Glasses - A scalable CNN implementation for image classification (from scratch)

Alessandro Ballerini

alessandro.ballerini1@studenti.unimi.it, Matr: 927412

***Abstract*—Machine learning model's effectiveness is strictly related to the data they are trained with, and the quality and quantity of data directly impact the results of the produced model. Dealing with the large amounts of data required for some models is not trivial and requires specific programming techniques and algorithms. In this paper, we will discuss a way of training a feedforward neural network across a distributed system to be able to deal with massive datasets. The proposed solution utilizes the Mapreduce programming techniques; it requires that small partitions of the dataset are processed on copies of the neural networks in different machines of a cluster. The outputs of the backpropagation phase of the training are then used to update the original neural network. The process is repeated till the network is trained.**

I. Introduction

Generic fully connected neural networks tend to perform poorly when dealing with images since the number of connections does not scale well with the size of the pictures. Convolutional neural networks are the standard tools for image recognition and classification. Assuming that the inputs are images allows the network to better adapt to its structure. Also, neurons in a convolutional layers are connected to a small region of the input volume and shares parameters with other neurons of the same volume depth. This is a major advantage over densely connected neural networks models (e.g. Multilayer Perceptrons) since the number of trainable features is drastically reduced. Despite their great advantages in handling images, convolutional neural networks still require examples to train, and the more complex is the task, the more data is required to accomplish it. Tasks that require a vast amount of training data are not difficult to imagine, and neither are problems that require more data than a single commodity computer could handle. Solutions to handle such kind of problem belong to the sphere of big data, where techniques and methodologies are developed with the exact purpose of process and analyze amounts of informations otherwise intractable.

In this paper I will propose and discuss a solution for the task of image classification using a convolutional neural network built from scratch and distributed computing techniques; the dataset will contain images of people with or without a pair of glasses, and the objective will be to classify them accordingly, assuming that the amount of training data would exceed the possibility of a single commodity computer.

II. Dataset

The dataset is generated by a Generative Adversarial Neural Network (GAN) using a 512 number latent vector. The image content represents people's faces and the objective is to determine whether a person is wearing glasses or not. The dataset provides 5000 PNG images of size 1024x1024x3 pixels. 4500 of those images are classified with a binary label that can be used for supervised learning. Both latent vectors and images are provided. Only the images will be used to train the model in this project.

A. Data preprocessing

Due to the nature of the task, images of this size (1024x1024x3) are not helpful since we do not need all the details they provide, and conversely, such large dimensions would slow down the training process. For this reason, images have been scaled down to the size of 80x80x3 pixels. The chosen size is still enough to allow a human observer to identify whether or not the person is wearing glasses or not, and so should be large enough for our model to perform the same task.

Standardization has been then applied to the images by centering the mean of each feature (pixel) to zero and dividing by the standard deviation. Another option would be to apply

normalization to the picture's pixels to have values that range within the [0,1] interval. Empirical testing on the dataset shows that both the options provide a huge improvement in the model training, with the first option suits slightly better this particular case.

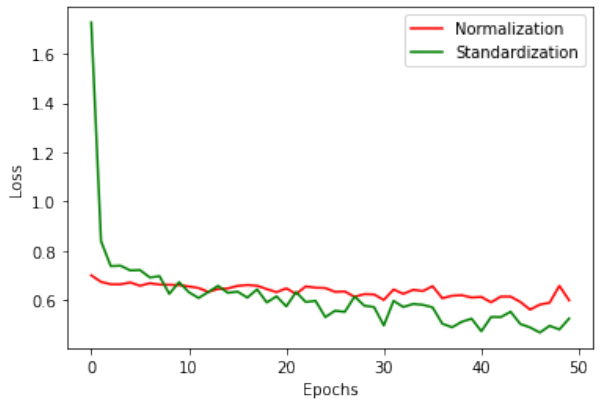

Fig. 1: Standardization and normalization in comparison during network training.

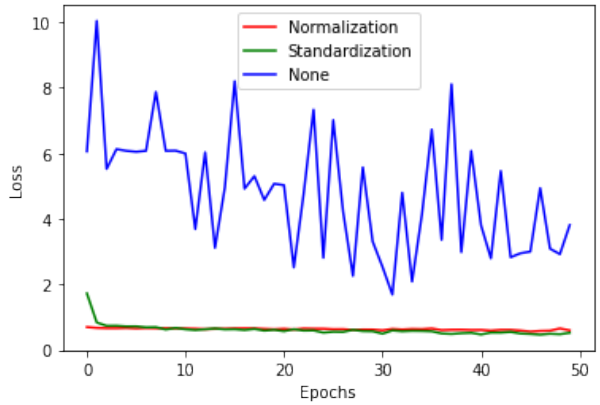

Fig. 2: Network training with no input standardization nor normalization.

III. Scalability

When the resources of a single computer are not enough for storing or computing the whole dataset, we enter the context of big data. Dealing with massive amounts of data is not an easy task, and it requires specialized hardware and programming techniques. One of the options at our disposal is to multiply the number of available computing units and share the computation among them. Focussing on raw numbers of multiple "cheaper" devices, instead of few high-performance ones has several advantages:

- it allows us to avoid single points of failure and makes the system more robust against hardware malfunctioning.
- reduces hardware costs since cutting-edge specialized technology is more expensive than multiple commodity devices.
- allows parallelization, which allows us to bypass physical limits on hardware performance.

Dealing with a huge amount of data, the first of our problems is how to store and handle our datasets. Distributed file systems help us organizing and accessing data shared between multiple devices connected to a network. Distributed datasets require rethinking algorithms to work in a distributed environment, and appropriate programming techniques, such as the MapReduce framework. For our project, we will use Hadoop and its distributed file system (HDFS), to handle the data and Spark to handle the MapReduce programming technique.

A. Map-Reduce

Mapreduce programming technique is suited to handle large datasets on distributed file systems. Map-Reduce is composed of different steps: First, the dataset is divided into splits of equal size and scattered between the workers' nodes. Each worker then applies the map function to its local data producing a key-value output. The result then is redistributed across the workers based on the keys values. Then the output is processed in parallel in the final reduce operation, and the computation ends. There may be some differences between map-reduce implementations.

B. Algorithm description

The solution proposed in this paper for the image classification task makes use of the MapReduce technique to handle large amounts of input data. The objective is to use the computation capability of several workers to parallelize the training of a single neural network. The algorithms developed would allow parallelizing any feed-forward neural network; the convolutional network architecture described afterwards is one possible application but does not represent a limit to the possible applications. The core idea behind this method is that when applying (batched) gradient descent, the data points submission and gradient calculation can be parallelized inside the scope of a single epoch; since a change in the network's state is performed only when all the data points are processed, each of them is submitted to a network that is in the same exact state as all the others. In our algorithm, a master node divides the training set into chunks that are sent to different worker nodes when a Map function can be applied. The Map function consists of computing the forward and backward training phase on a copy of the original neural network that has been sent to each worker with the data. The output of the Map function consists of the gradients of the weights calculated during the backward phase. Those are summed together in the Reduce operation and the produced output is sent back to the master node, which will perform the network weights update, completing an epoch of the training.

IV. Model Implementation

CNN networks make us of different types of layers. In this section, we will take an in-depth look at the layers utilized in the model implementation.

A. Convolutional layers

Convolutional layers are at the core of a convolutional neural network and serve the purpose of features extraction; they are composed of sets of learnable filters (or kernels) that are convolved over the input volume during the forward phase of the training. Filters have a depth equal to the input volume they are applied to and common filters height and width are 3x3, 5x5, and 7x7. During the convolution, the dot product is computed between filters and the input volume at every position. The output of the forward phase of a Conv layer is a volume which has for depth the number of filters employed in the convolution. Width and height depend on a series of parameters such as: the input volume's sizes, filter's sizes, stride, and padding, according to the following formulas:

$W_2 = (W - F + 2P)/S + 1$
$H_2 = (H - F + 2P)/S + 1$
$D_2 = K$

Where: WxHxD is the volume size, K is the number of filters, FxF is the filter's shape, S is the stride, P is the padding and the new volume size is W2xH2xD2. The stride value represents the size of the shift applied during the convolution. The padding consists of additional pixels on the outline of the input volume used to guarantee that the output will maintain the same width and height.

B. Pooling layers

Pooling layers perform a reduction in the size of the input volume for the width and height dimension, leaving the depth of the volume unaltered. This is useful to remove noise from the volumes and accentuate the features while reducing the computation effort. It also helps in reducing the number of parameters and therefore may help in controlling overfitting. The pooling technique adopted is MaxPooling: Given an input volume, it creates a new one by applying a Max operation for each non-overlapping volume area of size 2x2. Different filters shapes and stride values can be applied according to the case taken into account.

C. Fully connected layers

Fully connected layers are layers in which each neuron is connected to all the neurons in the previous and following layer, they are used to implement multi-layer perceptrons (MLP). They serve the purpose of classifying the features extracted in the precedent layers. They can be considered "structure agnostic" meaning that they do not need special assumptions about the input they are fed with. In the context of a convolutional neural network, fully connected layers are placed after the convolutional layers and eventual pooling layers, as the last layers of the whole network. The fully connected layer performs a dot product between its weights and the input values then adds the bais values. An activation function usually follows every fully connected layers.

$$out = x_1 w_1 + x_2 w_2 + ...x_n w_n + b$$

Where $x_1, x_2, ..., x_n$ represents the layer inputs, $w_1, w_2, ..., w_n$ represents the layer's weights and b represents the bias.

D. Flattening layer

One flattening layer is placed between the convolutional/pooling layers and the fully connected layers to adapt the output of the firsts to the input shape required by the former. The flattening layer converts the input matrix to a 1-dimensional array.

E. Dropout layer

Dropout is a technique used to avoid overfitting. It consists of temporarily disabling some randomly selected neurons. This alters the network structure at each pass, adds noise to the training process, and makes the network more robust toward overfitting. Dropout is implemented by adding a layer that randomly sets some values to zero with a certain probability.

F. Activation Function

Both Conv layers and fully connected ones perform similar operations. Whether through convolution or not, a dot product is performed between the filters/weights and the layer's inputs. After that, a scalar value (bias) is added to the result. Both dot product and bias additions are linear operations, thus they alone are not effective when dealing with non-linearly separable problems. For this reason, in order to generate a non-linear output, a non-linear function is applied after each neuron. Between all the possible non-linear functions that may be employed for this task, the following are the ones that are more often utilized:

1) Sigmoid and Tanh:

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

The rage of the function is the [0,1] interval, whit the maximum increase for x=0. Sigmoid function output assumes an asymptotic behavior toward the value 1.

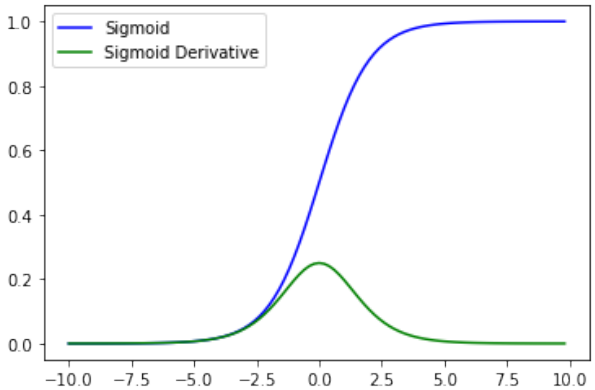

Fig. 3: Sigmoid and derivative.

$$Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-z}}$$

The rage of the function is the [-1,1] interval, the change in output accelerates close to x=0, which is similar to the Sigmoid function. It does also share its asymptotic properties with Sigmoid toward the value 1.

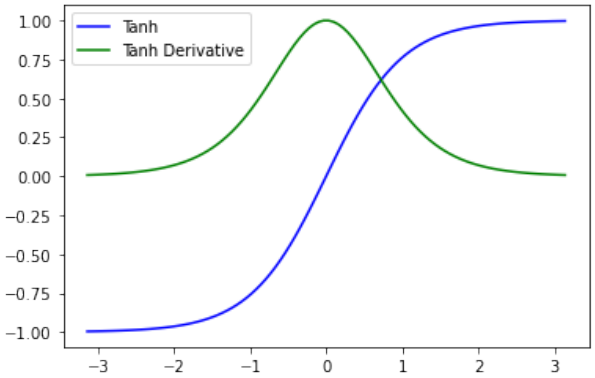

Fig. 4: Tanh and derivative.

The rage of the function is the [0,1] interval, whit the maximum increase for x=0. Sigmoid function output assumes an asymptotic behavior toward the value 1.

Issues with Sigmoid and Tanh: Both Tanh and Sigmoid tend to produce non-sparse models: Model training is facilitated when the model is less complex/sparse since simpler models tend to converge faster than complex ones. Complexity may be associated with the number of trainable features, thus fewer features mean a simpler model. If a model is able to ”deactivate” the less important neurons through training, its complexity will reduce with time, and convergence is facilitated. Due to the shape of the Tanh and Sigmoid function, is highly probable that will produce a non zero output value regardless of how small it may be. As a consequence, the model will tend to have more active but not influential neurons, which will slow down model convergence.

2) ReLU:

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

The Rectified Linear Unit (or ReLU) function is less sensitive to this issue and often a better choice for deep learning models.

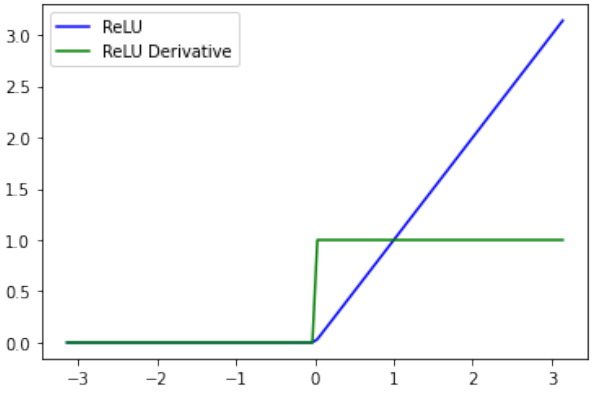

Fig. 5: ReLU and derivative

Since each value minor then zero comports the deactivation (output equal zero) of the neuron, ReLU function adoption favors model sparsity and eases convergence.

ReLU also helps with the vanishing gradients issue since the derivative is a constant value, either 0 or 1.

Lastly, ReLU is easier to compute than Sigmoin or Tanh, since it only needs to execute a max function between 0 and the input value.

Despite the advantages mentioned, the ReLU function also brings some challenges. The ease with which ReLU tends to ”deactivate” neurons can cause a problem called ”dying ReLU”, which occurs when too many neurons with large negative values are not able to recover from being stuck at 0. This occurs when neurons are not initialized properly or data is not normalized, and as a consequence, the network may stop learning.

A possible solution to the ”dying ReLU” problem can be the introduction of a slight variation in the ReLU function itself. This ReLU variation is called LeakyRelu because it allows a slight but meaningful information leak for values minor than zero, which should help the gradient recover from being stuck on zero.

$$LeakyRelu(x) = max(x * 0.01, x)$$

3) Softmax: The activation function of the neural network’s last layer plays an important role. Its output is forwarded to the loss function that has to assess the ”quality” of the whole network’s prediction. If we were to use the rectified linear activation function, negative input values would be clipped off, resulting in a loss of meaningful information. Also, we would not be able to quantify the degree of the error of the network since the output values would be independent of each other. The optimal solution would be to get the output of the neural network as a probability distribution; this would help to quantify the error, relativizing each output value to the correct one. The softmax functions solve those and other problems: It handles negative values using the natural exponential function applied to each input and then normalized them to obtain a probability distribution of the neural network according to its actual state and the submitted input values.

$$Softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} = \hat{y_i}$$

G. Loss Function

Loss function is the last element of a feedforward neural network and has the role to calculate how much the produced output (prediction) differs from the desired outcome. The choice of the loss function is crucial in the building of the network since it is at the basis of the learning process.

Considering the output of the neural network as a probability distribution, and dealing with a classification problem, our best choice for the loss function of our network is the categorical cross-entropy function.

Cross-entropy calculates a score that represents the average difference between two probability distributions for all the classes/categories involved.

$$Loss = -\sum_{i=1}^{N} y_i * log(\hat{y_i}))$$

Where N represents the number of outputs, y_i represents the correct probability for the i-th class for the current entry and

$\hat{y_i}$ represents the predicted probability of the network for the i-th class. Since we consider the correct label as our target distribution, the formula gets even simpler; all y_i values are zeros except for the one that represents the correct class. This leaves us with:

$$Loss = -log(\hat{y_i})$$

V. Model Training

A. Backpropagation

After the forward phase, where an entry (or a batch) is inserted in the network as input, and the network prediction is evaluated by the loss function, it is time to improve the network by adjusting its weights. In order to accomplish this, the network should be able to assess how much every single weight is responsible for the current outcome, and what to do to reduce the error. We use the backpropagation algorithm to propagate the error calculated by the loss function to the previous layers and compute a new value for the weights that reduce such error. This process involves calculating the gradient of the loss function with respect to the weights of the network through the application of the chain rule. Each layer performs its own backpropagation phase, which depends on the function and calculation applied to that layer, and propagates the error to the previous layer.

1) Softmax and cross-entropy gradient calculation: Softmax and categorical cross-entropy together provide an easy-to-compute gradient computation due to the way the formula simplifies during the application of the chain rule:

Given the softmax formula seen in the previous section, we must distinguish two cases; when the output index is equal to the input index and when it is not:

if $i = j$:

$$\frac{d\hat{y_i}}{dx_i} = \frac{exp(x_i)\sum_k exp(x_k) - exp(x_i)exp(x_i)}{(\sum_k exp(x_k))^2} =$$

$$\frac{exp(x_i)}{\sum_k exp(x_k)}(\frac{\sum_k exp(x_k)}{\sum_k exp(x_k)} - \frac{exp(x_i)}{\sum_k exp(x_k)}) =$$

$$\hat{y_i}(1 - \hat{y_i})$$

if $i \neq j$:

$$\frac{d\hat{y_i}}{dx_j} = \frac{-exp(x_j)exp(x_i)}{(\sum_k exp(x_k))^2} = -\hat{y_i}\hat{y_j}$$

Derivative calculation for categorical cross-entropy function.

$$\frac{dL}{d\hat{y_i}} = -\frac{1}{\hat{y_i}}$$

Application of the chain rule:

$$\frac{dL}{dx_i} = \frac{dL}{d\hat{y_i}}\frac{d\hat{y_i}}{dx_i} =$$

As before, we must distinguish two cases:

if $i = j$:

$$\frac{dL}{dx_i} = -\frac{1}{\hat{y_i}}\hat{y_i}(1 - \hat{y_i}) = \hat{y_i} - 1$$

if $i \neq j$:

$$\frac{dL}{dx_i} = -\frac{1}{\hat{y_j}}(-\hat{y_i}\hat{y_j}) = \hat{y_i}$$

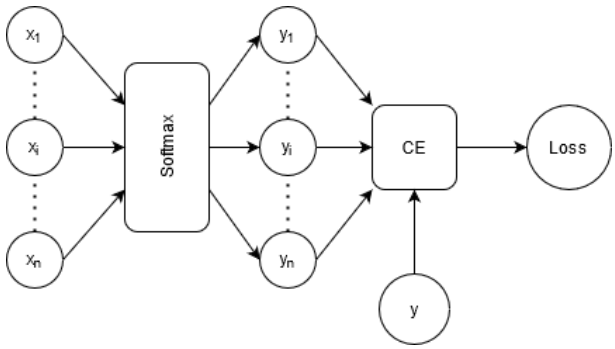

Fig. 6: Softmax and cross entropy network diagram.

B. Optimizer

The optimizer defines the strategy used by the network to update its weights and converge towards the optimal solution by minimizing the loss.

1) Gradient descent (GD): Gradient descent (or batched gradient descent) is one of the optimization algorithms used to train neural networks. It uses the negative gradient of the target function as a direction towards its local minima and approaches it through multiple steps. After the backpropagation phase, once the gradient for each of the weights is calculated, those are updated according to the following formula:

$$W_{new} = W_{old} - lr * \nabla f(W_{old})$$

Where W_{new} indicates the updated weight, lr indicates the learning rate, and $\nabla f(W_{old})$ indicates the gradient of the target function at its current state. The learning rate is a hyperparameter used to regulate the step size the function performs. If it is too small, the network may take a long time to reach its minimum. Conversely, a learning rate too big may cause the network to miss the minimum or keep "jumping" around it being unable to reach it.

In classical gradient descent, the network computes all data points in a single pass and then updates its weights. This operation requires the whole dataset to be loaded into the main memory at the same time to be computed. Since this requirement may not always be fulfillable, an alternative is to compute a single data point at a time and then update the weights. This variant of the original gradient descent algorithm is called Stochastic Gradient Descent.

2) Stochastic Gradient descent (SGD): SGD is a good alternative when dealing with larger datasets since it computes one data set at a time; it is also suitable for online learning. Due to the high variance frequent updates of the network SGD causes the objective function to be subjected to heavy fluctuations. While this may cause a slower convergence to a local optimum, it could also allow the network to "jump" to new potentially better local minima.

3) Mini-Batch Gradient descent (GD): Mini-Batch gradient descent represents a middle between classical gradient descent and stochastic gradient descent: Weight updates are performed after computing a subset (or mini-batch) of the whole dataset. By modifying the batch size one could approach the stability of the classic GD and at the same time obtain the advantages of the SGD.

C. Hyperparameters

Hyperparameters are variables that must be determined before starting the training of the model. They define the family of the algorithms that generate the model and heavily impact its ability to learn. The hyperparameters that will determine our model are the learning rate andthe batch size. Since an exhaustive search of the best combination of all the hyperparameters would be untractable we limit ourselves to subsets of the possible values for each hyperparameter.

- The learning rate is a value in the (0,1] interval. We will consider the subset of values 0.1, 0.01, 0.001, 0.0001.
- The batch size is a subset of the total number of data points. We will use the values of 16, 32, and 64 data points for each batch.

Another important aspect to take into account is the choice of the network architecture, intended as the number of layers, their type and shape, and their organization.

A CNN is usually structured as follow: An input layer followed by one or more convolutional layers, followed by one or more dense layers. Each convolutional and dense layer is always followed by a non-linear function (ReLU in our case).

In addition, pooling layers may be introduced after the convolutional layers, to form a more complex structure.

A common type of structure sees a pattern of few convolutional layers followed by a pooling layer, all repeated several times and followed by the dense layers.

After different tests, the following architecture was chosen as a starting point:

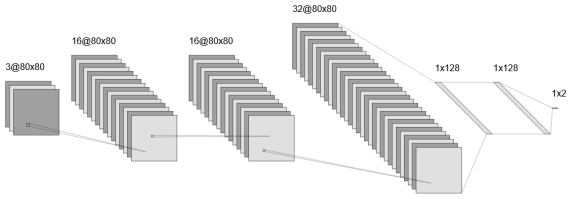

Fig. 7: Graphical representation of the structure of the convolutional neural network employed.

The input layer of shape 80x80x3 is followed by three Conv layers; the first two convolutional layers shape is 80x80x16 while the last convolutional layer shape is 80x80x32. A pooling layer (MaxPooling) is placed after the last convolutional layer. The pooling layer is followed by 3 dense layers; the first two dense layers have 128 neurons each while the last dense layer has only 2 neurons and is used as output layer. All the convolutional layers and dense layers are followed by a ReLU activation function. The first two dense layers are also followed by a dropout layer. The last dense layer's activation function is a Softmax function. It can be expressed as:

$INPUT- > [CONV- > RELU] * 3- > POOL- > [FC- > RELU] * 2- > FC- > SOFTMAX$

VI. Deployment

The project has been tested and executed using Google Cloud Platform on a cluster using the framework Spark for distributed computing.

The hardware at our disposal was:

- Master node (n2-standard-4 machine type) with 4vCPU, 16GB memory, 500GB primary disk, and 375GB local SSD.
- 5 x Worker nodes (n1-standard-4 machine type) with 4 vCPU, 15GB memory, 500GB primary disk, and 375GB local SSD.

VII. Results

After training the neural network over a dataset of 500 labeled images using 400 images for the training set and 100 images for the test set, according to the achieved results, we have identified as the best combination of parameters for the selected network architecture $lr = 10^-3$ and $batchsize = 16$.

The number of epoch used during training: 50.

lr	batch size	training loss	training acc	testing loss	testing acc
10^-4	16	0,502	0,765	0,535	0,730
	32	0,502	0,758	0,513	0,750
	64	0,506	0,773	0,560	0,680
10^-3	16	0,365	0,838	0,510	0,790
	32	0,368	0,835	0,689	0,720
	64	0,385	0,835	0,608	0,720
10^-2	16	1,709	0,623	1,709	0,680
	32	0,402	0,840	0,463	0,770
	64	0,637	0,648	0,610	0,640
10^-1	16	0,663	0,623	0,634	0,680
	32	0,663	0,623	0,633	0,680
	64	0,664	0,623	0,642	0,680

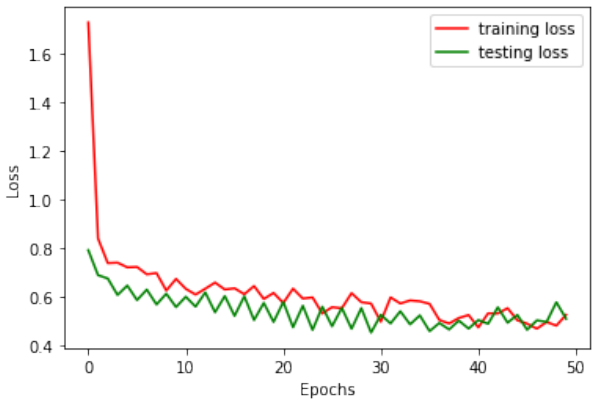

Fig. 8: Performance of the selected model during its training. Final testing loss value=0.50

After tuning the hyperparameters learning rate and batch size, some slight adjustments were made to the neural network architecture to improve its performances.

Two types of modification were performed separately on the neural network to see if they would cause any performance improvements: Adding more convolutional layers and adding more filters to the existing levels.

In the first case, three more convolutional layers and a pooling layer were inserted after the first set of Conv layers. The resulting structure can be expressed as follows:

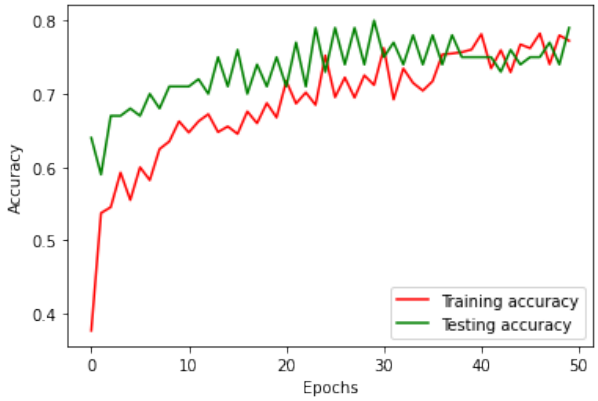

Fig. 9: Performance of the selected model during its training. Final testing accuracy value=0.79

	training loss	training acc	testing loss	testing acc
Original	0,365	0,838	0,510	0,790
I Variation	0.644	0.623	0.623	0.68
II Variation	0.474	0.765	0.509	0.8

$INPUT->[[CONV->RELU]*3->POOL]*2->[FC->RELU]*2->FC->SOFTMAX$ In this first case the no improvement was shown in the resulting model. Performance dropped in both loss and accuracy values.

In the second case, the filters of the convolutional layer were doubled while the overall structure remained the same. In this case, the model performed similarly to the original one. The graphs of accuracy and loss during the 50 epochs, however, showed a more "nervous" training behavior.

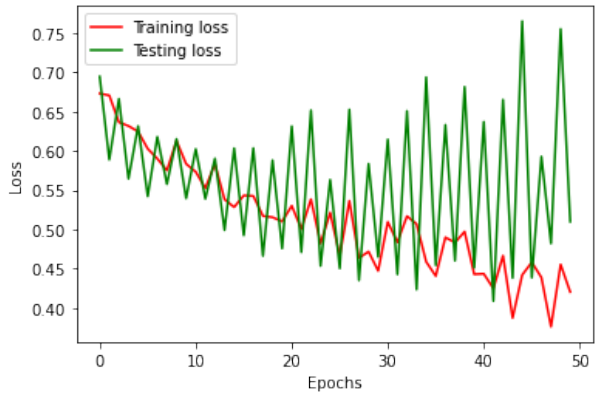

Fig. 10: Performance of the second model variation. Final testing loss value=0.509

The results show that the neural network is able to learn and, the scalability strategy allows to train a neural network dividing the load between the nodes of a cluster. This strategy also presents downsides; the higher is the number of epochs, the more are the quantity of data that has to travel through the network to the workers and back to the master. This is since the map-reduce operation is performed multiple times, and data is sent to the workers each of those times. By allowing the workers to cache the data to use for the following iterations would allow saving resources. It's worth noting also that the network itself is broadcasted for each iteration. As a consequence, the growth of the network also causes the growth of the amount of data that is sent to the workers and back to the master. According to the number of learnable parameters of the network, this may or may not be a problem, and in any case, is not dependant on the size of the dataset.

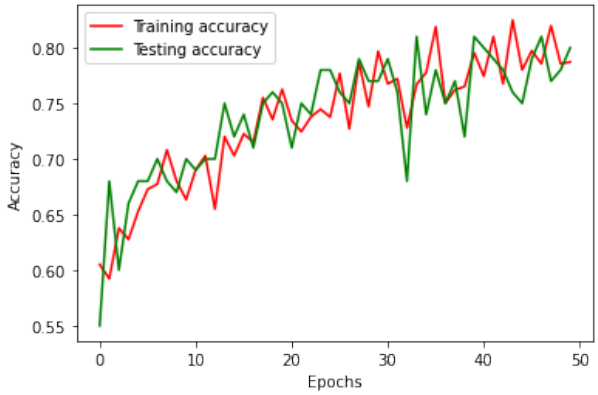

Fig. 11: Performance of the second model variation. Final testing accuracy value=0.80

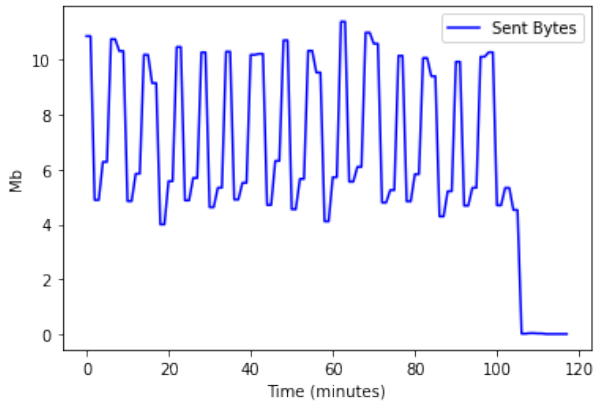

Fig. 12: Mb sent across the network during training

VIII. Conclusions

In this relation, we have described the functioning of a convolution neural network and proposed a possible strategy to handle massive datasets, by dividing the training of the network between the devices of a cluster using the map-reduce technique. The trained model learned correctly from the dataset and achieved an accuracy of 80% correct answers on the test set without bothering signs of overfitting. The neural networks seemed to give better results by keeping low the number of layers and increasing the number of filters. The reported results illustrated that despite the cluster success in training the network, the disadvantages of such a strategy may become overwhelming depending on the situation. The large amount of data shared between the nodes of the cluster may become the bottleneck of the training if not handled properly. Further development of this strategy should aim to reduce the amount of utilized bandwidth by caching part of the training data on the worker nodes. A possible solution could also be

avoiding merging the features gradients and train different simpler networks in the worker's nodes and rooting toward an ensemble approach.

IX. Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

References

[1] Y. Liu, J. Yang, Y. Huang, L. Xu, S. Li, and M. Qi, "Mapreduce based parallel neural networks in enabling large scale machine learning," *Intell. Neuroscience*, vol. 2015, Jan. 2016.

[2] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, vol. 19, p. 281, 2007.