CNN for Face/Comics recognition

DSE Unimi - Algorithms for Massive Data

Schiavoni Cosimo - 964563

August 6, 2022

Abstract

The Project consists in the realization of a Convolutional Neural Network (CNN) for the comics/faces recognition problem published on Kaggle. The code has entirely been created in the GoogleColaboratoy environment using Python 3.9, with the introduction of TensorFlow 2, published in the GitHub repository. The data set, consisting of 20.000 images, is publicly available and can automatically be downloaded during code execution. The experiment foresees the creation of a solution that is able to recognize the images and point out the correct label (real faces vs comic faces). To this extent, a single CNN structure has been conceived and it has been improved through the implementation of different regularization techniques: Lasso Regularization (L1), Ridge Regularization (L2) and Dropout. In addition, the solution has been conceived to face scale issues, employing distributed computing techniques in order to allow for Big Data analysis. Two main approaches have been introduced: TensorflowOnSpark (TFoS) library and Tensorflow API embedded features. The results showed two important insights: firstly, the implementation of regularization techniques can improve the efficiency of the model, reducing eventual overfitting issues; secondly, the introduction of scaling up techniques improves the time of training, reducing it by almost 10 times.

1 Declaration of authenticity

I declare that this material, which I now submit for assessment, is entirely my/our 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.

2 Introduction

The project consists in the realization of a Convolutional Neural Network algorithm, employed for the recognition and classification of real faces versus comics faces.
The algorithms have been developed in Python version 3.9, with the implementation of Keras (high-level API in the Tensorflow library).

3 Theory Backgroung

Convolutional neural networks (ConvNets or CNNs) are a type of neural network utilized in machine learning for classification and computer vision tasks. They leverage principles from linear algebra, specifically matrix multiplication, to identify patterns within an image.

They are comprised by three main types of layers, which are:
- Convolutional layer
- Pooling layer
- Dense Layer
- Fully-connected layer

The convolutional layer is typically the first layer of a convolutional network, additional convolutional and pooling layers can follow before feeding the data into a fully connected neural network. With each layer, the CNN increases its complexity, identifying every time greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object.

Convolutional Layer
The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires 3 main components, which is to say input data, a filter, and a feature map. For the extent of this project, the input is a color image, which is converted into a 3-D matrix of pixels. The first two dimensions are height and width, which represent image size. The third is depth, due to the presence of 3 RGB image color layers.
The filter (or kernel) is a 2-D array of weights that can vary in size (typically a

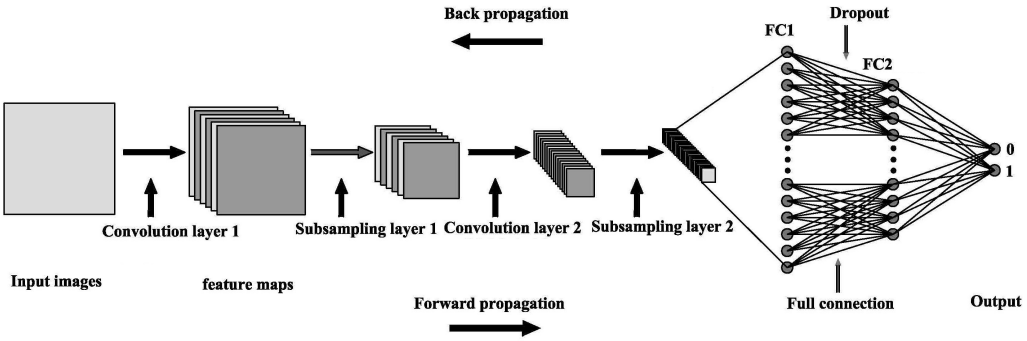

Figure 1: Structure of a Convolutional Neural Network.

3x3 matrix). It is applied iteratively (shifting across the entire image, each time by a stride) to specific areas of the image called receptive fields, which is to say portions of the image having the same size as the filter. At each movement, a dot product is calculated between the input pixels of the specific receptive field and the filter. The final output from the series of dot products is known as a feature map, activation map, or convolved feature. This entire process is known as a convolution.

Before training a network, it can be useful to initialize the values of the weights, a good practice is to set limits to these weights. One of the most popular techniques is the Glorot weight initialization algorithm, which basically uses limits that are based on the number of nodes in the network.

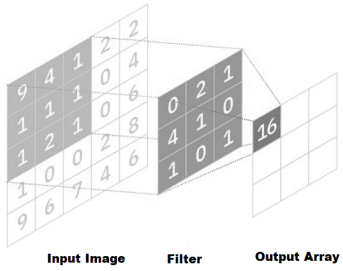

Figure 2: Numerical example of a convolution operation.

As can be noticed in the image above, each output value in the feature map does not directly connect to each pixel value in the input image. It only connects to the receptive field, where the filter is being applied. For that reason, convolutional (and pooling) layers are commonly referred to as "partially connected" layers. This characteristic can also be described as local connectivity. Note that the weights in the feature detector remain fixed as it moves across

the image, which is also known as parameter sharing. This behavior is different with respect to other parameters like the weight values in NN, which adjust during training through the process of backpropagation and gradient descent. During the convolution process, there are three hyperparameters that affect the volume size of the output that needs to be set before the training of the neural network begins:
1. The number of filters: defines the depth of the output (e.g. 3 filters produce 3 feature maps)
2. Stride: is the number of pixels that the kernel takes into consideration during iterative shifts across the image (typically the stride is set to 1 pixel, note that a larger stride yields a smaller output).
3. Padding: sets all elements that fall outside of the input matrix to zero, producing a larger or equally sized output. It is usually employed when the filters do not fit the input image or when it is useful to receive a feature map of the same size as the input image.

There are three types of padding:
- Valid padding (No-padding): drops the last convolution if dimensions do not align;
- Same padding: ensures that the output layer has the same size as the input layer;
- Full padding: adds zeros to the border of the input image in order to increase the size of the output.

As we mentioned earlier, another convolution layer can follow the initial convolution layer. When this happens, the structure of the CNN can become hierarchical as the later layers can see the pixels within the receptive fields of prior layers.

The convolution process is basically conducted through linear transformations, in order to introduce nonlinearity to the model CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map afterward.

Pooling Layer
Pooling layers (or Subsampling), conduct dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter iteratively across the entire input. However, the filter does not have any weights, the kernel just applies an aggregation function to the values within the receptive field, populating the output array. As parameters, it only needs the shape of the pooling filter.

There are two main types of aggregation that a pooling layer can carry out:
- Max pooling: As the filter moves across the input, it selects the pixel with the maximum value to send to the output array.
- Average pooling: As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

The main drawback of Pooling is that a lot of information is lost, on the contrary, it helps to reduce complexity, improve efficiency, and limit risk of overfitting.

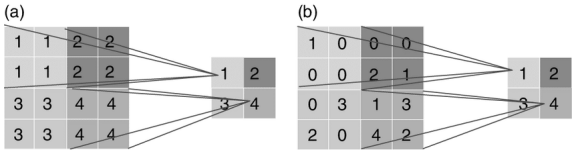

Figure 3: Numerical example of Average Pooling and Max Pooling operations.

Dense Layer
The Convolutional and pooling layers convert the image into numerical values in 2-D matrixes, which are re-organized in a 1-D array by a dense layer in order to allow the neural network to analyze data in the Fully-Connected Layer.

Fully-Connected Layer
Node layers contain an input layer, one or more hidden layers, and an output layer. In the fully-connected layer, each node in the output layer connects directly to a node in the previous layer and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network. This layer performs the task of classification based on the features extracted through the previous layers and their different filters. While convolutional and pooling layers tend to use ReLu functions, Fully-connected layers usually leverage a Softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.

$$\hat{y} = sgn(\theta^T x) \tag{1}$$

Where θ^T is the transpose of the weight matrix and x is the example. The Softmax function is more suitable in classification tasks, with respect to the Sigmoid function, whenever the outputs are mutually exclusive, which is to say that an image cannot be at the same time a comic and a real face.

For $x, \theta \in R_p$, feedforeward process in the fully connected layer can be written as:

$$E = \sigma(\theta_L \cdot \sigma(\theta_{L-1} \cdot \sigma(\dots \sigma(\theta_1 \cdot x + b_1) \dots) + b_{L-1}) + b_L) \tag{2}$$

Where b_i represents the bias vector, θ_i represents the weights and σ represents the Softmax activation function.This process in a neural network can be conceived as multiple instances of logistic and multivariate regression stacked on each other, so that the input of each sigmoid function is a linear combination of outputs of other sigmoid functions.

4 Empirical Analysis

1. Dataset: the dataset consists of 20.000 images and it's composed by two sets, the first one containing 10.000 real faces, the second one containing 10.000 comics faces. Each set of images is organized in a specific folder that is used to point out the label of the examples (in the specific case "faces" and "comics"), all the 1024 x 1024 color-scaled pixel images are composed by 3 color layers (RGB). Values are triplets of eight-bit numbers, contained in a range from 0 to 255, which point out the intensity of each color layer inside the pixel.

Figure 4: Example of the RGB composition of an image extracted from the original dataset.

2. Data Organization: The dataset is firstly collected into batches, the size of each batch is set as 32 images. Secondly, a split is carried out in order to organize the batches into three different groups, which is to say training, validation and test sets. The split allows for the creation of different input pipelines X that will be used for the training phase (train and validation sets) and for the test phase. The output is the binary class label, represented by a vector $Y = (Y_0, Y_1)$ of 2 dummy variables, let's say 1 for faces and 0 for comics. To this extent, the training set amounts to 70% of the total dataset, which is to say 14.000 images collected into 438 batches. The rest of data is collected into the validation set, which amounts to 30% of the total (6.000 images collected into 151 batches). The test set is nothing but a share of the validation set and amounts to 20% of it (1.184 images collected into 37 batches).

   A validation set is extremely important as far as it is used during training, to approximately validate the model on each epoch. This helps to update the model by giving "hints" as to whether it's performing well or not. Additionally, is not necessary to wait for an entire set of epochs to finish to get a more accurate glimpse at the model's actual performance. Using a validation set it is possible to interpret whether a model has begun to overfit significantly in real-time, and based on the disparity between

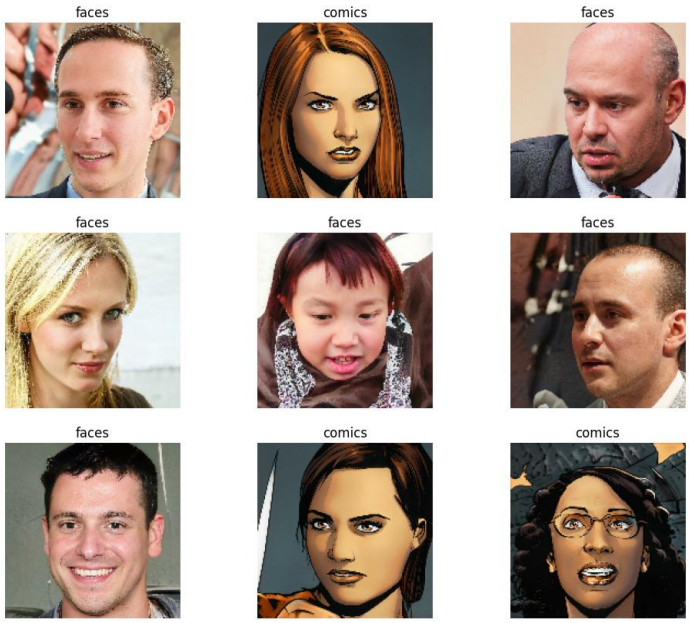

Figure 5: Example of a random permutation from Training Set.

the validation and training accuracies, is possible to trigger responses, such as automatically stopping training or updating the learning rate. The three input pipelines are prefetched in order to prepare subsequent elements simultaneously, hence speeding up batch iterations by exploiting the available computational memory in a more efficient way. The buffer size parameter, which defines the quantity of memory allocated for the computation, is set automatically.

3. Data Pre-processing: different data pre-processing techniques are employed in order to avoid overfitting, hence increasing the capacity of the model to intercept the right output even when images represent the same subjects in different conditions. At first, the image shape is reduced in size to 350 x 350 pixels, and shuffle is applied in order to retrieve a random permutation of images in each dataset. Secondly, Keras preprocessing has a class called ImageDataGenerator. It generates batches of tensor image data with real-time data augmentation. It is possible to add any form of data augmentation technique and specify the validation split size.

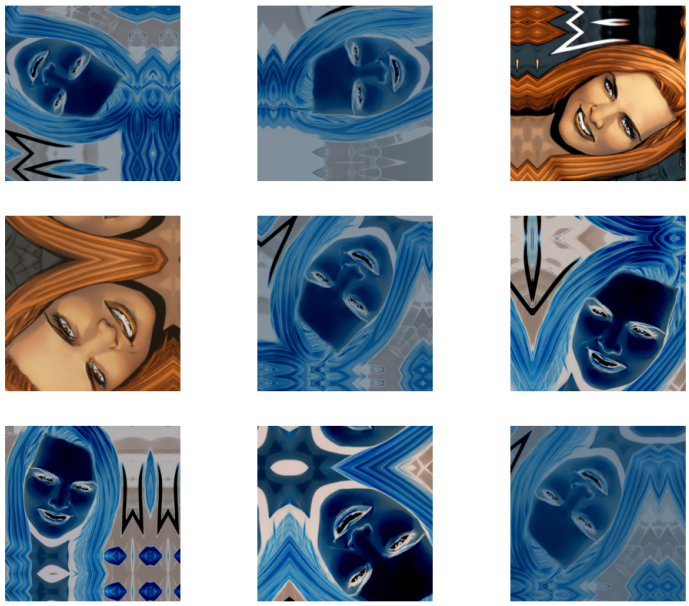

Figure 6: Example of data augmentation generated by all the functions introduced.

Several different functions are applied in order to transform the images:

(a) Random Flip: which randomly flips images in a horizontal or vertical sense;

(b) Random Rotation: which applies a random rotation by 30%;

(c) Random Contrast: which randomly applies a contrast by 10%;

(d) Random Invert: which randomly inverts the colors of the image by 50%;

(e) Random Zoom: which randomly zooms in by 30% or zooms out by 20%;

(f) Random Translation: which randomly translates the height and the width of the image;

(g) Image Normalization: which divides each RGB triplet value of a pixel by 255, changing the range to 0 - 1.

The reason why it should be used preprocessing is that in the real world pictures can be taken from any angle. For instance, by using flipping the model is possible to generalize on different scenarios. If it were to say, train on horizontal images of hands it might not predict hands in a vertical orientation.

5 Algorithms and implementation

For the extent of this project, the following structure of a CNN algorithm has been employed:

Layer (type)	Output Shape	Param num
Sequential	(None, 350, 350, 3)	0
Conv2D	(None, 173, 173, 32)	2432
MaxPooling2D	(None, 86, 86, 32)	0
Conv2D	(None, 42, 42, 32)	9248
MaxPooling2D	(None, 21, 21, 32)	0
Conv2D	(None, 10, 10, 32)	4128
MaxPooling2D	(None, 10, 10, 32)	0
Flatten	(None, 3200)	0
Dense	(None, 128)	409728
Dense	(None, 1)	129

Total params: 425,665
Trainable params: 425,665
Non-trainable params: 0

In order to check for different possible performances, 3 different regularization techniques have been employed and compared on the same CNN model. Regularization techniques are employed in order to improve the model's prediction effectiveness whenever results are affected by overfitting. Overfitting may normally happen when the model performs extremely well on the training set, while predictions on new data are pretty

poor.

The training phase is an optimization problem in terms of the difference between the true value, y, and the actual value, $\hat{y}$. This difference is often referred to as the loss, or the risk, denoted by L and is to be minimized with respect to the model parameters:

$$\min_{\{W_i\}_{i=1}^{L},\{b_i\}_{i=1}^{L}} L(y, \hat{y}) \tag{3}$$

In this model, the "binary cross-entropy" loss function has been employed, given that a binary classification problem is meant to be solved. The formula can be expressed as follows:

$$L_{log} = -\frac{1}{N}\sum_{i=1}^{n}(y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)) \tag{4}$$

Where $\hat{y}_i$ is the prediction of class 1, and is the prediction of class 0. When the observation belongs to class 1 the first part of the formula remains and the second part vanishes and vice versa in the case observation's actual class is 0.

Lasso regularization(L1):
Conventionally, θ represents the parameter that is multiplied by the input in order to get a prediction. λ is a hyper-parameter (regularization parameter) that is not learned during the training but is to be set before initiating the training. In L1 regularization λ is multiplied by the sum of and the modulus of all parameters before being added to the loss function. The most λ increases, the most the parameters will be small due to a higher penalization. The most evident drawback of L1 regularization is that parameters could result in sparse values (if they are too sparse predictions could suffer by underfitting). The values of the parameters will be close to zero.

$$L(y, \hat{y}) = L_{log} + \lambda\sum_{i=1}^{n}|\theta_1| \tag{5}$$

Ridge regularization (L2):
In L2 regularization the element to add at the end of the loss function is the sum of all the parameters squared. Same as L1 the increase of λ leads to a decrease in the parameters' values, due to penalization. The difference is that the weights do not result to be sparse and accuracy is better with respect to L1.

$$L(y, \hat{y}) = L_{log} + \lambda\sum_{i=1}^{n}\theta_1^2 \tag{6}$$

The penalization is actually performed during backpropagation, which is to say a technique of weights adjustment employed during the training phase when an error occurs. In fact, at every step the model makes predictions, it finds out the difference between the actual output and the prediction, then if the difference is not the minimum the model must make another prediction closer to the output. In backpropagation, it is necessary to express $\frac{\delta L}{\delta \theta}$ for all elements θ in all weight matrices and bias vectors, then the optimization problem can be solved by setting the derivative to zero for all parameters. One of the simplest algorithms for finding the zeros of the derivative is Gradient Descent, an iterative algorithm, where in each iteration every parameter is updated a small step, α, called the learning rate, in the direction of the negative gradient. In this specific project, the Adam optimizer (adaptive moment estimation) has been employed in order to optimize the computation. The Adam optimizer involves a combination of two gradient descent methodologies and it can be expressed as follows:

1) Momentum:
the algorithm computes the parameters after taking the gradient of the loss function at each step and accelerates the gradient descent algorithm by taking into consideration the 'exponentially weighted average' of the gradients:

$$\theta_{t+1} = \theta_t + \alpha m_t \tag{7}$$

Where

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \left[\frac{\delta L}{\delta \theta_t} \right] \tag{8}$$

Having, m_t the aggregate of gradients at time t (at t_0, $m_t = 0$), θ_t weights at time t, α the learning rate, β the moving average parameter, δL the derivative of the loss function and $\delta \theta_t$ the derivative of weights at time t.

2) Root Mean Square Propagation (RMSP):
Root mean square prop or RMSprop is an adaptive learning algorithm that considers the exponential moving average.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \left[\frac{\delta L}{\delta \theta_t} \right] \tag{9}$$

Where

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\delta L}{\delta \theta_t} \right]^2 \tag{10}$$

Having, v_t the sum of square of past gradients (i.e $\sum_{t=0}^{T}(\frac{\delta L}{\delta \theta_{t-1}})$. At t_0, $v_t = 0$) and ϵ a small positive constant to ensure that the denominator is always greater than 0. The combination of the two methods allows Adam optimizer for reaching the global minimum efficiently. Indeed a simple Gradient Descent algorithm has more probability to reach a local minimum rather than a global one during backpropagation.

Since m_t and v_t are initialized at t_0 as 0, that normally show a tendency to be biased towards 0. For that reason, it is useful to introduce bias-corrected algorithms:

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t} \tag{11}$$

and

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t} \tag{12}$$

In this way, the algorithms adapt to gradient descent after every iteration so that they can remain controlled and unbiased throughout the process. Finally, introducing the bias-corrected weight parameters $\hat{m_t}$ and $\hat{v_t}$ into the backpropagation process, the new optimizer is:

$$\theta_{t+1} = \theta_t - \hat{m_t}\left(\frac{\alpha}{\sqrt{\hat{v_t}} + \epsilon}\right) \tag{13}$$

Given that the loss function includes regularization terms, the adjustment of parameters carried out through backpropagation will consider this penalization as well, in fact both m_t and v_t are based on δL .

Dropout:
Dropout is another regularization technique that consists in dropping some random nodes in dense layers in the CNN. As a result, no node in the network is assigned with high parameter values, the parameter values are dispersed and the output of each layer doesn't depend on a single node.

As mentioned, the role of hyper-parameters is to reduce the complexity of the model. If a model is allowed to be very complex it can easily fit perfectly on the training data, but that does not mean that it will generalize well to new data, a phenomenon known as over-fitting. On the other hand, a simpler model may not capture all the details of the training data (which might be sample-specific), but only the more evident tendencies (which are more likely to be present in all data realizations) and thus generalize better. This can be thought of as a trade-off between variance and bias. A simple, highly regularized, model will have a higher bias and a lower variance than a complex, non-regularized model, and somewhere in between is the optimal trade-off. If regularization techniques

lead to poor accuracy, it could mean that too many parameters have been removed, hence causing underfitting (the opposite issue with respect to overfitting).

6 Scaling Up Techniques

Traditional machine learning models normally require big amounts of time for training when dealing with massive datasets. In order to face this issue, it can be useful to introduce some scaling-up techniques, which basically make the model able to ingest bigger volumes of data in a smaller amount of time. To this extent, two main strategies have been considered in this project.

TensorflowOnSpark (TFoS)
The first strategy may consider the integration of a Keras sequential model with a Spark architecture, which basically is a Big Data API able to manage massive datasets in an efficient way. There are several libraries that can carry out this integration task (e.g. TensorflowOnSpark). This approach allows for scaling up the existing TensorFlow application with minimal changes in code. Moreover, it supports all current TensorFlow functionalities and adds the possibility of synchronous/asynchronous training and Model/Data parallelism.
The functioning of this method is based on the launch of distributed TensorFlow clusters using Spark executors. To this extent, the first step of the realization of the strategy is the creation of a Spark Context and a Spark Session. Spark SparkContext is an entry point to Spark used to programmatically create Spark RDD, accumulators and broadcast variables on the cluster. In this step, the user should assign the app name and set the master node. The user can create only one Spark Context per time. Since Spark 2.0 most of the functionalities (methods) available in SparkContext are also available in SparkSession. The SparkSession is an entry point to Spark and creating a SparkSession instance would be the first statement needed to program with RDD, DataFrame, and Dataset. The user can create several Spark session objects, moreover many of them are required when is necessary to keep Spark tables (relational entities) logically separated. The Spark driver program creates and uses SparkContext to connect to the Spark Cluster manager to submit Spark jobs, and know what resource manager to communicate to. It is the heart of the Spark application. During cluster configuration, the user should normally be able to define the number of executors that are to be employed in distributed computing, this number should be lower or equal to the available executors in the nodes.

Spark pipelines normally work with a specific data structure called RDD "Resilient distributed datasets", which is to say collections of objects

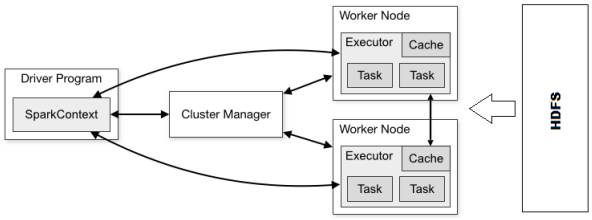

Figure 7: Spark logical infrastructure.

across a cluster with user-controlled partitioning and storage capabilities. It is a read-only collection of records, partitioned and distributed across the nodes in a cluster. It can be transformed into some other RDD through operations. Once an RDD is created, it cannot be changed, rather, a new RDD is generated. In Learning and Training applications the most frequent way to create RDDs is by taking an existing in-memory collection and passing it to parallelize the method of SparkContext. One important feature of this method is the number of partitions the collection is broken into. The user can specify the number of partitions as a parameter in the parallelize method, if the number is not specified, Spark automatically defines the parameter basing on the cluster.

On the other side, completed the configuration of the Big Data engine, it is necessary to setup the Machine Learning algorithm. In order to make the Keras CNN sequential model compatible with the Spark engine some transformations are needed (The detailed guide for converting TF applications to TFoS applications is provided here:
https://github.com/yahoo/TensorFlowOnSpark/wiki/Conversion-Guide).
During the conversion, one key parameter to define is the InputMode. In general, InputMode.TENSORFLOW is preferred (which avoids the conversion of Tensors to RDD pipelines), unless it is required to feed data directly from a Spark RDD or DataFrame (InputMode.SPARK). This feature is probably the element that fundamentally distinguishes the functioning of the final application.
In the Spark mode, everything is launched from the background process. TensorFlow generally has a python worker active on each executor in order to run a python task. The TensorFlow nodes are launched into the workers. In TensorFlow, the model can be represented by a computation graph, a representation of a computation by which nodes represent elements of that computation and edges represent the external data or computation results. The Parameter server nodes are special because as far as they are active, they block further tasks arriving from the node. The TensorFlow workers themselves are started on a background task along with a queue, which freezes up the python workers for that partitions call to feed data.

So once the TensorFlow nodes all together are started, they will communicate to each other via TensorFlow standard distributed ClusterSpec (which represents the set of processes that participate in a distributed TensorFlow computation. To create a cluster with n jobs and n tasks, it is necessary to specify the mapping from job names to lists of network addresses, statically defined a priori). Then each RDD partition is inserted into a queue, which is read by the TensorFlow feed dictionary call whenever TensorFlow is ready for new data.
On the contrary, in the TensorFlow mode, everything is launched from the foreground process. Tensorflow has full control, workers are connected to each other from the cluster and workers will read directly from HDFS (Hadoop Distributed File System, which divides files into blocks and stores each block on a DataNode. Multiple DataNodes are linked to the master node in the cluster, the NameNode. The master node distributes replicas of these data blocks across the cluster). It's like starting up TensorFlow on nodes that happen to be in the cluster. Another big difference is related to failure recovery.
In the Spark mode, TensorFlow nodes are run in the background, so Spark has no visibility of any error happening there. Instead, it has visibility of the data feeding visibility job, which is pushing data to the TensorFlow nodes, so if it happens a temporary glitch on a partition, spark retry it and feed data again to the node.
On the contrary, in the TensorFlow mode TensorFlow has to fail in order for Spark to retry the task.

The biggest problem of TFoS applications happens, for both the input modes, whenever the system loses not just a task but the entire executor. This issue is due to the fact that the Cluster Manager does not necessarily respawn the executor on the same host and node. As mentioned above, Tensorflow ClusterSpec is actually statically defined and whenever a TensorFlow node is started, it needs to know where exactly each of the other nodes is.

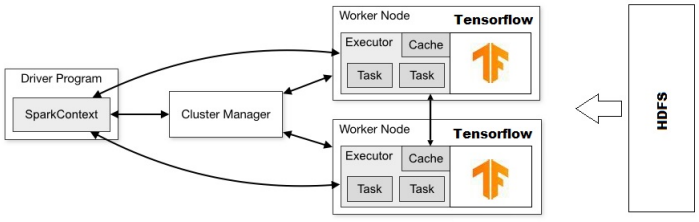

Figure 8: Spark logical infrastructure with the integration of the Tensorflow engine.

This approach can be useful in order to exploit a more complete set of machine learning techniques provided by Tensorflow, maintaining all the features of a Big Data API such as Spark. It can also serve in the case in which a Big Data pipeline is already functioning and there is a need to build a machine learning model upon it. However, the main drawback is the management and the combination of two different engines which are integrated into the same system.

The benefit that can be obtained from this technique can be defined by a linear function, in fact the estimated computing time normally decrease by almost 50% per each device added in the cluster.

Tensorflow API

The second strategy is simpler and requires only the introduction of distributed training techniques in Tensorflow API. This technique is pretty recent and has been conceived in order to employ distributed training as an embedded solution in a machine learning library. In fact, more recently the need to scale up machine learning models and the need to expand big data API functionality toward machine learning are converging (Spark is now equipped with a ML module as well).
For sake of simplicity, the second approach has been fully developed in this project.

The approach foresees a central-storage strategy (the MirroredStrategy creates copies of all variables in the model's layers on each worker in a device), which is to say bunches of data are separated and sent to different workers in all the available devices, while the model and variables are replicated across all the devices into replicas.

The job is run synchronously across all devices, every replica in each device keeps its own copy of every variable (sync is garanteed due to an update of variables). It is based on the "All-reduce"algorithm: "All" information is transferred from any device to any device, "reduce" can be any reduction operation, such as sum, multiplication, max or min.

Synchronous training can be represented as follows. At first, each device takes a subset of the training data and does a foreward pass by using the local copies of the variables. Secondly, it does a backward pass computing gradients using the same local copied variables. Then gradients are aggregated by using all-reduce algorithm, which communicates a single aggregated variable value to all the replicas. Each replica applies that gradient to its local variables afterward. Since the all-reduce alorithm produces the same value for all the replicas, the updates are all the same across all the devices, hence values are synchronized. In terms of performance, it is a good approach because each step (at each layer) can start immediately due to the fact that variables are copied locally and always available, hence allowing for parallelism. Even if no previous information

is available on the performances, it can be showed from this experiment that significant improvements can be obtained, passing from a 10 minutes per epoch training time to a 1 minute per epoch, resulting in almost a 10x increase. It would be interesting to compare the results obtained by this approach with respect to the other one, defining the same number of devices and workers in the cluster.

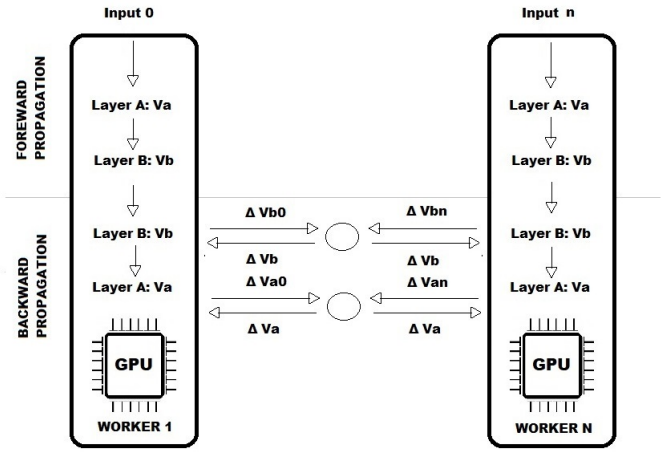

Figure 9: Logical schema showing the parallelization of computation between several workers.

Moreover, the same approach can scale to a multi-worker environment. The MultiWorkerMirroredStrategy creates copies of all variables in the model's layers on each device across all workers.

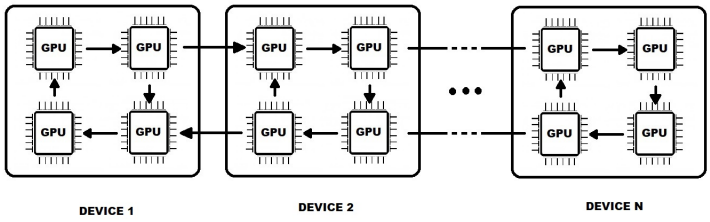

Figure 10: The parallelization can be elevated to multiple devices.

7 Eperiment Description

The experiment started by coding a single node CNN model, which was supposed to provide a first starting point for the final result. In the beginning, a one convolutional-pooling layer model structure has been developed, more convolutional-pooling layers have been introduced in order to improve the complexity and receive better results afterward. Secondly, once the model seemed to work efficiently in image forecasting, the model has been scaled up by employing a Mirrored-strategy, which is an embedded functionality in TensorFlow 2 that automatically distributes the computation to all the available workers in the cluster. Thirdly, the forecast ability of the model has been evaluated by providing new images downloaded from the web and executing the test. Finally, new versions of the basic model have been considered in order to improve forecast results. To this extent, regularization techniques have been introduced and analyzed.

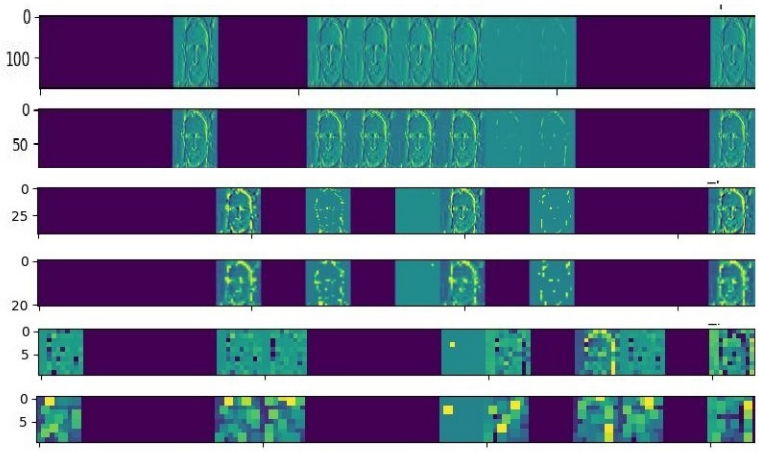

Figure 11: The image shows the feature maps resulted by each step of convolution and pooling processes.

8 Comments and Discussion

The image below shows the experimental results for both training and validation sets for all the models. Comparing the curves, it's possible to appreciate the differences for every methodology. The base CNN model is the one that increases at a faster rate toward higher/lower levels of accuracy/error, while the regularized models normally increase at a slower rate. This means that in fewer iterations the base CNN model is able to

collect more information with respect to the other ones. Moreover, the results provided by the base CNN model show a smoother line, which means that the model has a similar performance running on validation and training sets. On the contrary, regularized models reveal lower smoothness in validation lines, due to a lower level of robustness of the models. This is particularly true for the Lasso Regularized CNN model, which shows a greater distance between accuracy and training results with respect to the other models.

Running the test on a small sample of 9 random images extracted from the test set, the Base CNN model performs pretty well. In fact, all the randomly sorted images are correctly classified.

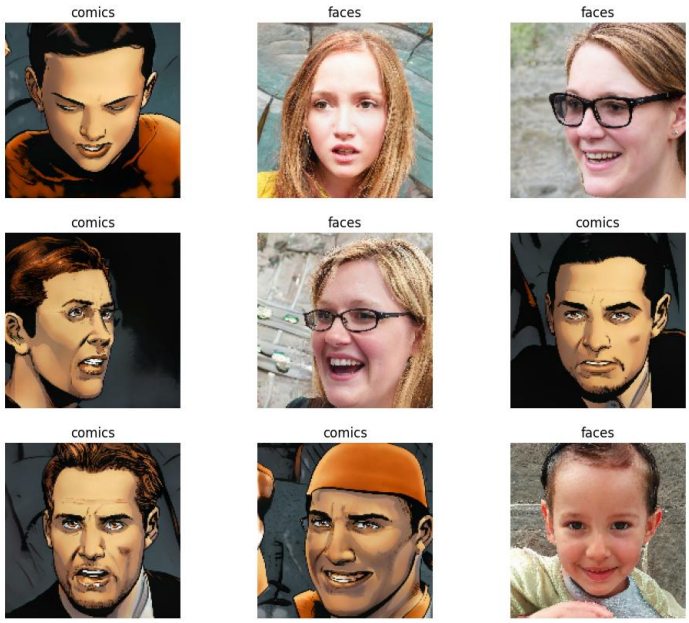

Figure 12: Results of the Normal CNN Model over a subset of the testing set images.

However, these results seem too good to be effective in real-world case forecasting. To this extent, a final test has been carried out with new images downloaded from the web. As it can be noticed, results provided by the Base CNN Model in a real-world case are poor enough to consider the model affected by overfitting. The risk of overfitting is given by the

fact that predictors generated in the training phase are too good at forecasting the labels. This issue allows for very good predictions on the test set, in the case in which data do not differ too much from the training set. As regards regularization methods, they are generally not useful to solve this issue whenever it is present, but they can help to mitigate it. Indeed, regularization models reach high accuracy levels in the validation and training phases (which can be a signal of overfitting), however, in a real-world case, the prediction seems to be more precise.
As regards the new images, 3 kinds of pictures have been introduced: real pictures, comics pictures inspired by real pictures, and real pictures showing comics pop-art/makeup (for which the algorithm is not trained). The aim of this test is to check how good the algorithm is at distinguishing comics pictures, similar to real pictures. Moreover, the second aim is to observe the precision of the algorithms at making classification decisions when facing ambiguous data (real pictures with makeup).

As regards Base CNN Model, 3 errors over 7 have been registered (excluding the 2 ambiguous images), while the Dropout CNN Model returned remarkable results, only 1 error. The algorithm failed at classifying images 1a and 2a, which are correctly classified as comics by the Dropout CNN Model. Both models failed at classifying image number 5, which should ideally represent a comics image. Finally, regarding ambiguous images (number 3 and number 8), Base CNN Model performed better with respect to Dropout CNN Model. In fact, even if both of them should be classified as real faces, image 3 is clearly identifiable as a real face with makeup, while image 8 is much more close to the face of a comic character. Obviously, this decision depends on the sensibility of the data scientist, which can arbitrarily define which is the separation grade of correct/wrong classifications, basing on his own impressions. An improvement of the model could be considered in this sense, by introducing ambiguous labeled pictures in the training and validation sets.

In conclusion, given that the model is not supposed to be able to classify ambiguous data, hence it is not possible to objectively measure the goodness of these predictions, regularized methods seem to be better at predicting faces/comics faces in real-world case.

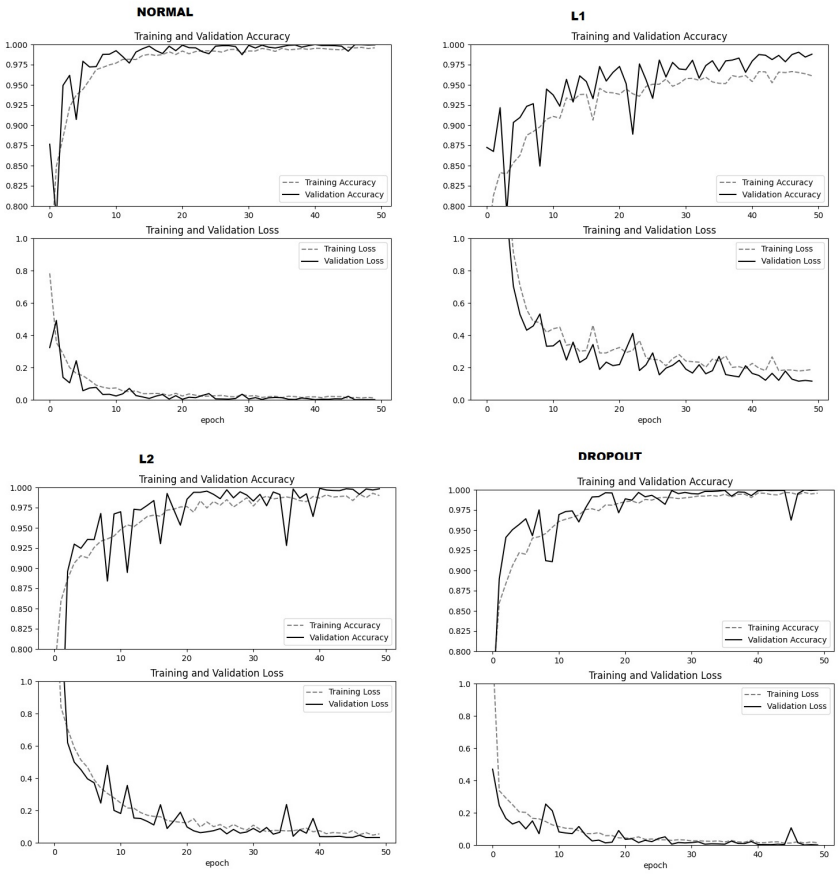

Figure 13: Graphs of 50 epochs. Accuracy and loss rate detail for each model.