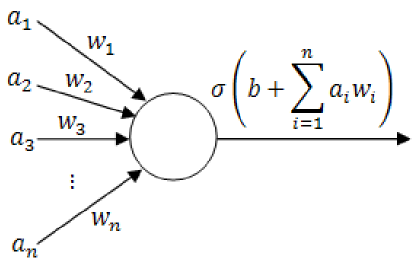


Figure 2.2: The mathematical model of a artifical neuron

The multi-layer perceptron (MLP) consists of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the subsequent layer. In many applications the units of these networks apply a sigmoid function as an activation function. MLP are very useful. One of their success's reason is that they can learn non-linear representation.

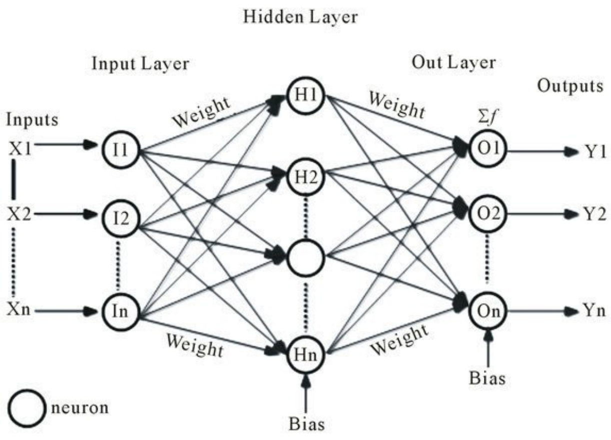


Figure 2.3: Feedfoward network

2.1.3 Loss

The Loss function, also called cost function or error function, measures the inconsistency between the predicted value and the actual one. This function is very important because it give us a way to quantify how good is our network. Training a network, in fact, consists in trying to minimize its loss. The data loss is the average of the data losses for every individual example. Where n is the number of training data. There are many loss functions referred to individual examples, the most used are:
The Absolute error, used to measure how close forecasts or predictions are to the actual outcomes.

The Squared error, widely used in linear regression (the task of predicting real-valued quantities):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

To minimize loss, we can change network's weights and biases, but we must understand how they work and influence our result.
To analyse it in an easier way, we will study it in a simple Feedfoward Neural network with a Sigmoid activation function

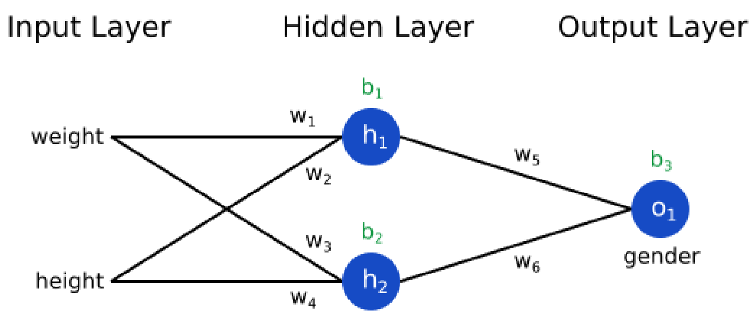


Figure 2.4: Simple neural network

We can describe loss as a multivariable function:

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

If we tweak w1, the loss L will change, we can express how it change with the partial derivate:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

If we assume that y_{true}=1 we can calculate $\frac{\partial L}{\partial y_{pred}}$:

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial(1 - y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1 - y_{pred})}$$

We can write y_{pred} as:

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

Now we can calculate $\frac{\partial y_{pred}}{\partial w1}$:

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

And we do the same thing for $\frac{\partial h1}{\partial w1}$:

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

Now we can replace f(x) and f'(x) with the Sigmoid function:

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

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) * (1 - f(x))$$

In this way we have all we need to calculate $\frac{\partial L}{\partial w1}$:

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

This system of calculating partial derivatives by working backwards is known as **back-propagation.**

2.1.4 Training

As we said before, the training's goal is to minimize loss. One of the most frequent algorithms used to train a neural network is the Stochastic Gradient Descent (SGD). It tells us how to change our weights and biases to minimize loss through the equation:

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

η is a constant called the learning rate that controls how fast we train. All we're doing is subtracting $\eta\frac{\partial w1}{\partial L}$ from w1: If $\frac{\partial w1}{\partial L}$ is positive, w1 will decrease, which makes L decrease, if $\frac{\partial w1}{\partial L}$ is negative, w1 will increase, which makes L decrease. If we do this for every weight and bias in the network, the loss will slowly decrease, and our network will improve.

2.2 Convolutional Neural Network

MLPs are very used but they have several drawbacks when they come to image processing. First, MLPs use one perceptron for each input (e.g. pixel in an image, multiplied by 3 in RGB case). The amount of weights rapidly becomes unmanageable for large images. Another common problem is that MLPs are not translation invariant: they react differently to an input and its shifted version. For this reason, spatial information is lost when the image is flattened into an MLP.
To solve this problem, we need a network that consider the nearby pixels more strongly related than distant ones and objects that are built up out of smaller parts. The influence of nearby pixels is analysed with a filter. For each image, we can take a filter of a size specified by the user (a rule of thumb is 3x3 or 5x5) and we can move this across the image from top left to bottom right. For each point on the image, a value is calculated based on the filter using a convolution operation. A filter could be related to any object and it would give us an indication of how strongly that object seems to appear in an image, and how many times and in what locations they occur. This reduces the number of weights that the neural network must learn compared to an MLP, and also means that when the location of these features changes it does not throw the neural network off.

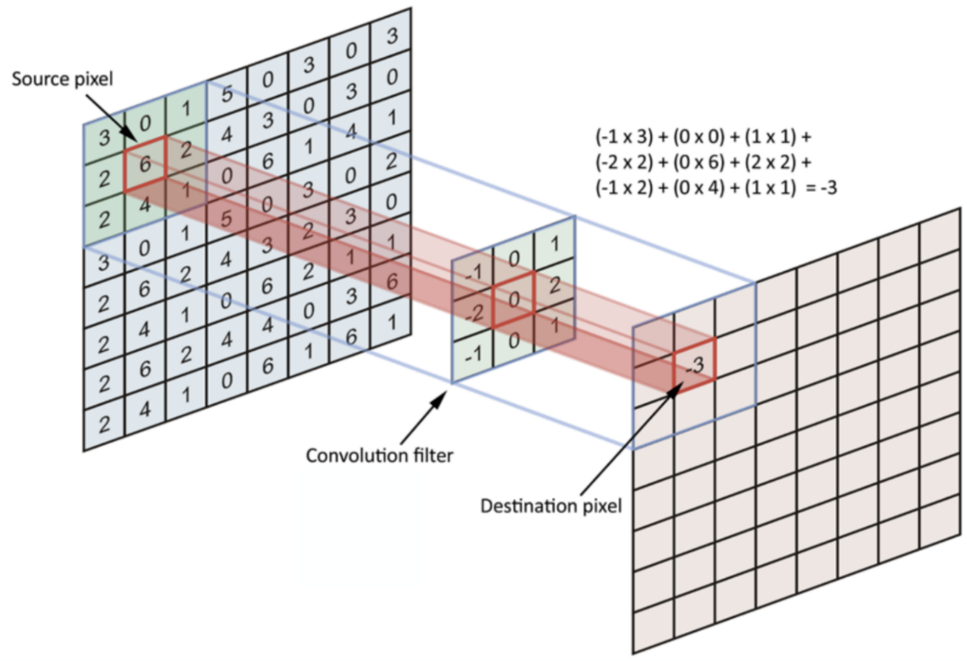

Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
Destination pixel

Figure 2.5: The convolution operation

When building the network, we randomly specify values for the filters, which then continuously update themselves as the network is trained. After the filters have passed over the image, a feature map is generated for each filter. These are then taken through an activation function, which decides whether a certain feature is present at a given location in the image. We can then do a lot of things, such as adding more filtering layers and creating more feature maps, which become more and more abstract as we create a deeper CNN. We can also use pooling layers in order to select the largest values on the feature maps and use these as inputs to subsequent layers. CNN's are also composed of layers, but those layers are not fully connected: they have filters, sets of cube-shaped weights that are applied throughout the image. Each 2D slice of the filters are called kernels. These filters introduce translation invariance and parameter sharing and they are applied with convolution.
Each filter is convolved with the entirety of the 3D input cube but generates a 2D feature map. The feature map dimension can change drastically from one convolutional layer to the next: we can enter a layer with a 32x32x16 input and exit with a 32x32x128 output if that layer has 128 filters. Convolving the image with a filter produces a feature map that highlights the presence of a given feature in the image.

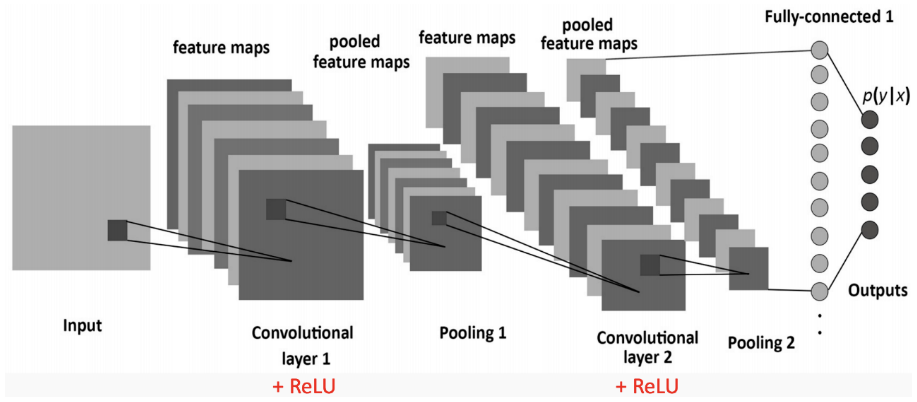


Figure 2.6: An example CNN with two convolutional layers, two pooling layers, and a fully connected layer which decides the final classification of the image into one of several categories.

There are three types of layers in a convolutional neural network: convolutional layer, pooling layer, and fully connected layer. Each of these layers has different parameters that can be optimized and performs different tasks on the input data.
Convolutional layers are the layers where filters are applied to the original image, or to other feature maps in a deep CNN. This is where most of the user-specified parameters are in the network. The most important parameters are the number of kernels and the size of the kernels.
Pooling layers are similar to convolutional layers, but they perform a specific function such as max pooling, which takes the maximum value in a certain filter region, or average pooling, which takes the average value in a filter region. These are typically used to reduce the dimensionality of the network.
Fully connected layers are placed before the classification output of a CNN and are used to flatten the results before classification. This is like the output layer of an MLP.

2.2.1 ResNet CNN Networks

A residual neural network (ResNet) is Convolutional Neural Network. There are a lot of models of ResNet, some are very deep and can arrive to 152 layers. In our experiments we decided to use the ResNet-18 model. This network is very important because it paved the way for residual networks thanks to its technique known as skip connection. In summary, the processed data transferred to the next level is also added to the output of the previous ones, according to a defined scheme.

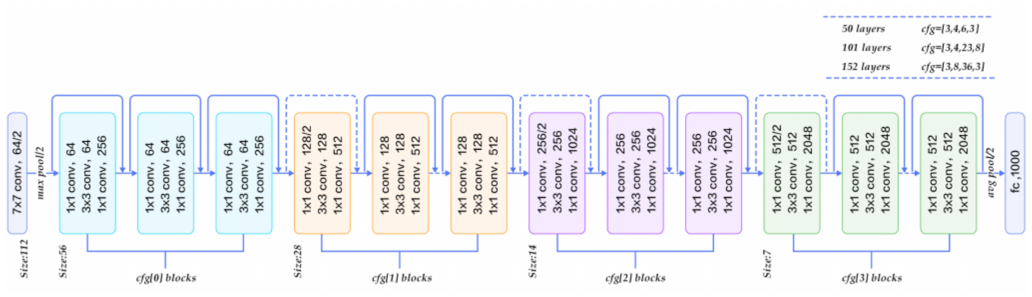


Figure 2.7: The Resnet

Resnet is organized in defined residual block. Thanks to its architecture, the ResNet can solve problems of exploding or vanishing gradient: in fact, instead of waiting for the backpropagation one level at a time, the skip connection path allows it to reach the initial nodes effectively by skipping the intermediate ones.

2.3 Deep Learning

Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher level features from the raw input. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces.

The word "deep" in "deep learning" refers to the number of layers through which the data is transformed. More precisely, deep learning systems have a substantial credit assignment path (CAP) depth. The CAP is the chain of transformations from input to output. CAPs describe potentially causal connections between input and output. For a feedforward neural network, the depth of the CAPs is that of the network and is the number of hidden layers plus one (as the output layer is also parameterized). For recurrent neural networks, in which a signal may propagate through a layer more than once, the CAP depth is potentially unlimited. No universally agreed upon threshold of depth divides shallow learning from deep learning, but most researchers agree that deep learning involves CAP depth higher than 2. CAP of depth 2 has been shown to be a universal approximator in the sense that it can emulate any function. Beyond that, more layers do not add to the function approximator ability of the network. Deep models (CAP > 2) are able to extract better features than shallow models and hence, extra layers help in learning the features effectively.

Deep learning architectures such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, machine vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance.

2.4 Classification

Classification is, probably, the most famous machine learning's task. In fact, there are many applications in classification in many domains such as in credit approval, medical diagnosis, target marketing etc. Classification is the process of predicting the class of given data points. Classes are sometimes called as targets/ labels or categories. Classification predictive modelling is the task of approximating a mapping function (f) from input variables (X) to discrete output variables (y). For example, spam detection in email service providers can be identified as a classification problem. This is s binary classification since there are only 2 classes as spam and not spam. A classifier utilizes some training data to understand how given input variables relate to the class. In this case, known spam and non-spam emails have to be used as the training data. When the classifier is trained accurately, it can be used to detect an unknown email.
There are two types of learners in classification as lazy learners and eager learners:

Lazy learners: they simply store the training data and wait until a testing data appear. When it does, classification is conducted based on the most related data in the stored training data. Compared to eager learners, lazy learners have less training time but more time in predicting. Ex. k-nearest neighbor, Case-based reasoning.

Eager learners: they construct a classification model based on the given training data before receiving data for classification. It must be able to commit to a single hypothesis that covers the entire instance space. Due to the model construction, eager learners take a long time for train and less time to predict. Ex. Decision Tree, Naive Bayes, Artificial Neural Networks.

There is a lot of classification algorithms (classifiers) available now, but it is not possible to conclude which one is superior to other. It depends on the application and nature of available data set. So, we will consider the most relevant to our case.

k-Nearest Neigbor (KNN)
k-Nearest Neighbor is a lazy learning algorithm which stores all instances correspond to training data points in n-dimensional space. When an unknown discrete data is received, it analyses the closest k number of instances saved (nearest neighbors) and returns the most common class as the prediction and for real-valued data it returns the mean of k nearest neighbors.

$$w \equiv \frac{1}{d(x_q, x_i)^2}$$

In the distance-weighted nearest neighbor algorithm, it weights the contribution of each of the k neighbors according to their distance using the following query giving greater weight to the closest neighbors.
Usually KNN is robust to noisy data since it is averaging the k-nearest neighbors.

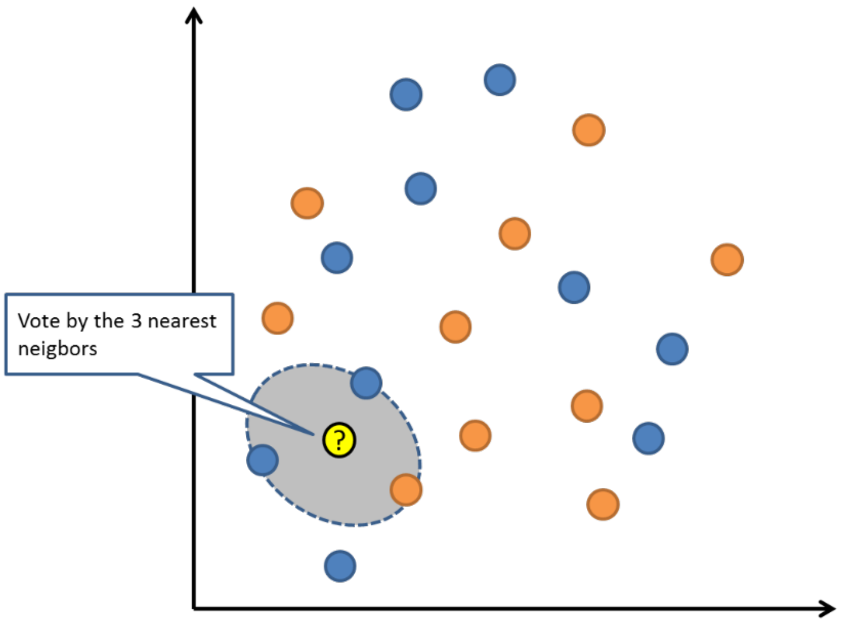


Figure 2.8: k-Nearest Neighbor

Artificial Neural Networks
There are many network architectures available now like Feed-forward, Convolutional, Recurrent etc. The appropriate architecture depends on the application of the model. For most cases feed-forward models give reasonably accurate results and especially for image processing applications, convolutional networks perform better. But Artificial Neural Networks have performed impressively in most of the real-world applications. It is high tolerance to noisy data and able to classify untrained patterns. Usually, Artificial Neural Networks perform better with continuous-valued inputs and outputs. All the above algorithms are eager learners since they train a model in advance to generalize the training data and use it for prediction later.
Linear Classification A linear classifier does classification decision based on the value of a linear combination of the characteristics. Imagine that the linear classifier will merge into its weights all the characteristics that define a peculiar class.

$f(\vec{x}, \vec{W}, \vec{b}) = \sum_j (Wjxj) + b$

x: input vector
W: Weight matrix
b: Bias vector

There are two variables to defines if the score, and consequently the weights W, are good.
Top-1 accuracy is the conventional accuracy, which means that the model answer (the one with the highest probability) must be exactly the expected answer.
Top-5 accuracy means that any of your model that gives 5 highest probability answers that must match the expected answer.

Evaluating a classifier After training the model the most important part is to evaluate the classifier to verify its applicability. There are several methods to evaluate a classifier:

- **Holdout method:** the most common. n this method, the given data set is divided into 2 partitions as test and train 20% and 80% respectively. The train set will be used to train the model and the unseen test data will be used to test its predictive power.

- **Cross validation:** Over-fitting is a common problem in machine learning which can occur in most models. k-fold cross-validation can be conducted to verify that the model is not over-fitted. In this method, the dataset is randomly partitioned into k mutually exclusive subsets, each approximately equal size and one is kept for testing while others are used for training. This process is iterated throughout the whole k folds.

- **ROC curve (Receiver Operating Characteristics):** ROC curve is used for visual comparison of classification models which shows the trade-off between the true positive rate and the false positive rate. The area under the ROC curve is a measure of the accuracy of the model. When a model is closer to the diagonal, it is less accurate and the model with perfect accuracy will have an area of 1.0.

2.5 Dataset

Machine learning typically works with two data sets: training and test. Both should randomly sample a larger body of data. The training set is the largest one. It is used by a neural network during the training phase when the net adjusts them coefficients to minimize errors in the results.
The second set is the test set. It functions as a seal of approval, and it is not used until the end. After the model if trained, you can test the neural net against this final random sampling. The results it produces should validate that the net accurately recognizes images or recognizes them at least a percentage of them.
The dataset used in our experiments is the IMAGENET.

The ImageNet project is a large visual database designed for use in visual object recognition software research. More than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided. ImageNet contains more than 20,000 categories with a typical category, such as "balloon" or "strawberry", consisting of several hundred images. The database of annotations of third-party image URLs is freely available directly from ImageNet, though the actual images are not owned by ImageNet. Since 2010, the ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes.

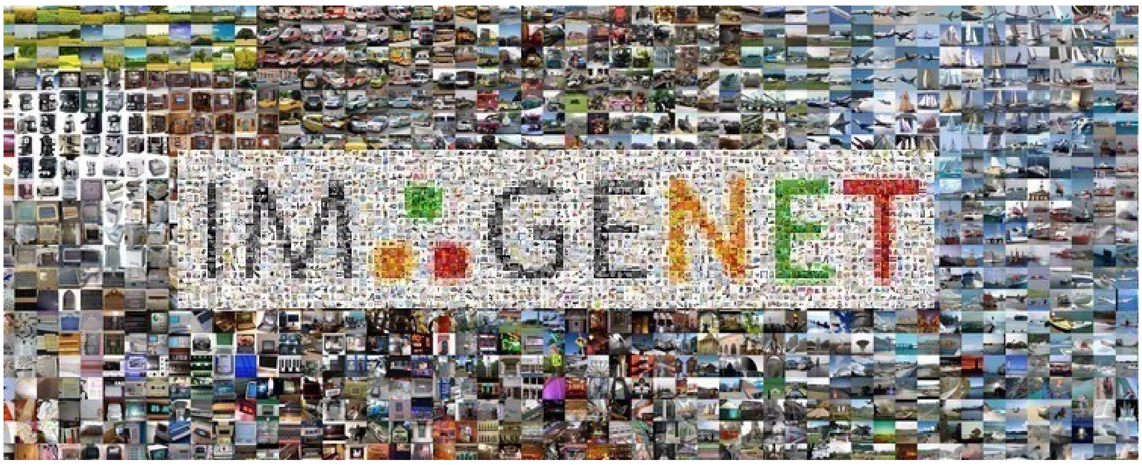


Figure 2.9: Imagenet's Logo

Chapter 3

GPUs and Deep Learning

Neural networks are created from large numbers of identical neurons, they are highly parallel by nature. This parallelism maps naturally to GPUs, which provide a significant speedup over CPU-only training.
Neural networks rely heavily on matrix math operations, and complex multi-layered networks require tremendous amounts of floating-point performance and bandwidth for both efficiency and speed. GPUs have thousands of processing cores optimized for matrix math operations, providing tens to hundreds of TFLOPS of performance. GPUs are the obvious computing platform for deep neural network-based artificial intelligence and machine learning applications.
In this chapter we describe the NVIDIA V100, the GPU used in our experiments, and we delve into its new Volta architecture by focusing on its innovations. Finally, we describe Mixed Precision Training, a new technology developed by NVIDIA that permit to reduce the required amount of memory and the computational time during training.

3.1 NVIDIA V100

The NVIDIA V100 is the GPU used in our experiments. The NVIDIA V100 Tensor Core GPU is the world's most powerful accelerator for deep learning, machine learning, high-performance computing (HPC), and graphics. Powered by NVIDIA Volta, a single V100 Tensor Core GPU offers the performance of nearly 32 CPUs—enabling researchers to tackle challenges that were once unsolvable. The V100 won MLPerf, the first industry-wide AI benchmark, validating itself as the world's most powerful, scalable, and versatile computing platform.
The NVIDIA V100 has 640 Tensor Cores and 5.120 CUDA cores which allow it to reach from 7 to 8.2 TFLOPS for Double-Precision Performance, from 14 to 16.4 TFLOPS for Double-Precision Performance and from 112 TFLOPS to 130 TFLOPS for Tensor Performance (depending on the V100 version selected).
NVIDIA V100's main innovations are:

- **Volta Architecture:** By pairing CUDA cores and Tensor Cores within a unified architecture, a single server with V100 GPUs can replace hundreds of commodity CPU servers for traditional HPC and deep learning.

- **NVLink:** NVIDIA NVLink in V100 delivers 2X higher throughput compared to the previous generation. Up to eight V100 accelerators can be interconnected at up to gigabytes per second (GB/sec) to unleash the highest application performance possible on a single server.

- **HBM2:** With a combination of improved raw bandwidth of 900GB/s and higher DRAM utilization efficiency at 95%, V100 delivers 1.5X higher memory bandwidth over Pascal GPUs as measured on STREAM.

NVIDIA V100 offers a support for every deep learning framework (Pytorch, TensorFlow, Theano)

Figure 3.1: NVIDIA Volta V100

3.2 Volta Architecture In-Depth

Volta features a new Streaming Multiprocessor (SM) architecture that delivers major improvements in performance, energy efficiency, and ease of programmability. Major features include:

1. New mixed-precision Tensor Cores purpose-built for deep learning matrix arithmetic, delivering 12x TFLOPS for training, compared to GP100, in the same power envelope

2. 50% higher energy efficiency on general compute workloads

3. Enhanced high performance L1 data cache
4. A new SIMT thread model that removes limitations present in previous SIMT and SIMD processor designs

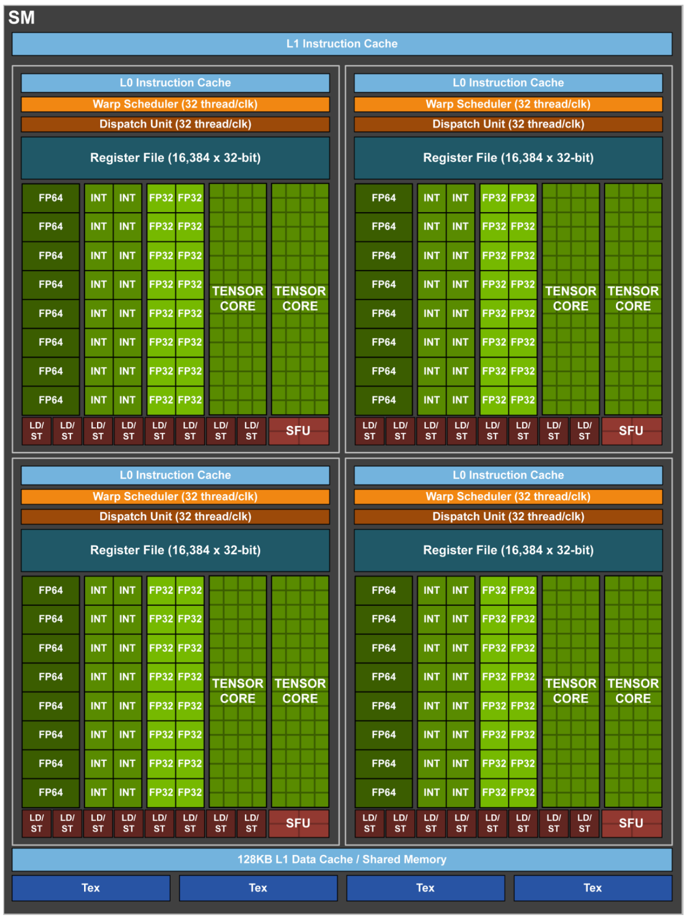

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure 3.2: Volta Architecture

3.2.1 Tensor Core

New Tensor Cores are a key capability enabling the Volta GV100 GPU architecture to deliver the performance required to train large neural networks.
The Tesla V100 GPU contains 640 Tensor Cores: eight per SM and two per each processing block (partition) within an SM. In Volta GV100, each Tensor Core performs 64 floating point FMA operations per clock, and eight Tensor Cores in an SM perform a total of 512 FMA operations (or 1024 individual floating point operations) per clock. Tesla V100's Tensor Cores deliver up to 125 Tensor TFLOPS for training and inference applications. Tensor Cores provide up to 12x higher peak TFLOPS on Tesla V100 that can be applied to deep learning training.
Matrix-Matrix multiplication (GEMM) operations are at the core of neural network training and inferencing and are used to multiply large matrices of input data and weights

in the connected layers of the network. Tensor Cores and their associated data paths are custom-designed to dramatically increase floating-point compute throughput with high energy efficiency.
Each Tensor Core operates on a 4x4 matrix and performs the following operation: $D = A * B + C$
where A, B, C, and D are 4x4 matrices. The matrix multiply inputs A and B are FP16 matrices, while the accumulation matrices C and D may be FP16 or FP32 matrices

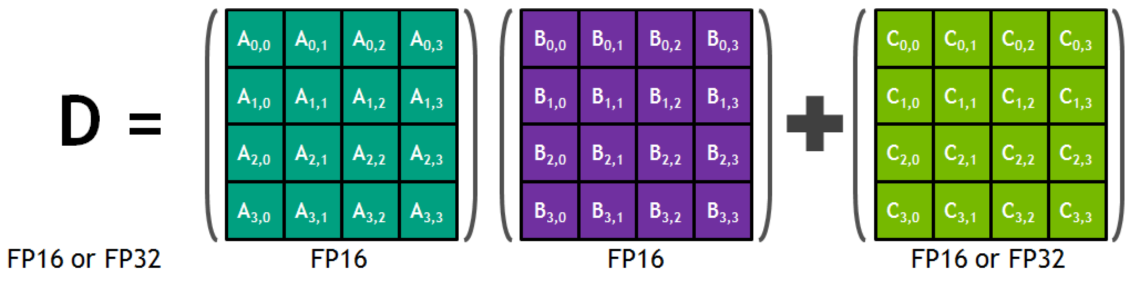


Figure 3.3: Operation in Mixed Precision

Tensor Cores operate on FP16 input data with FP32 accumulation. The FP16 multiply results in a full precision product that is then accumulated using FP32 addition with the other intermediate products for a 4x4x4 matrix multiply. In practice, Tensor Cores are used to perform much larger 2D or higher dimensional matrix operations, built up from these smaller elements.

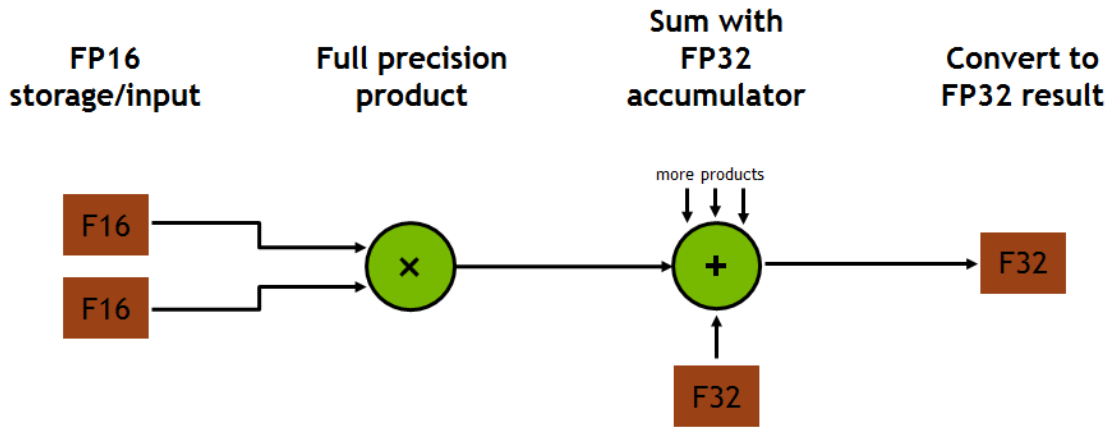


Figure 3.4: Implementation of Mixed Precision

3.2.2 L1 Cache and Shared Memory

Combining data cache and shared memory functionality into a single memory block provides the best overall performance for both types of memory accesses. The combined capacity is 128 KB/SM, more than seven times larger than the GP100 data cache, and all of it is usable as a cache by programs that do not use shared memory. A key reason to merge the L1 data cache with shared memory in GV100 is to allow L1 cache operations to attain the benefits of shared memory performance. Shared memory provides high bandwidth, low latency, and consistent performance (no cache misses), but the CUDA programmer needs to explicitly manage this memory. Volta narrows the gap between applications that explicitly manage shared memory and those that access data in device memory directly.

3.2.3 NVLink

NVLink is NVIDIA's high-speed interconnect technology first introduced in 2016. NVLink provides significantly more performance for both GPU-to-GPU and GPU-to-CPU system configurations compared to using PCIe interconnects. Tesla V100 introduces the second generation of NVLink, which provides higher link speeds, more links per GPU, CPU mastering, cache coherence, and scalability improvements.
Each link now provides 25 Gigabytes/second in each direction. The number of links supported has been increased from four to six pushing the supported GPU NVLink bandwidth to 300 Gigabytes/second. The second generation of NVLink allows direct load/store/atomic access from the CPU to each GPU's HBM2 memory. Coupled with a new CPU mastering capability, NVLink supports coherency operations allowing data reads from graphics memory to be stored in the CPU's cache hierarchy.

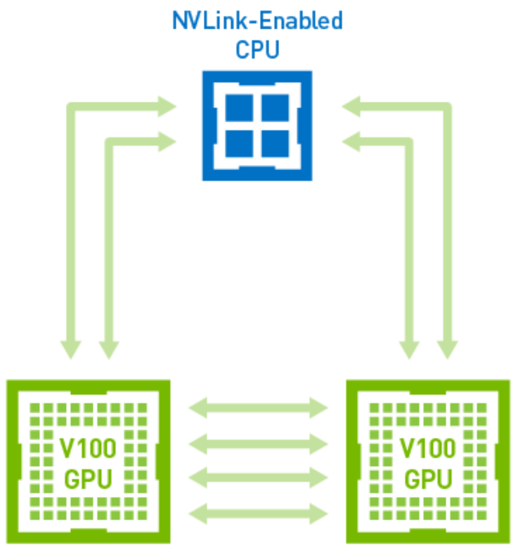


Figure 3.5: NVLink

3.2.4 HBM2 Memory Architecture

HBM2 memory is composed of memory stacks located on the same physical package as the GPU, providing substantial power and area savings compared to traditional GDDR5 memory designs, thus permitting more GPUs to be installed in servers.

HBM2 in Tesla V100 uses four memory dies per HBM2 stack, and four stacks, with a maximum of 16 GB of GPU memory. The HBM2 memory delivers 900 GB/sec of peak memory bandwidth across the four stacks.

3.3 Mixed Precision Training

There are numerous benefits to using numerical formats with lower precision than 32-bit floating point. First, they require less memory, enabling the training and deployment of larger neural networks. Second, they require less memory bandwidth, thereby speeding up data transfer operations. Third, math operations run much faster in reduced precision, especially on GPUs with Tensor Core support for that precision. Mixed precision training achieves all these benefits while ensuring that no task-specific accuracy is lost compared to full precision training. It does so by identifying the steps that require full precision and using 32-bit floating point for only those steps while using 16-bit floating point everywhere else.

Mixed precision training offers significant computational speedup by performing operations in half-precision format, while storing minimal information in single-precision to retain as much information as possible in critical parts of the network. Since the introduction of Tensor Cores in the Volta and Turing architectures, significant training speedups are experienced by switching to mixed precision – up to 3x overall speedup on the most arithmetically intense model architectures. Using mixed precision training requires two steps:

1. Porting the model to use the FP16 data type where appropriate.
2. Adding loss scaling to preserve small gradient values.

Mixed precision is the combined use of different numerical precisions in a computational method.

Half precision (also known as FP16) data compared to higher precision FP32 vs FP64 reduces memory usage of the neural network, allowing training and deployment of larger networks, and FP16 data transfers take less time than FP32 or FP64 transfers.

Single precision (also known as 32-bit) is a common floating point format (float in C-derived programming languages), and 64-bit, known as double precision (double).

One way to lower the required resources to train a complex CNN is to use lower-precision arithmetic, which has the following benefits:

Decrease the required amount of memory: Half-precision floating point format (FP16) uses 16 bits, compared to 32 bits for single precision (FP32). Lowering the required memory enables training of larger models or training with larger mini-batches.

Shorten the training or inference time: Execution time can be sensitive to memory or arithmetic bandwidth. Half-precision halves the number of bytes accessed, thus reducing the time spent in memory-limited layers. NVIDIA GPUs offer up to 8x more half precision arithmetic throughput when compared to single-precision, thus speeding up math-limited layers.

The Volta generation of GPUs introduces Tensor Cores, which provide 8x more throughput than single precision math pipelines. Each Tensor Core performs D = A x B + C, where A, B, C and D are matrices. A and B are half precision 4x4 matrices, whereas D and C can be either half or single precision 4x4 matrices. In other words, Tensor Core math can accumulate half precision products into either single or half precision outputs. The reason half precision is so attractive is that the V100 GPU has 640 Tensor Cores, so they can all be performing 4x4 multiplications all at the same time. The theoretical peak performance of the Tensor Cores on the V100 is approximately 120 TFLOPS. This is about an order of magnitude (10x) faster than double precision (FP64) and about 4 times faster than single precision (FP32).
In a training, using mixed precision training requires three steps:

1. Converting the model to use the float16 data type where possible;
2. Keeping float32 master weights to accumulate per-iteration weight updates;
3. Using loss scaling to preserve small gradient values.

There are some frameworks that support fully automated mixed precision training. They also support automatic loss scaling and master weights integrated into optimizer classes and automatic casting between float16 and float32 to maximize speed while ensuring no loss in task-specific accuracy.

Chapter 4

PyTorch and Distributed Training

One of the biggest problems with Deep Learning models is that they are becoming too big to train in a single GPU. If the current models were trained in a single GPU, they would take too long. In order to train models in a timely fashion, it is necessary to train them with multiple GPUs. We decided to use PyTorch to implement our Distributed Training. In fact, this framework provides a handy use without giving up performance.
In this chapter we describe briefly PyTorch and its main features then we will talk about distributed training and how it is implemented in our framework.

4.0.1 PyTorch

PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR). It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface. A number of pieces of Deep Learning software are built on top of PyTorch, including Tesla, Uber's Pyro and Catalyst. PyTorch provides two high-level features:

1. **Tensor** computing with strong acceleration via graphics processing units (GPU)

2. Deep **neural networks** built on a tape-based automatic differentiation system

A **Tensor** in PyTorch is a multi-dimensional matrix containing elements of a single data type. There are 9 types for CPU and 9 types for GPU.
Neural networks can be constructed using the torch.nn package. We have already described what a Neural Network is in the first chapter. We have to know that in PyTorch an nn.Module contains layers, and a method forward(input) that returns the output. PyTorch built two ways to implement distribute training in multiple GPUs: **nn.DataParallel** and **nn.DistribuitedDataParallel**. We must describe these two models to understand how distributed training exactly works.

DataParallel replicates the same model to all GPUs, where each GPU consumes a different partition of the input data. It is not difficult to implement and it can significantly accelerate the training process, but it does not work for some use cases where the model is too large to fit into a single GPU.
DistribuitedDataParallel (DDP) works with model parallel. DDP implements data parallelism at the module level which can run across multiple machines. Applications using DDP should spawn multiple processes and create a single DDP instance per process. DDP uses collective communications to synchronize gradients and buffers. More specifically, DDP registers an autograd hook for each parameter given by model.parameters() and the hook will fire when the corresponding gradient is computed in the backward pass. Then DDP uses that signal to trigger gradient synchronization across processes.
The recommended way to use DDP is to spawn one process for each model replica, where a model replica can span multiple devices. DDP processes can be placed on the same machine or across machines, but GPU devices cannot be shared across processes.
Although DistributedDataParallel is more complex to use and implement, it has some advantage compared to DataParallel:

- DataParallel is single-process, multi-thread, and only works on a single machine, while DistributedDataParallel is multi-process and works for both single- and multi-machine training. DataParallel is usually slower than DistributedDataParallel even on a single machine due to GIL contention across threads, per-iteration replicated model, and additional overhead introduced by scattering inputs and gathering outputs.
- If the model is too large to fit on a single GPU, you must use model parallel to split it across multiple GPUs. DistributedDataParallel works with model parallel; DataParallel does not currently. When DDP is combined with model parallel, each DDP process would use model parallel, and all processes collectively would use data parallel.

4.1 Distributed Training

As we said before, distributed training is used when models too big to be trained in a single GPU or to reduce training time.
First, let us go over how training a neural network usually works. There are four main steps for each loop that happens when training a neural network:

1. The forward pass, where the input is processed by the neural network
2. The loss function is calculated, comparing the predicted label with the ground-truth label
3. The backward pass is done, calculating the gradients for each parameter based on the loss (using back-propagation)

4. The parameters are updated using the gradients

In PyTorch we can do distributed training with DataParallel and DistribuitedDataParallel. Let us analyse how they work and their difference.
DataParallel helps distribute training into multiple GPUs in a single machine. There are a few steps that happen whenever training a neural network using DataParallel:

1. The mini-batch is split on GPU:0
2. Split and move min-batch to all different GPUs
3. Copy model out to GPUs
4. Forward pass occurs in all different GPUs
5. Compute loss with regards to the network outputs on GPU:0 and return losses to the different GPUs. Calculate gradients on each GPU
6. Sum up gradients on GPU:0 and use the optimizer to update model on GPU:0

All these passages are illustrated in the image below.

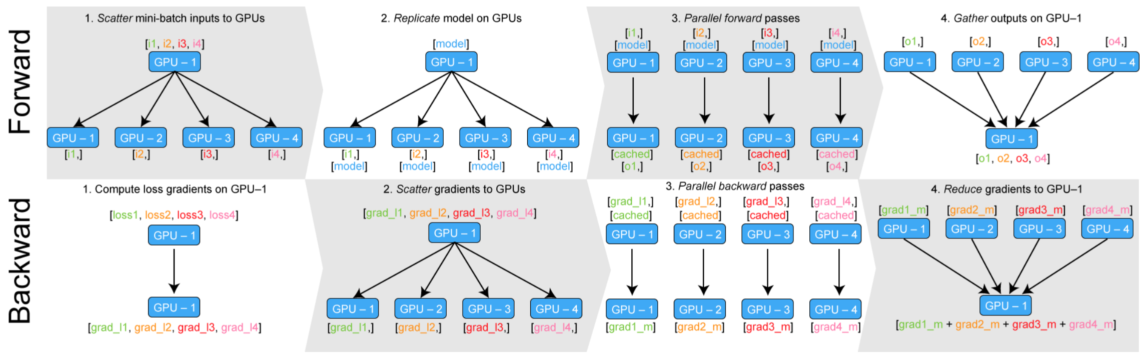

Forward
Backward
1. Scatter mini-batch inputs to GPUs
2. Replicate model on GPUs
3. Parallel forward passes
4. Gather outputs on GPU–1
1. Compute loss gradients on GPU–1
2. Scatter gradients to GPUs
3. Parallel backward passes
4. Reduce gradients to GPU–1
GPU – 1
GPU – 2
GPU – 3
GPU – 4
[i1, i2, i3, i4]
[model]
[cached]
[o1, o2, o3, o4]
[loss1, loss2, loss3, loss4]
[grad_l1, grad_l2, grad_l3, grad_l4]
[grad1_m + grad2_m + grad3_m + grad4_m]

Figure 4.1: Training with DataParallel

For **DistributedDataParallel**, the machine has one process per GPU, and each model is controlled by each process. The GPUs can all be on the same node or across multiple nodes. Only gradients are passed between the processes/GPUs.
During training, each process loads its own mini-batch from disk and passes it to its GPU. Each GPU does its forward pass, then the gradients are all-reduced across the GPUs. Gradients for each layer do not depend on previous layers, so the gradient all-reduce is calculated concurrently with the backwards pass to further alleviate the networking bottleneck. At the end of the backwards pass, every node has the averaged gradients, ensuring that the model weights stay synchronized.

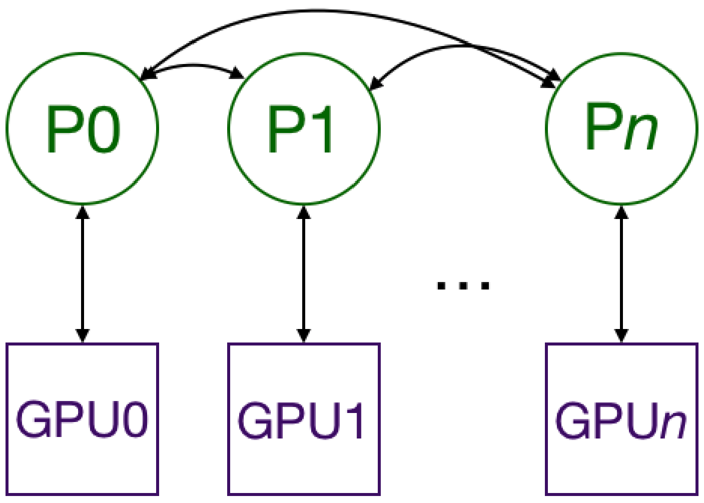


Figure 4.2: Processes with DDP

There are a few problems that might occur whenever running the same model in a few GPUs instead of one GPU. The biggest problem that can occur is that the main GPU may run out of memory. The reason for that is because the first GPU will save all the different outputs for the different GPUs to calculate the loss.
In order to solve this problem, and reduce the amount of memory usage, we use two techniques:

1. Reduce some parameters that increase memory consumption (e.g. batch-size)
2. Use Apex for mixed precision

The first technique is straightforward, and usually involves just changing one hyper-parameter.
The second technique, as we said before, means that we are going to decrease the precision of the weights that are used in the neural network, and therefore use less memory. Mixed-precision means you use 16-bit for certain things but keep things like weights at 32-bit.

Chapter 5

Marconi 100

MARCONI 100 is the new accelerated cluster based on IBM Power9 architecture and Volta NVIDIA GPUs, acquired by Cineca within PPI4HPC European initiative. This system opens the way to the pre-exascale Leonardo Supercomputer expected to be installed in 2021. It is available from April 2020 to the Italian public and industrial researchers. Its computing capacity is about 32 PFlops.
In this chapter we will describe the architecture of Marconi 100 to understand how we implement the distributed training in this cluster. Finally, we introduce SLURM: an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for Linux clusters. After this chapter we will have all the ingredients to understand how we implemented the distributed training in our system.

5.1 System Architecture

The M100's architecture is IBM Power 9 AC922, its internal network is Mellanox Infiniband EDR DragonFly+. The storage is 8 PB (raw) GPFS of local storage.
Marconi 100 has 980 nodes (and 8 for login) and each node has:

- Processors: 2x16 cores IBM POWER9 AC922 at 2.6(3.1) GHz
- Accelerators: 4 x NVIDIA Volta V100 GPUs/node, NVlink 2.0, 16GB
- Cores: 32 cores/node, Hyperthreading x4
- RAM: 256 GB/node (242 usable)

The peak performance of M100 (980 compute nodes + 8 login nodes) is about 32PFlops. The performance of the single node is 32 TFlops due to 0.8 for the CPU part and 7.8x4 for the four GPUs on the node. The theoretical peak performance is 988*(0,8+4*7,8) = 31,6 PFlop/s.
Marconi100 consists of 980 compute nodes and 8 login nodes, connected with a Mellanox Infiniband EDR network arranged into an architecture called DragonFly ++.

The login nodes and the compute nodes are the same. Each node consists in 2 Power9 sockets, each of them with 16 cores and 2 Volta GPUs (32 cores and 4 GPUs per node). The multi-threading is active with 4 threads per physical core (128 total threads – or logical cpus – per node).

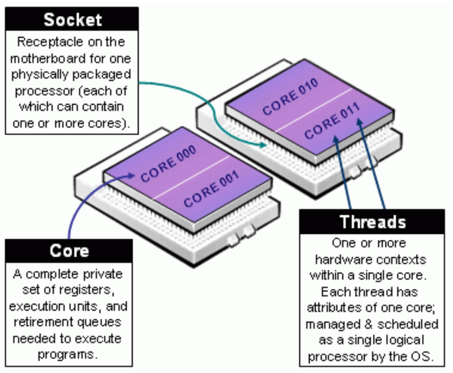

Socket
Receptacle on the motherboard for one physically packaged processor (each of which can contain one or more cores).
CORE 010
CORE 011
CORE 000
CORE 001
Threads
One or more hardware contexts within a single core. Each thread has attributes of one core; managed & scheduled as a single logical processor by the OS.
Core
A complete private set of registers, execution units, and retirement queues needed to execute programs.

Figure 5.1: The structure of the core

The cpus are numbered from 0 to 127 because of a hyperthreading. The two Power9 sockets are connected by a 64 GBps X bus. Each of them is connected with 2 GPUs via NVLink 2.0.

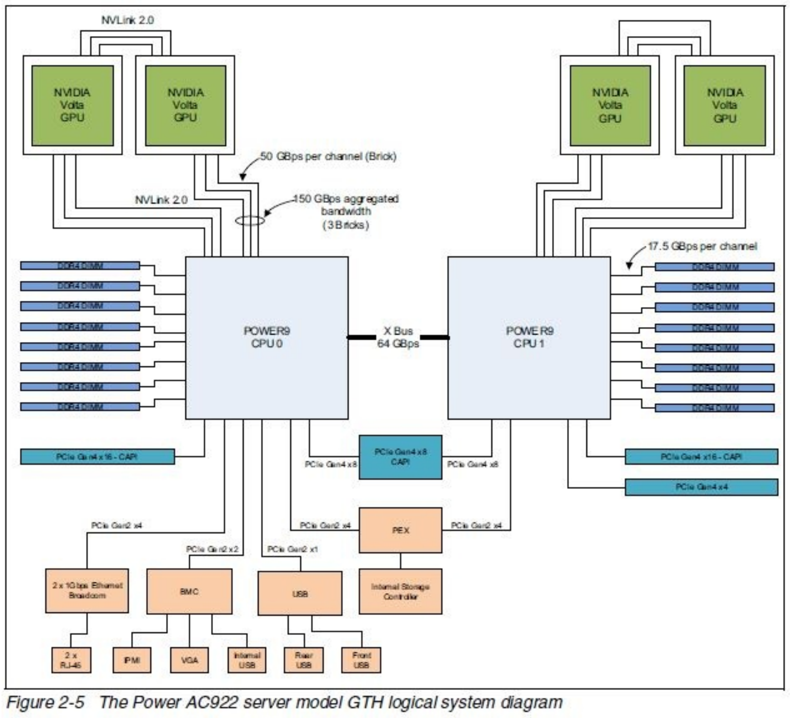


Figure 5.2: The structure of the node

The knowledge of the topology of the node is important for correctly distribute the parallel threads of your applications in order to get the best performances. The internode communications are based on a Mellanox Infiniband EDR network, and the OpenMPI and IBM MPI Spectrum libraries are configured so to exploit the Mellanox Fabric Collective Accelerators (also on CUDA memories) and Messaging Accelerators. NVIDIA GPUDirect technology is fully supported (shared memory, peer-to-peer, RDMA, async), enabling the use of CUDA-aware MPI.

5.2 SLURM

Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained. As a cluster workload manager, Slurm has three key functions. First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work.
Slurm consists of a slurmd daemon running on each compute node and a central slurmctld daemon running on a management node (with optional fail-over twin). The slurmd daemons provide fault-tolerant hierarchical communications. The user commands include: sacct, salloc, sattach, sbatch, sbcast, scancel, scontrol, sinfo, sprio, squeue, srun, sshare, sstat, strigger and sview. All the commands can run anywhere in the cluster.

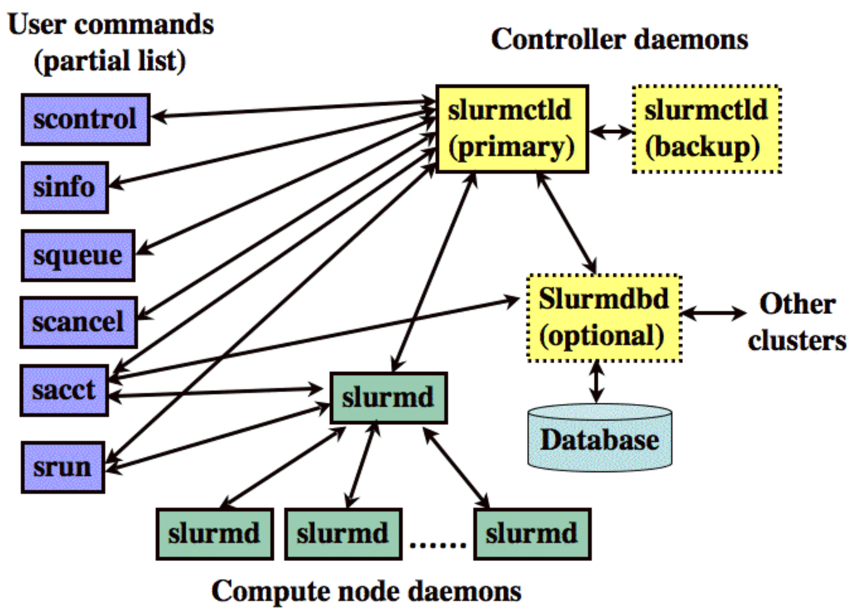


Figure 5.3: SLURM scheme

The entities managed by these Slurm daemons, shown in Figure 2, include nodes, the compute resource in Slurm, partitions, which group nodes into logical (possibly overlapping) sets, jobs, or allocations of resources assigned to a user for a specified amount of time, and job steps, which are sets of (possibly parallel) tasks within a job. The partitions can be considered job queues, each of which has an assortment of constraints such as job size limit, job time limit, users permitted to use it, etc. Priority-ordered jobs are allocated nodes within a partition until the resources (nodes, processors, memory, etc.) within that partition are exhausted. Once a job is assigned a set of nodes, the user can initiate parallel work in the form of job steps in any configuration within the allocation. For instance, a single job step may be started that utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation.
In Marconi 100, a serial (or parallel) program, also using GPUs and needing more than 10 minutes can be executed interactively within an "Interactive" SLURM batch job.
A request for resources allocation on the compute nodes is delivered to SLURM with the salloc/srun commands, the request is queued and scheduled as any other batch job but, when granted, the standard input, output, and error streams of the interactive job are connected to the terminal session from which the request was launched. SLURM automatically exports the environment variables you defined in the source shell, so that if you need to run your program in a controlled environment (i.e. specific library paths or options), you can prepare the environment in the origin shell being sure to find it in the interactive shell.
As usual on HPC systems, the large production runs are executed in batch mode. This means that the user writes a list of commands into a file and then submits it to a scheduler (SLURM for Marconi100) that will search for the required resources in the system. As soon as the resources are available the script is executed and the results and sent back to the user. We will analyse our batch scripts in the next chapter.
Using SLURM directives you indicate the account number (-A: which project pays for this work), where to run the job (-p: partition), what is the maximum duration of the run (–time: time limit). Moreover, you indicate the resources needed, in terms of cores, GPUs and memory. It's important to remember that in Marconi 100 each node exposes itself to SLURM as having 128 (virtual) cpus, 4 GPUs and 246.000 MB memory. SLURM assigns a node in shared way, assigning to the job only the resources required and allowing multiple jobs to run on the same node/nodes. If you want to have the node/s in exclusive mode, use the SLURM option "–exclusive" together with "–gres=gpu:4".

Chapter 6

Experimental Results

After analysing Marconi100 with its job scheduling system SLURM and PyTorch, we have all the elements we need to understand how we implemented and ran the distributed training.
First, we describe the training and the launching script and how they work. In the second part we analyse all the data collected in our experiments. The aim is to find how some parameters (e.g. number of GPUs, number of nodes, number of workers) can influence the epoch time and the best configuration possible.

6.1 Implementation

The implementation is divided in two parts: training script and launching script. The first is written in Python and uses PyTorch with all the elements described in the preview chapters, the second is written in bash and uses SLURM.

6.1.1 Training

The first part is dedicated to the arguments parsing. In this part we used the python library *argpars*. The main parameters requested are:

- The dataset's path
- Batch-size: the number of samples to work, for each GPU, through before updating the internal model parameters
- Number of workers
- Learning-rate: we gave its definition in the first chapter
- World-size: the total number of processes to run, which is equal to the total number of GPUs times the number of nodes
- Number of epochs: the number times that the learning algorithm will work through the entire training dataset.

After that, we create the Dataloader (a Python iterable over a dataset) for the training and evaluating dataset. At this point, we can create our distributed model with Distributed-DataParallel:

```
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.gpu])
```

Now we have all we need to start the training loop. For each epoch, each process loads its own mini-batch from disk and passes it to its GPU. Each GPU does its forward pass, then the gradients are all-reduced across the GPUs. Gradients for each layer do not depend on previous layers, so the gradient all-reduce is calculated concurrently with the backwards pass to further alleviate the networking bottleneck. At the end of the backwards pass, every node has the averaged gradients, ensuring that the model weights stay synchronized.

6.1.2 Launch

We created two scripts to run the training: one for single node and the other for multiple nodes. In each script we must import all the modules we need for distributed training with the command:

```
module load profile/deeplrn autoload pytorch
```

In a single node, to run the distributed launch on multiple GPUs, we need to use the command srun from SLURM

```
srun -N1 --cpus-per-task=32 --partition=m100_usr_prod --gres=gpu:volta:"$GPU"
-A PHD_aimageth --time=02:00:00 python -m torch.distributed.launch
--nproc_per_node="$NPROC" --use_env train.py --epochs “$EPOCHS” --world-size
"$NPROC" --workers "$WORKERS"
```

To run this command, we need to specify: the number of cpus requested, the Marconi100's partition, the number of GPUs requested, the project account and the maximum time.
In multiple nodes, the distributed launch is executed in batch mode. This means that we have to write a list of command into a file and then submit it to the SLURM scheduler that will search for the required resources in the system. As soon as the resources are available the script is executed and the results are sent back to us.

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32
#SBATCH --gres=gpu:4
#SBATCH --partition=m100_usr_prod
#SBATCH -A PHD_aimageth
#SBATCH -t 02:00:00
```

```
#SBATCH --output=log/prova_out.txt
#SBATCH --error=log/prova_err.txt
#SBATCH --nodelist=r244n01,r244n15,r244n17,r244n20,r245n01,r245n03,r246n20
```

This is the example of the batch script in 8-nodes configuration. We will show all the configuration and their performance in the results part.
In this way, we ask to the scheduler 8 tasks, one for each node, and 8 nodes. For each node we request all the 32 cpus and all the 4 GPUs. Then in the multi-node script we write as many srun commands as the required nodes:

```
srun -N1 -n1 -w r244n01 --gres=gpu:4 python -m torch.distributed.launch
--nproc_per_node=4 --nnodes=8 --node_rank=0 --master_addr="$MAST_ADDR"
--master_port=35000 --use_env train.py &

srun -N1 -n1 -w r244n15 --gres=gpu:4 python -m torch.distributed.launch
--nproc_per_node=4 --nnodes=8 --node_rank=1 --master_addr="$MAST_ADDR"
--master_port=35000 --use_env train.py

wait
```

The difference from the single-node script is that here we have to insert the requested node (e.g. r244n01), the node rank and the IP address of the master node,

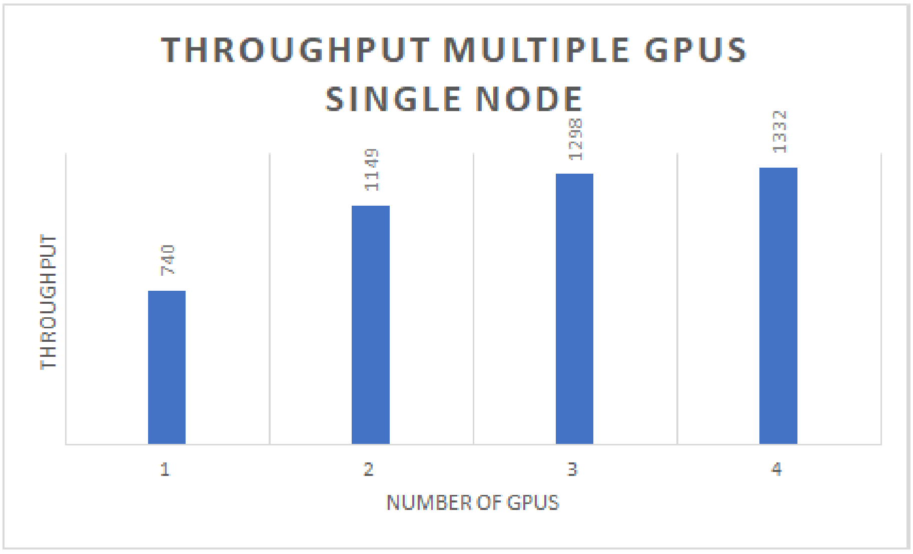
6.2 Results

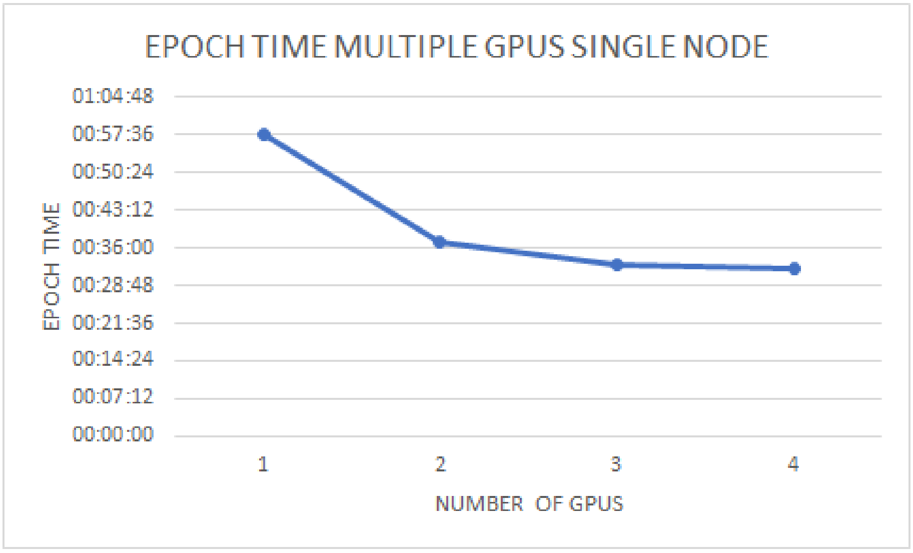
Our main goal is to minimize training time. We can achieve this result in two ways: reducing epoch time or increasing precision (to reduce the number of the epochs required). Reducing the epoch time is the most impact way initially. We can reach this goal in many ways: all of them are described in the following sub-paragraphs.

6.2.1 Multiple GPUs in a Single Node

The epoch time in a single node with a single GPU is 00:57:46. With this configuration we cannot use DistributedDataParallel and the single working GPU must process all the 2.8 million images of the training dataset. All these experiments are done with the same number of workers (16) and the same batch-size.
Using 2 GPUs, the epoch time is 00:37:12. The speedup is 1.55 and there is a 64% decrease in time. The ideal speedup is 2, we have this difference for two reasons.
First, with a distributed launch we have an overhead introduced by the communication process between GPUs.The overhead becomes stronger in the case of 3 and 4 GPUs. In fact, with 3 GPUs the epoch time is 00:32:57 and with 4 is 00:32:06. With 3 GPUs the speedup is 1.75 instead the ideal 3.0 and with 4 GPUs is 1.8 instead of 4.0. Second, the

workers, that are constantly set on 16 for all experiment in this group, cannot feel all the batches of data when the number of requested GPUs is too high.

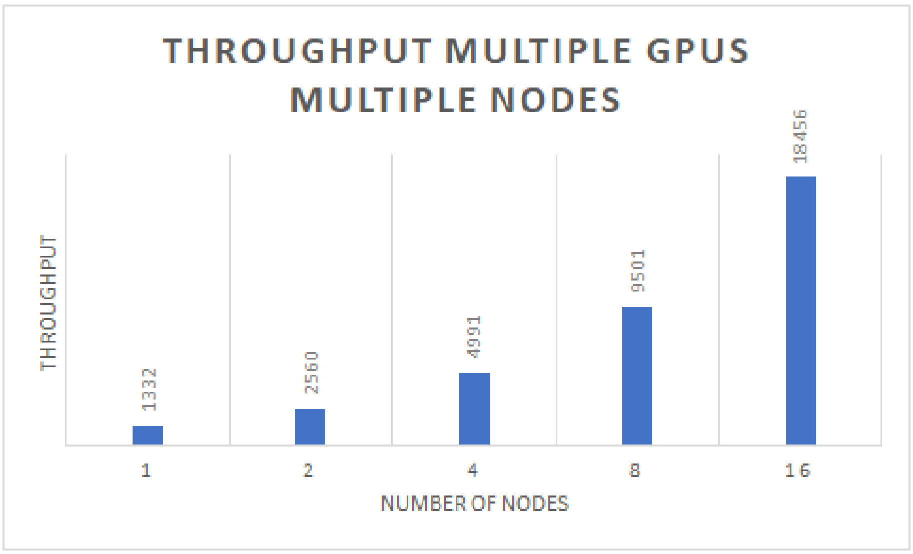


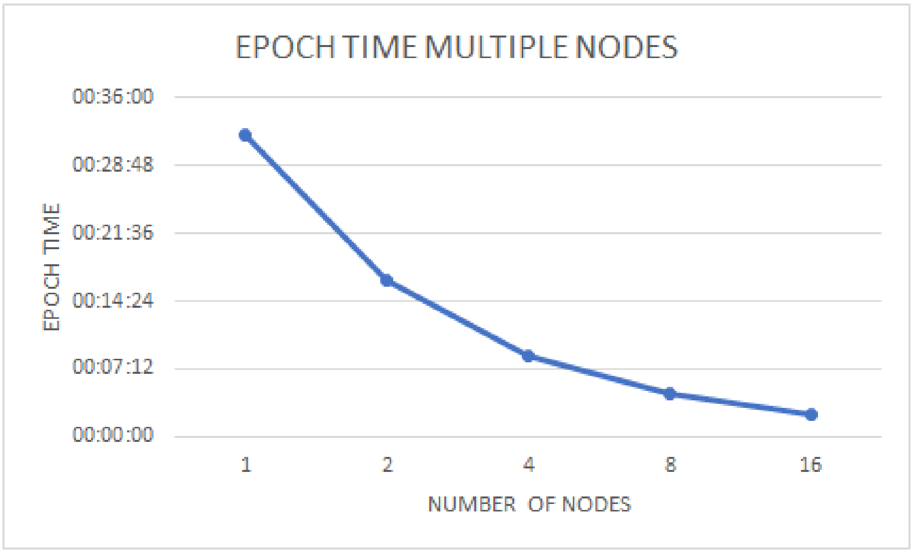


6.2.2 Multiple GPUs in Multiple Nodes

The best performance achieved in a single node is an epoch time of 00:32:06. Scaling in multiple nodes is different than scaling the number of GPUs in a single node because in each node you can require all its 32 cpus. If we define a computational unit an entire node (with 32 cpus and 4 GPUs), scaling in two nodes means duplicating the computational unit; the situation is different in a single node where the 32 cpus are divided by all the GPUs requested.
With 2 nodes the epoch time is 00:16:42. The speed up is 1.92, very close to the ideal 2.0 one. The same situation is similar in the other cases.
With 4 nodes the epoch time is 00:08:34 with a speedup of 3.74.

With 8 nodes the epoch time is 00:04:30 with a speedup of 7.13.
With 16 nodes, the maximum possible number in our SLURM partition, the epoch time is 00:02:19 with a speedup of 13.86.
The last result permits to save a lot of time. A complete training involves, on average, 90 epochs. In a standard configuration (1 GPU and 1 worker) the epoch time is 00:57:46, this means that a complete training lasts 3 days 14 hours and 36 minutes. With 16 nodes (64 GPUs and 512 cpus) the epoch time is 00:02:19, this means that a complete training lasts 3 hours and 28 minutes.

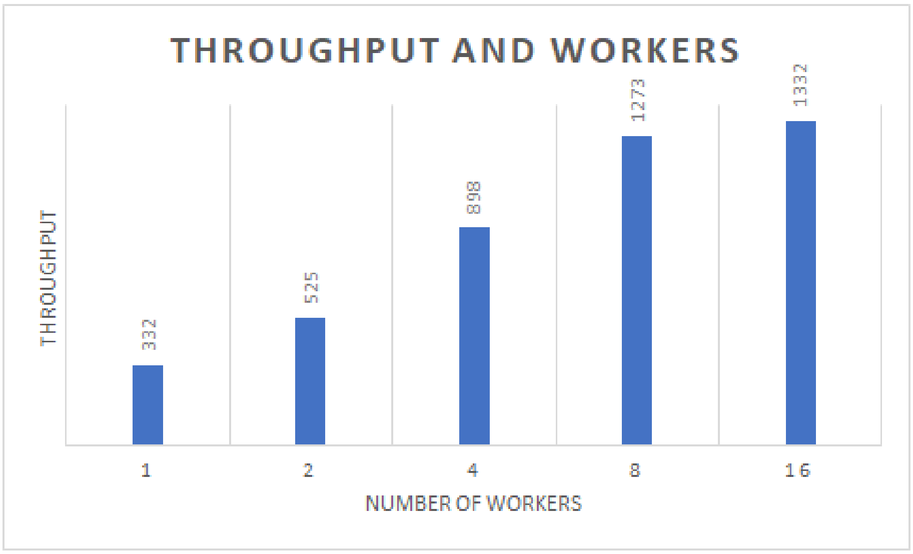


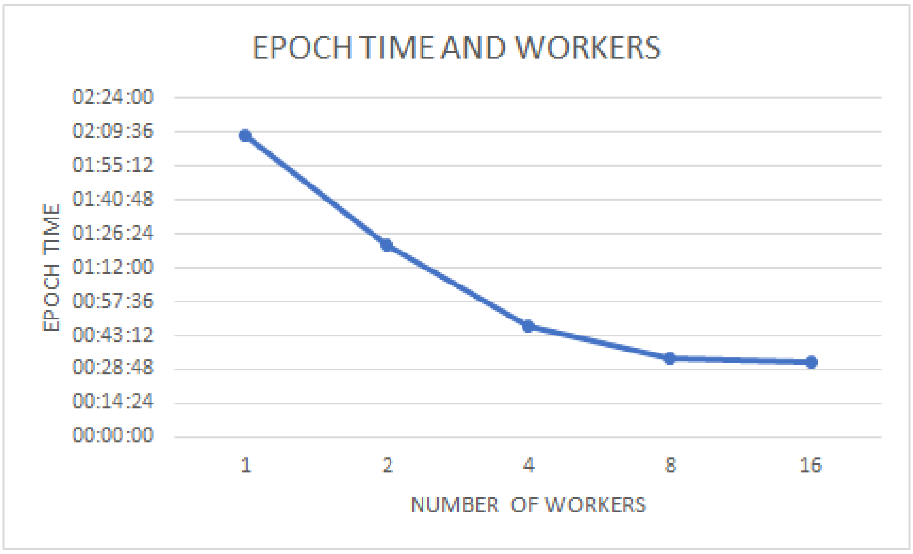


6.2.3 Multiple Workers

Workers are sub-processes used for data loading. By default, the number of workers is set to zero, and a value of zero tells the loader to load the data inside the main process.
This means that the training process will work sequentially inside the main process. After

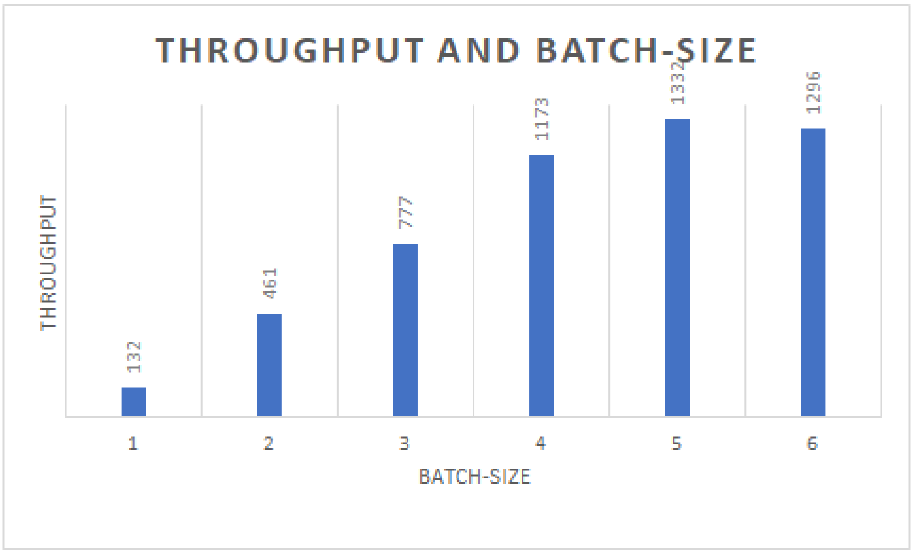
a batch is used during the training process and another one is needed, we read the batch data from disk. Now, if we have a worker process, we can make use of the fact that our machine has multiple cores. This means that the next batch can already be loaded and ready to go by the time the main process is ready for another batch. This is where the speed up comes from. The batches are loaded using additional worker processes and are queued up in memory.
Once reached the number of workers needed to keep the queue full of data for the main process, then adding more batches of data to the queue is not going to do anything. Just because we are adding more batches to the queue does not mean the batches are being processes faster. Thus, we are bounded by the time it takes to forward and backward propagate a given batch.
In our experiments we can see the speedup given by the increase of the number of workers: all these tests have 4 GPUs, batch-size=32 and same learning rate.
With only 1 worker the epoch time is 02:08:24. The trend is described in the graphics below.
With 2 workers the epoch time is 01:21:22 with a speedup of 1.58.
With 4 workers the epoch time is 00:47:34 with a speedup of 2.70.
With 8 workers the epoch time is 00:33:35 with a speedup of 3.82.
With 16 workers the epoch time is 00:32:06 with a speedup of 4.00.
With a number of workers greater than 16, we cannot see time improvements. We can affirm that the number of workers needed to keep the queue full of data is 16 and it is the best value of this parameter for our configuration.

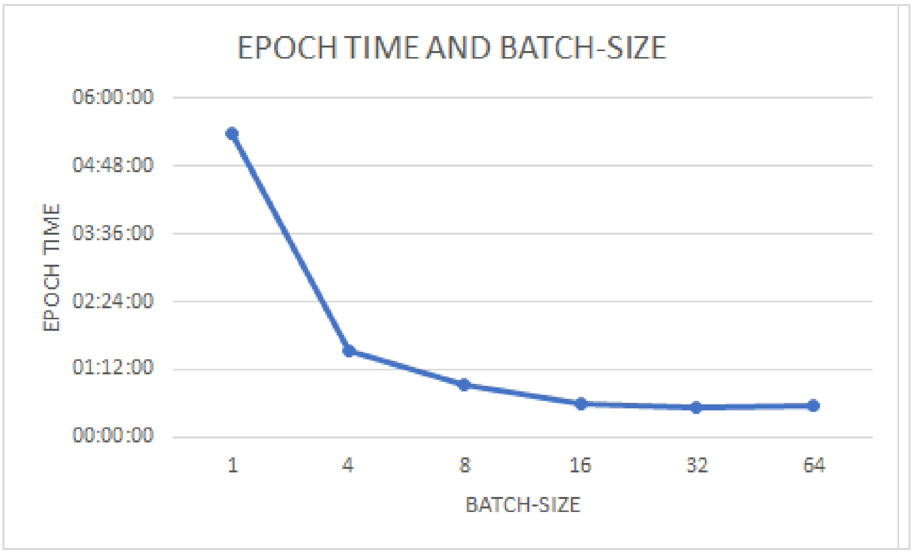




6.2.4 Batch-size

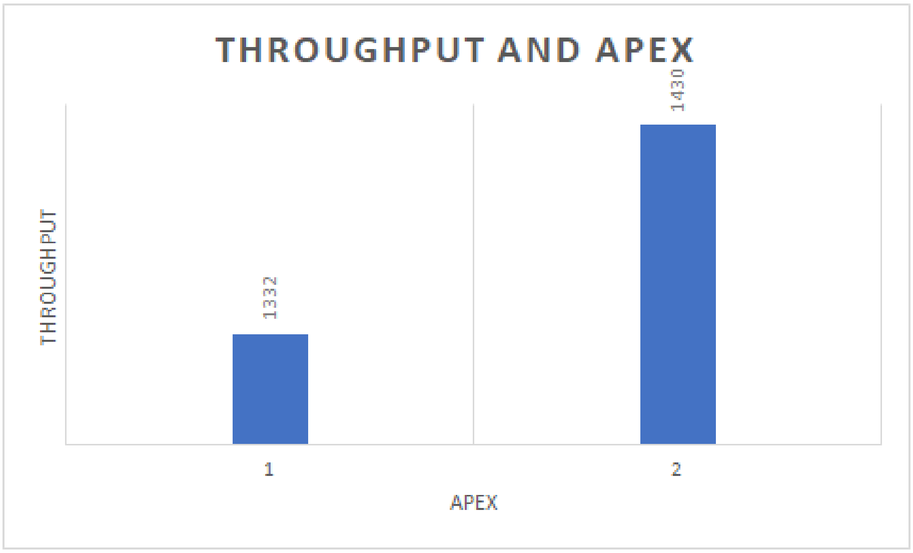
It is very important to find the perfect batch-size according to the number of workers available. The number of workers that to achieve the best performances is 16.
If the batch-size is too large, the GPUs have to wait the workers to have the batches full of data and we there is a waste time. If the batch-size is too small, not all workers are necessary to keep the queue full of data and the cpus are not used as maximum as possible.
We can see in our experiments that the perfect batch-size is 32.

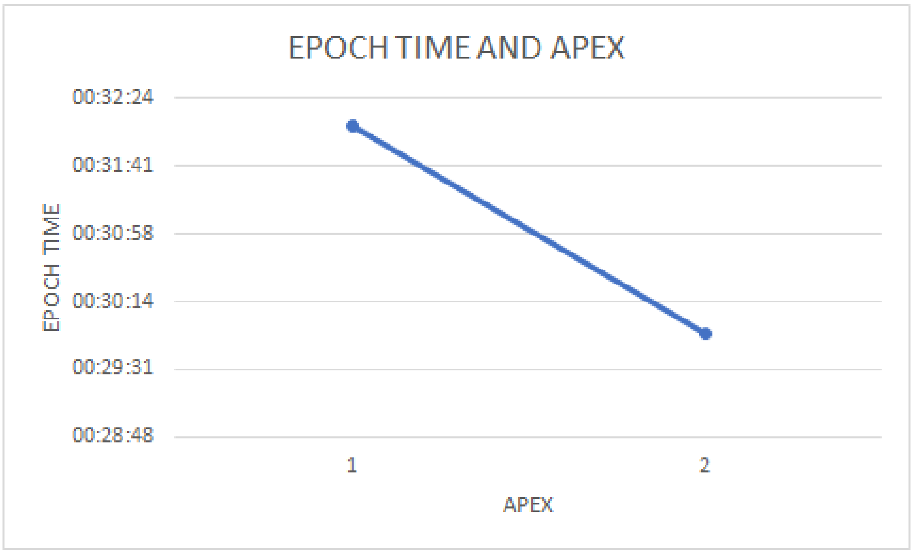




6.2.5 Apex

We have already described Mixed Precision Training in the preview chapters. Our experiment is done in a single node with 4 GPUs, 32 cpus, 16 workers and 32 of batch-size.
Without Apex the training time is 00:32:06.
With Apex the training time is 00:29:54 with a 6.9% decrease of time.




EPOCH TIME AND APEX
00:32:24
00:31:41
00:30:58
00:30:14
00:29:31
00:28:48
EPOCH TIME
1
2
APEX

Chapter 7

Conclusion and Future Work

The work in this thesis describes the benefits of Distributed Training. The new complex multi-layer structured CNNs require a lot of training time. As we have seen before, thanks to Distributed Training, we can reduce the classification training time of the Resnet18 on ImageNet from 3 days, 15 hours and 36 minutes (with a single GPU) to 3 hours and 28 minute (with 64 GPUs and 512 cpus).
For future work, it could be introduced a study on the precision of the model in order to achieve the best possibile performance by finding the best trade-off between the speedup of the epoch time and the increasing of the precision. In addition, the training script could be optimized with Horovod, a framework used for optimization of distributed training.

Bibliography

[1] Karanbir Singh Chahal, Manraj Singh Grover, and Kuntal Dey. A hitchhiker's guide on distributed training of deep neural networks. *CoRR*, abs/1810.11787, 2018.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[3] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

[4] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[5] Farhana Sultana, Abu Sufian, and Paramartha Dutta. Advancements in image classification using convolutional neural network. *CoRR*, abs/1905.03288, 2019.