Big Data course project

Autoencoder-based Recommendation System: Integrating Spark and Deep Learning models

University of Verona

Alessandro Rodeghero VR474699

1 Introduction

For the Big Data project I wanted to explore the potential of Apache Spark beyond loading, extracting and transforming data. I wanted to integrate data processing with the creation of a recommendation system based on a deep learning model, i.e. an Autoencoder (AE).

Since in today's industry large deep learning (DL) models are trained using a cluster to perform distributed training, I wanted to try this approach in my project. Taking advantage of the potential of distributed computing offered by Apache Spark, I wanted to train the AE in a distributed way, comparing the performance of the model obtained compared to what is obtained with the classic local training of deep learning models. To perform distributed training I used the Elephas library, which integrates Keras and Spark functionalities to allow parallelization of model training.

Machine/Deep Learning is a field that interests me and I took the opportunity of the Big Data course to integrate, in the creation of a DL model, the entire part preceding the training, i.e. what concerns the preparation and processing of the data, seizing the chance to acquire proficiency in utilizing Apache Spark.

This project was designed and built from scratch. Different analyzes will be presented regarding the processing of data of different sizes, also comparing parallel and non-parallel computation. Furthermore, the performance of the recommendation models obtained will then be evaluated.

2 Materials and Methods

2.1 Dataset

For the development of this project I used the MovieLens dataset which is a widely used and well-known dataset in the field of recommender systems. It contains movie ratings and user information, making it valuable for building and evaluating recommendation algorithms. The MovieLens dataset is compiled from movie ratings data sourced from the movielens.org website, maintained by the GroupLens Research Lab. This website offers a non-commercial service that delivers personalized movie recommendations, leveraging user-uploaded ratings

as its foundation.
For this project, i utilized the main two parts of the dataset:

- movies: informations about the dataset's movies (movieId, title, genre). In the title string, it is also provided the year of publication.

- ratings: all collected users' ratings (userId, movieId, rating, timestamp). Each row refers to the rating that a user left about a movie. Ratings values starts from 0.5 to a maximum value of 5.

Both movies and ratings files are provided in *.csv* format. The dataset contains data about 62423 movies and 25 millions ratings left by 162541 users.

2.2 Spark

Apache Spark is an open-source distributed computing framework designed for processing large volumes of data in a fast and fault-tolerant manner. It was created in 2012 as a response to the limitations of the MapReduce model, with a focus on addressing the need for faster and more efficient data processing. Spark offers a high-level API that allows developers to write distributed data processing applications using a functional programming style. It provides implicit data parallelism, which means that Spark automatically distributes data and computations across a cluster of machines, making it easy to leverage the full potential of a cluster for processing large datasets.

Resilient Distributed Datasets (RDD) are the fundamental data structure in Spark. They are immutable, distributed collections of data that can be processed in parallel across a cluster. RDDs provide fault tolerance by allowing the framework to trace data transformations, enabling re-computation if needed.

This means that even if a node collapses, Spark can recover the lost data and continue processing without interruption. When a node collapses, Spark can reassign tasks to other available nodes and recompute the lost data.

Spark offers two categories of operations on RDDs:

- Transformations: operations that create a new RDD from an existing one. They are lazily evaluated, meaning they are not executed immediately. Examples of transformations include *map* and *filter*.

- Actions: operations that trigger the computation the RDD and return the desidered results to the driver program or write data to an external storage system. Actions initiate the entire Spark computation plan, but do not return an RDD. Examples of actions include *count*, *collect*, and *saveAsTextFile*.

The MapReduce model conducts data operations by relying on disk storage. In contrast, Spark performs in-memory data processing, reducing I/O overhead and improving speed, offering more performance compared to MapReduce. It allows iterative algorithms to be run efficiently, which is essential for machine learning tasks.

For this project I utilized PySpark, which is the Python version of Apache Spark.

2.3 Autoencoder Based Recommendation System

An Autoencoder based Recommendation System is a recommender system that utilizes deep learning techniques to provide personalized recommendations to users.

The foundation of the recommendation system developed for this project lies in the ”ratings array”. This array is essentially a representation of each user's interactions with movies. Each index in the array corresponds to a specific movie, and the value at that index reflects the user's rating for that movie. If a user hasn't rated a movie, the corresponding entry in their ratings array is set to 0.

To build this system I employed an autoencoder, whose main goal is to learn a meaningful and compact representation of rating arrays in a lower-dimensional latent space and then reconstruct them from this latent representation. The model is trained to minimize the difference between the input ratings array and the reconstructed one.

When a recommendation has to be made, the user's array is passed through the autoencoder. From the reconstructed array, one of the films with the highest rating is chosen among those that have a rating of zero in the original rating array, i.e. that the user has never seen.

The idea behind this way of recommendation is that users with similar tastes are likely to have similar rating arrays. Consequently, also their embeddings are similar and so located near in the latent space. The reconstruction of an array is influenced by nearby embeddings in the latent space and in this way the reconstructed array carries with it the influences of its neighbors, that could be movies ratings that the starting user has never provided, therefore films he have never seen but might appreciate.

The recommendation system leverages on these perturbations in the reconstruction of the rating array, checking for new films to propose.

2.4 Elephas

Elephas is a Python library that extends Keras, enabling the execution of distributed deep learning models on a large scale using PySpark. Elephas offers support for various applications, including data-parallel training of deep learning models and distributed inference and evaluation of these models.

By integrating Keras with Spark, Elephas keeps the user-friendly nature of Keras, facilitating rapid development of distributed models that can process large datasets. Elephas achieves this by implementing a set of data-parallel algorithms built on top of Keras utilizing Spark's RDDs.

The workflow involves initializing Keras models on the Spark driver, serializing and sending them to worker nodes along with data and broadcasted model parameters. The Spark workers then deserialize the models, train on their respective data chunks, and send back gradients to the driver. An optimizer on the driver side updates the *master model* either synchronously or asynchronously.

The fundamental building block in Elephas is the *SparkModel*, created by providing a pre-compiled Keras model, specifying update frequency of the master model and parallelization mode. Once initialized, the model can be easily trained on an RDD. The Elephas `fit` function offers the same options as a typical Keras model, including parameters like epochs and batch size.

The SparkModel can also be employed for distributed inference and model evaluation. The `predict` and `evaluate` methods follow the same data parallelism concept seen in training. This means that the model is serialized and dispatched to the worker nodes, where it is used to evaluate segments of the testing data.

In the context of my project, Elephas was used to train the autoencoder used by the recommender system in a distributed way. In my case I have specified a synchronous mode of updating the master model with a single epoch as frequency. In this way, each worker trains its local model for a single epoch and then sends the gradients to the master model which calculates the average gradients starting from the those provided by each worker. The master model is then updated based on the calculated average gradients. This is done synchronously, i.e. when all workers have finished their training for an epoch. After this phase, the parameters of the main model are transmitted to the worker nodes to update their local model and the all process repeats.

To speed up the training process, an asynchronous update mode could be used, which does not take worker alignment into account. In this way, however, the resulting model would perform less well due to less accurate training.

3 Project Phases

To create the recommendation system of this project, the following steps were followed:

1. **Data loading**: reading movie and rating datasets using the built-in Spark features.

2. **Ratings data transformation**: process the ratings data so that each record is represented in the tuple format (movieId, (userId, rating)).

3. **Movies data transformation**: process the movies data by removing genre information and extracting release year information from the movie title. Then, filter the movies based on their release year and present the data as (movieId, (userId, rating)).

4. **Data integration**: integrate the information extracted from both the ratings and movies data obtained in previous phases. The join operation leverages the *movieId* key to create a unique RDD, combining the previous

two. Also, this step filters the ratings and only keeps those related to the selected movies.

5. **Statistical analysis**: perform various operations to collect statistics related to the subset of data obtained. This includes gathering information about the number of films selected, the total number of ratings, and the count of unique users.

6. **Ratings arrays generation**: create ratings array for each user by grouping the ratings based on *userId* key. This allows for the aggregation of all ratings provided by a user, enabling the construction of their ratings array.

7. **Model training**: after splitting the obtained dataset into train and test sets, Elephas is used to perform the distributed training of the autoencoder with the users' ratings arrays.

8. **Model evaluation**: reconstruct the ratings arrays based on the train and test sets. Evaluate the performance of the model in both cases by calculating Mean Squared Error (MSE).

9. **Recommendation testing**: demonstrate the recommendation system's functionality through an example recommendation test.

4 Experiments' Setup

For this big data project I wanted to test Spark's performance using three different subset of the MovieLens dataset, selecting the films by release year range and consequently reducing the number of ratings and users participating:

year rage	**# movies**	**# users**	**# ratings**
1950 - 60	2821	77048	565530
1950 - 70	6271	102319	1288730
1950 - 80	10691	136921	2737837

Table 1: MovieLens subsets' statistical analysis.

I wanted to create subsets by filtering MovieLens by years in order to emulate the progressive release of films from year to year. I tried to grow the subsets by observing the number of ratings in such a way that they approximately doubled with each cut of the dataset.

For this project I also wanted to test the effectiveness of the distributed computation offered by Spark, both in terms of the speed of parallel computation in manipulating large quantities of data, and in terms of the quality of a deep learning model trained in a distributed manner on the cluster compared to the classic local approach. Clearly not having a real cluster, the experiments were carried out on a single machine taking advantage of its multi-core capabilities to emulate a cluster.

Regarding the setup for parallel computation, I initialized the Spark Context in such a way that it could use all the available cores, maximizing the distribution of the computation. To do this I specified `local[*]` as `setMaster` parameter for the *SparkConf* object to run Spark locally with as many worker threads as available CPU cores. To ensure that all cores were used, I also specified `spark.executors.instances` to 8, the number of logical cores available on the machine on which the experiments were performed.

To mimic traditional computing behavior, I configured `setMaster` to `local[1]` and `spark.executors.instances` to 1. As a result, the cluster operates with just one worker, utilizing only a single CPU core for computations. Additionally, I explicitly specified the RDDs to be partitioned into a single partition using `rdd.repartition(1)`. This approach closely simulates local computing, where all data resides within a single partition. Furthermore, by dividing the RDDs into a single partition it is possible to train the autoencoder in a classic way, i.e. with the entire dataset available without having to aggregate multiple models trained on different data chunks.

The configuration of the experiments refers to each combination of dataset cut and computation mode. I paid attention to the use of RAM(GB), CPU(%) and time(s) spent for all phases described in the *Project Phases* section for each configuration of the experiment, except for Model Evaluation and Recommendation Testing phases, which concern model performance evaluations and not to the use of Spark. I employed the `resources_monitor.py` script to collect information about resource usage.

Since Spark initiates computations only when an *action* has to be performed, at the conclusion of each phase, I triggered the computation by using `rdd.count()`. This decision was made because, in scenarios where a project phase solely consists of *transformations* without any actions, Spark would not automatically initiate computations. Consequently, it would be impossible to monitor and record resources usage and time spent. The choice of employing *count()* as the triggering action is based on its lightweight nature. It involves minimal computation and impacts minimally on system resources and processing time. It operates by simply counting the elements within the RDD, but it still necessitates the execution of all prior transformations and actions. This approach allowed to effectively capture resource utilization in all project's phases.

To handle the different cuts of the dataset, adjustments were required in memory allocation settings to prevent memory-related issues with Spark. The modified parameters include:

- `spark.driver.memory`: specifies the amount of memory to be allocated to the driver program, which manages the SparkContext and coordinates tasks on worker nodes. Sets the heap memory size for the driver process.
- `spark.executor.memory`: specifies the amount of memory to be allocated to each executor in the cluster.
- `spark.driver.maxResultSize`: determines the maximum size of results that Spark's driver program can collect from the executor nodes. This configuration helps prevent the driver from running out of memory when collecting large result sets.

Initially, when analyzing the 50-60 subset, the specified values were 9GB, 2GB, and 2GB, respectively. However, these parameters needed changes when working with larger subsets. For instance, with the 50-70 subset, it became necessary to increase *maxResultSize* to 3GB and *driver.memory* to 13GB. Similarly, for the 50-80 subset, *driver.memory* was increased to 15GB, and *maxResultSize* set to 5GB.

The autoencoder used for the experiments is a fully connected model and maintains the same architecture for all cuts of the dataset, changing only the size of the input and output layers, which adapt to the number of movies present in the subset used. The latent space dimension is set to 25, so the ratings arrays informations are compressed to this lower dimensional space.

```
-----------------------------------------------------------------
Layer (type)                Output Shape              Param #
=================================================================
dense (Dense)               (None, 150)               423300
activation (ReLU)           (None, 150)               0
dense_1 (Dense)             (None, 25)                3775
activation_1 (ReLU)         (None, 25)                0

dense_2 (Dense)             (None, 150)               3900
activation_2 (ReLU)         (None, 150)               0
dense_3 (Dense)             (None, num_movies)        425971
activation_3 (Softmax)      (None, num_movies)        0
=================================================================
```

The parameters for the training phase also do not vary with the experiment:

```
- epochs = 50
- batch_size = 64
- loss = mse (Mean Square Error)
- optimizer = Adam()
- learning_rate = 0.001
```

One last thing to mention is that the total memory available on the machine used is 24GB. The CPU is an Intel i7-1165G7 with 4 physical cores and 8 threads.

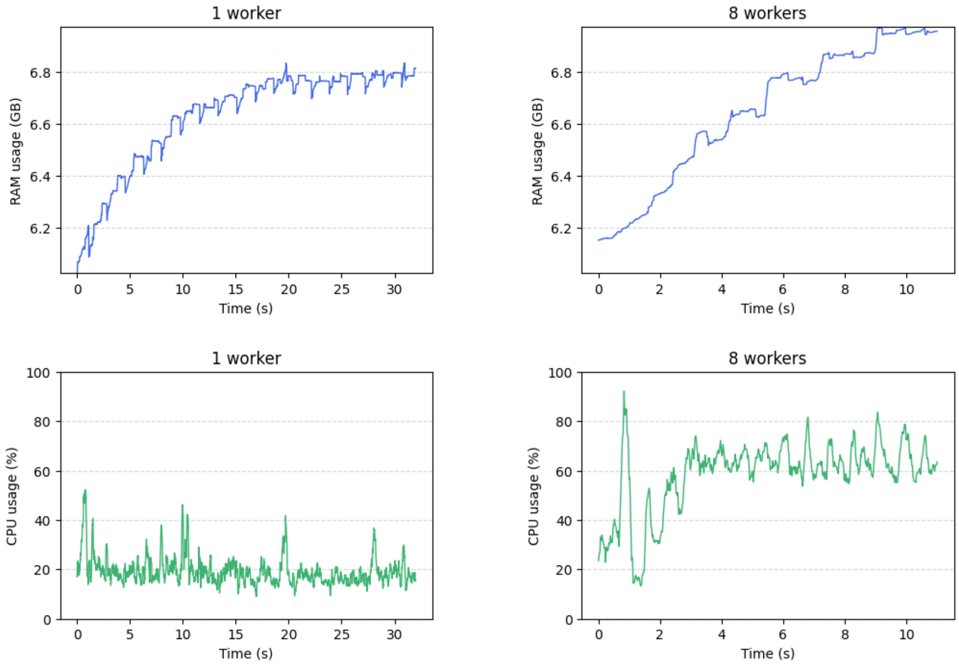


Figure 1: Data Loading resources usage.

5 Results

In this section, the results collected for each phase of the project are presented, considering the different configurations of the experiments. For each phase I will give an introduction to the operations that are performed to process the data. From now on, in the charts, the green line will refer to the CPU usage (%) while the RAM usage (GB) is shown with the blue line. The evolution of resources usages are shown in respect to time(s). For each chart, the configuration is indicated, i.e. which subset it refers to and the type of computation. ”`1W`” indicates the computation with single core and single partition, while ”`8W`” indicates complete use of the cluster.

Since a comparison will be made later between the model trained in a distributed and non-distributed way, I also wanted to present the data processing performance using the distributed(8W) and non-distributed(1W) computing.

5.1 Data loading

For the *data loading* phase (fig.1), I reported the trend of resource usage regarding the loading of *ratings.csv*. The different cuts of the dataset are not reported as the data is not yet been filtered, but it is interesting to compare the performance between parallel and non-parallel computation.

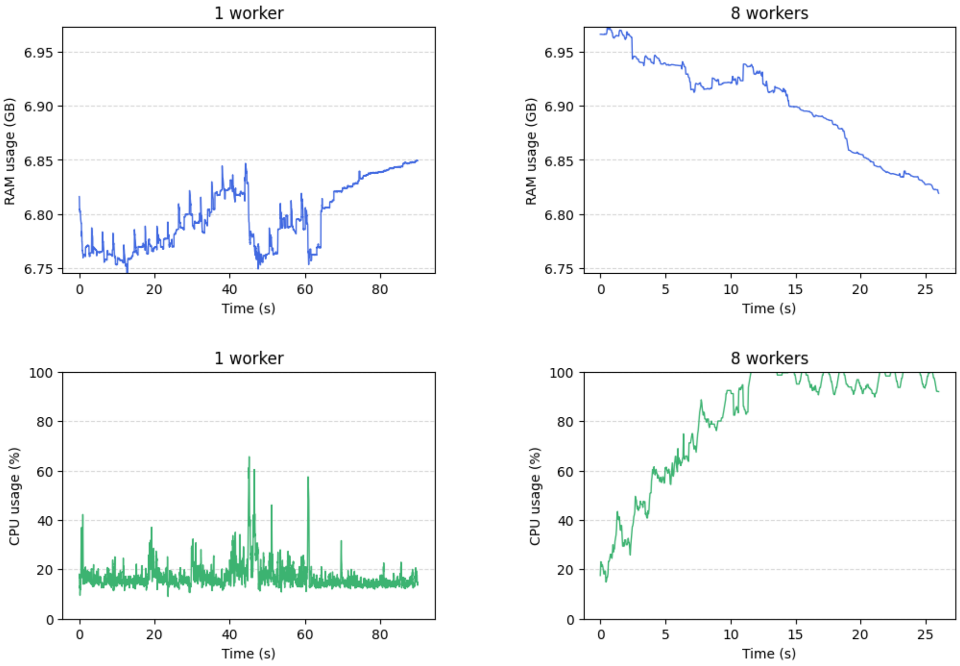


Figure 2: Ratings data transformation resources usage.

In PySpark, the `textFile()` method is used to read data from text files and create an RDD from the contents of those files, but it doesn't immediately load the data into memory. This method is a lazy transformation, i.e. the reading of the data and the creation of the RDD aren't performed until an action has to be executed. This is the first example of why I needed to use `rdd.count()` at the end of each phase to monitor resources usage.

Looking at the charts, we can see how parallel computation takes less time than using a single worker. Furthermore, the CPU is used more in 8W configuration since all CPU cores are available, unlike 1W where the use is restricted to only one core. RAM grows about the same in both configurations. This makes sense because it is being loaded into memory the same data in both cases.

The time used to load the data probably seems too long but it must be considered that the various partitions have to be allocated and the result of `rdd.count()` must also be computed.

5.2 Ratings data transformation

In this phase(fig.2) the ratings are processed and prepared. The only filtering operation is the removal of the header line since `textFile()` does not allow for skipping the first line when reading text files. Then the values are separated by observing the commas delimiters and the ratings are reshaped in the form of a tuple (movieId, (userId, rating)).

Also in this case there are no filters to create the subsets so the comparison remains only between 1W and 8W. We can observe a much more massive CPU usage in 8W compared to the data loading phase. This is probably due to filtering on the lines searching for the header one and reformatting the rating tuple, operations that involve computation. The RAM is slightly larger in 8W, on average by a hundred MBs. Since Spark can utilize multiple threads for the computation, can consume more RAM because each thread may have its own data in memory and potentially processing more data concurrently can lead to higher RAM usage. However, in 1W, as there is only one worker and one partition, the data is processed sequentially, therefore consuming less memory at the same time. The RAM has a rather constant behavior in both cases, with oscillations around 100 MBs. It makes sense that the RAM remains almost constant, as new data is not generated but existing data is reformatted.

5.3 Movies data transformation

To process movies data(fig.3), several operations are necessary. First of all, here too, we need to remove the header line. At this point there are several steps to take since it is possible that commas appear in the title, a factor that could alter the correct division of the line into elements. Additionally, only titles containing commas are enclosed in double quotes. To resolve these issues of poor format consistency, several mapping steps are performed. Once the year of publication has been extracted, each tuple is reformatted as (movie_id, (title, year)), and at this point it was possible to filter with respect to the year to create the movies subset.

This is the first case in which the three subsets are present. However, processing movies data is a very simple task computationally. With so few lines to process we can't even perceive the differences between the various cases. To make a comparison with the previous phase, the ratings are 25 million lines while the films are only 62 thousand. In fact, in the previous phase, with much more data available, the faster processing capacity of parallelization is much more evident. Looking at the this phase's charts, there are no particular differences in any of the 6 configurations: they all show a constant trend in RAM and CPU usages. The duration of the tasks is also completely comparable between the 6 configurations. Furthermore, all the operations performed are lazy transformations so it was necessary to call `rdd.count()` to start the effective computation.

5.4 Data integration

At this point it is necessary to integrate the selected movies and the available ratings in order to eliminate those ratings that are not relevant to the films in the subset. To do this I used `ratings.join(movies)` leveraging the `movieId` key. By doing so, only the ratings that have a key in common with a movie are kept, thus resulting in filtering. `join()` then returns an RDD with filtered ratings

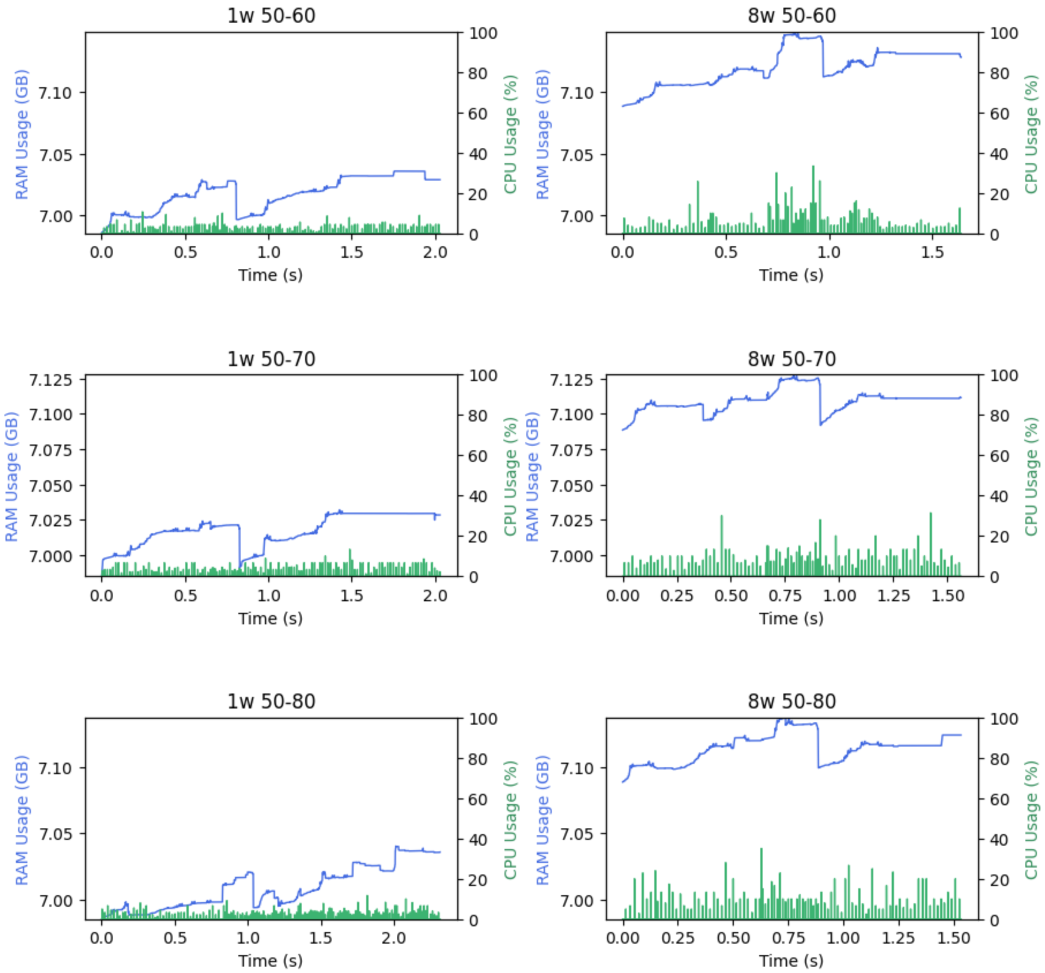


Figure 3: Movies data transformation resources usage.

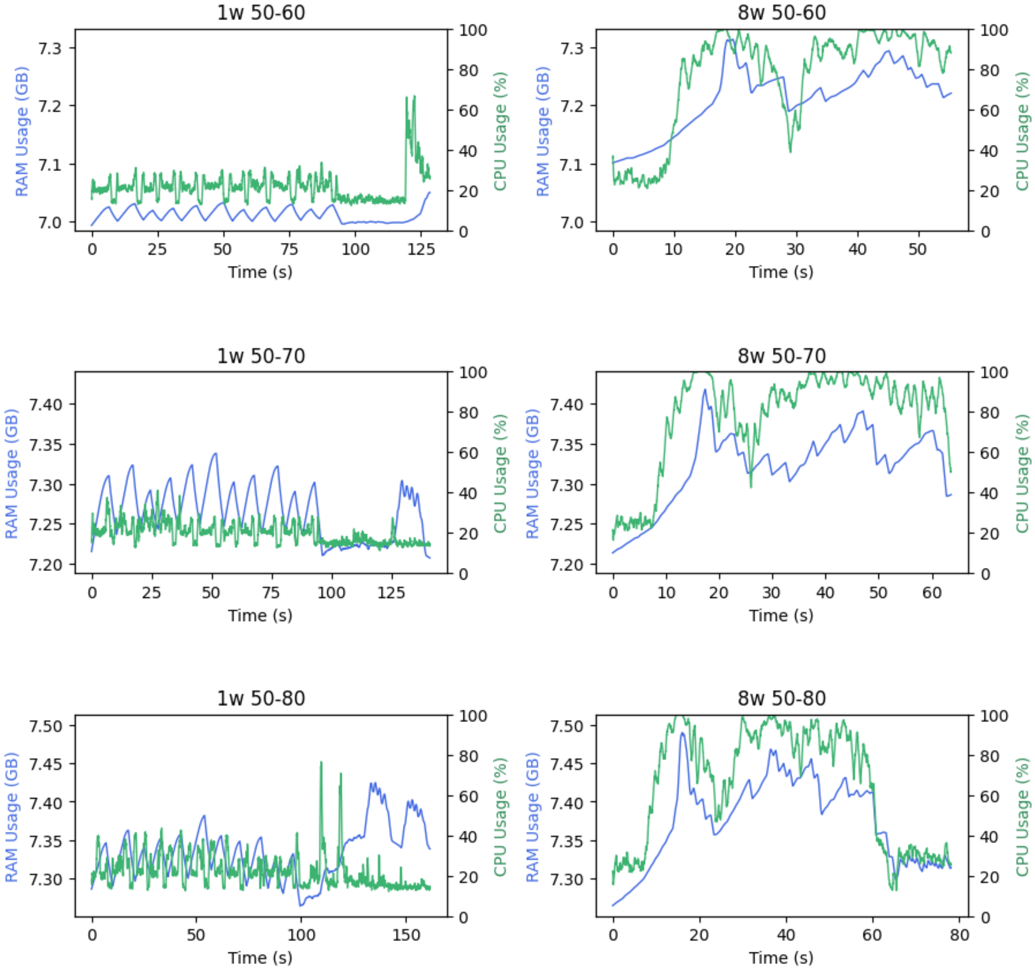


Figure 4: Data integration resources usage.

whith additional (title, year) informations, that are not useful. The second operation carried out is therefore to reformat the data into (userId, movieId, rating).

Given that the information contained in the final RDD was already entirely contained in the ratings RDD, actually the most computationally lightweight process would have been to extract the movieIDs by collecting them in a list and filtering the ratings by keeping only those with identifiers of the films present in the list . I chose to use join() to experiment with it and to emphasize the differences between the various configurations in resource usage. Also, with larger datasets it may be more correct to use the join.

Charts(fig.4) shows how in 1W the RAM remains almost constant, always considering some oscillations in the order of a hundred MBs. CPU usage remains throttled in the single computation case, around 25%, as in the previous cases. In 8W, we can see how RAM and CPU work at full capacity to ensure fast execution, with the CPU at some moments touching 100% usage. Unlike in 1W, in parallel computation we see the RAM grow much more and then settle back to more suitable values at the end of the process. There is a greater use of RAM in the central part of the graph, most likely due to the greater amount of data present in memory favored by parallelization and probably also a few additional MBs to manage cluster execution. It is correct to note how RAM usage increases with the size of the subset, even though the size of the analyzed data is still relatively small. In the ratings arrays generation phase, this behavior is much more evident.

5.5 Statistical analysis

The statistical analysis of the subset is an operation that is actually not completely necessary, as it returns the count of films, users and ratings. Its only two utilities are the count of films present and the collection of movieIDs used to specify which film the index of the ratings array will refer to. The movies count value will then be used to specify the size of the autoencoder's input layer. I wanted to collect the other statistics to make the machine work and extend this process only for the purposes of presenting the results.

The graphs(fig.5) show how this is a memory-task. The CPU is not stressed in any case, not even in parallel computation, while the memory makes small changes in the early stages of the execution. The changes in memory usage are probably due to the instantiation of two temporary RDDs to collect information about the movies. The `unpersist()` method will then be applied to these RDDs to indicate to Spark that their allocation can be freed. Subsequently the memory usage returns to values similar to the initial ones.

5.6 Ratings arrays generation

This is the phase of the project in which the actual final dataset is created, which will be used for training the autoencoder. First of all, the UserID key is isolated so that we can group all the ratings of a user with `groupByKey()` action. What

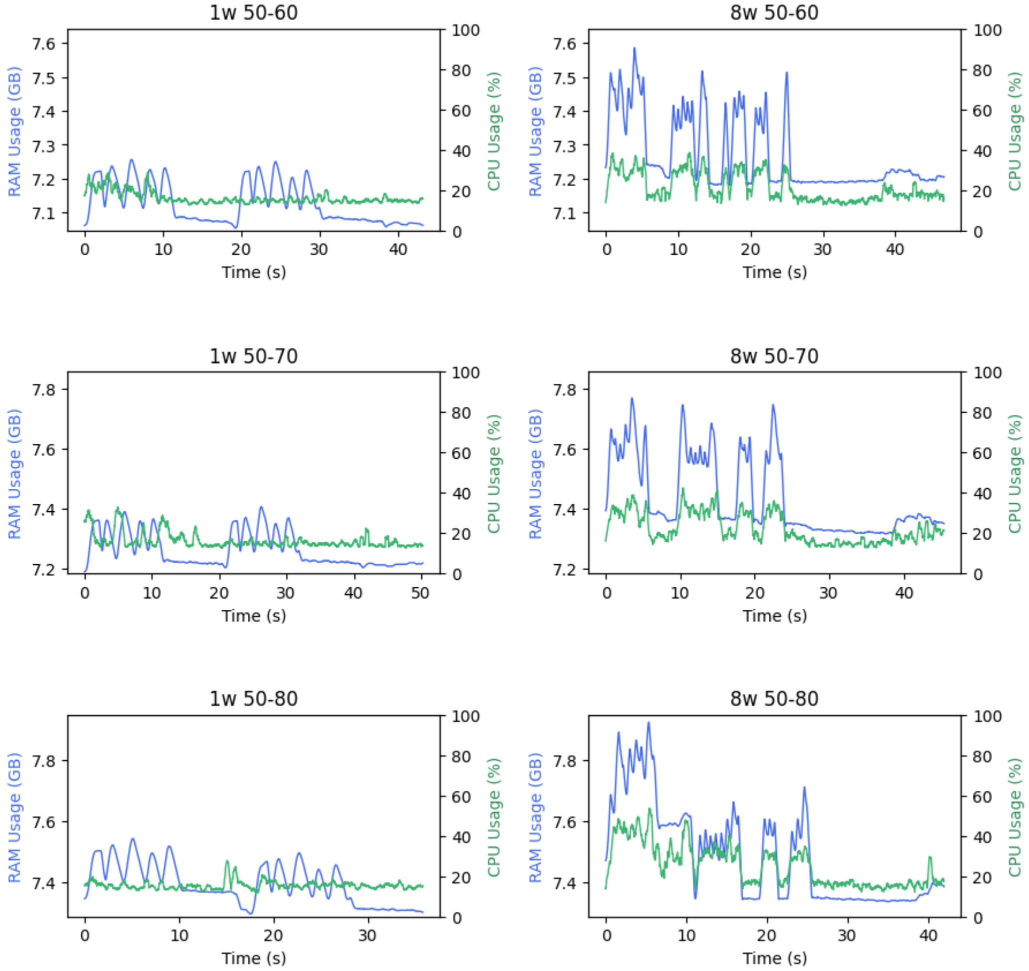


Figure 5: Statistical analysis resources usage.

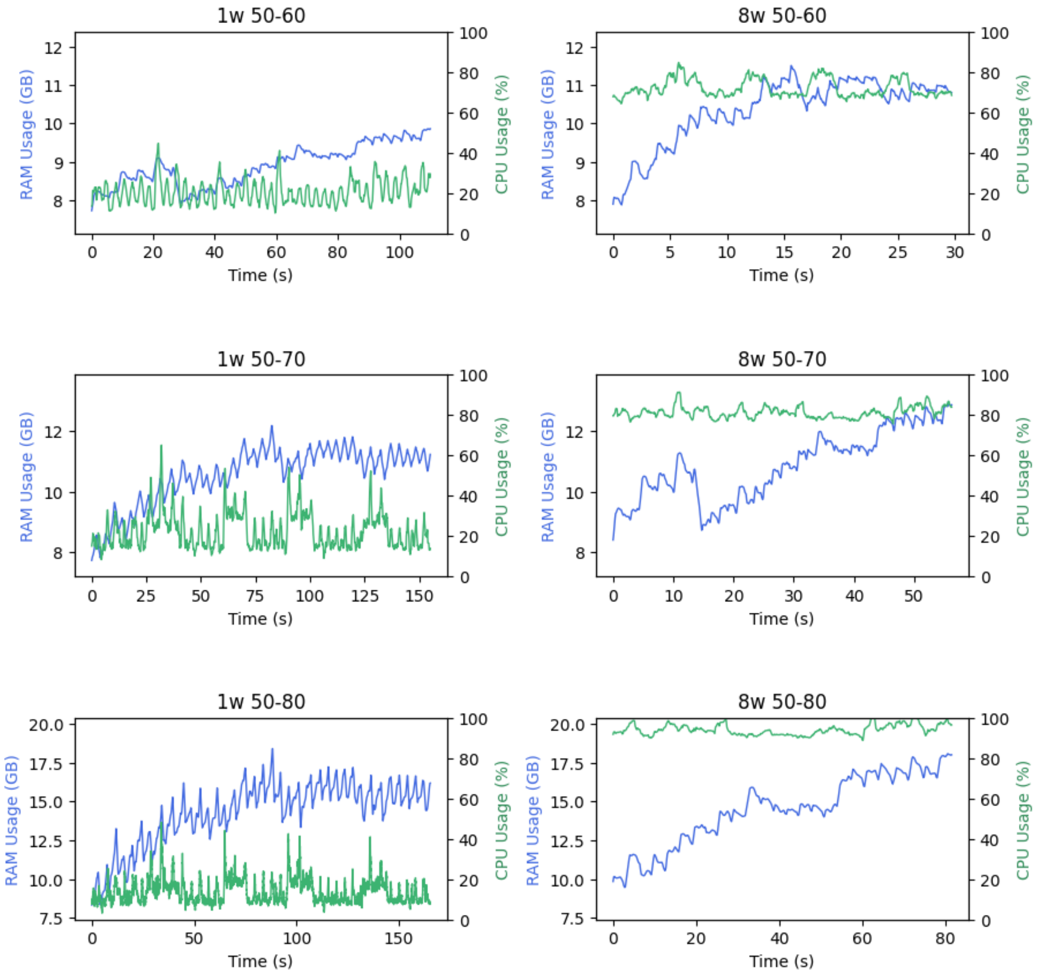


Figure 6: Ratings arrays generation resources usage.

is obtained is therefore a sequence of pairs (MovieId, rating) grouped based on the UserId. The second step is to create the actual ratings array, immediately initialized as a series of as many zeros as the number of films, value calculated in the previous phase. The movieId value present in each tuple is used to select the correct index of the array in which to write the corresponding rating. The index is found checking in a dictionary produced in the previous phase. Once created, all the users' ratings arrays are then normalized, i.e. the values are scaled into the range [0, 1]. At this point the dataset is ready to be used for model training.

Charts(fig.6) immediately show how the memory used increases hugely during the computation, since all the values to create the ratings array are generated in this phase. Clearly, the level of how much memory increments depends on the size of the dataset, and we can make some considerations on the total weight of the ratings array with respect to the memory used. If for each user we have a rating array of as many values as there are films in the subset, to find the number of total values it is sufficient to multiply the number of users with the number of films (table 1). We obtain that the subsets, in ascending order of size, generate 217 M, 641 M and 1.46 B values respectively. The multiplication factor is 2.5× between the first two subsets and just over 2× between the second and third subsets.

There is a correlation between the number of values and the memory used. In 1W the memory grows about 1.50 GB in the 50-60 case, by around 3.50 GB in 50-70 and more than 6.50 GB in 50-80. In the three cases, how much the memory grows between the beginning and the end of the computation is in line with the previous multiplicative factors.

Even in the 8W we can see this fact, with the memory increasing by approximately the same amount although there seems to be a slightly higher memory usage. My consideration regarding this fact is always due to the parallelization which favors greater data processing in respect to the same time interval. As usual the CPU is castrated at around 20% in 1W while in 8W it is used for around 90% more or less. In terms of execution times, Spark's parallelization is noticeable although I would have expected a greater advantage offered by the multi-core cluster compared to the single-core one.

5.7 Model training

In this section I present the resources usage to train the model. In the following sections I will evaluate its performance in various use cases. Few data transformations take place in this phase.

First of all, the dataset is split into train and test_set. At this point we could start training the model, if Elephas did not require the RDD to be used as training_set is in the form (input, target). Since this model is an autoencoder, input and target are the same and therefore it is necessary to replicate each vector also in the target position, effectively doubling the size of the RDD relating to the training set. This is a major flaw of Elephas as it requires unnecessary data redundancy and waste of memory, which could have been used

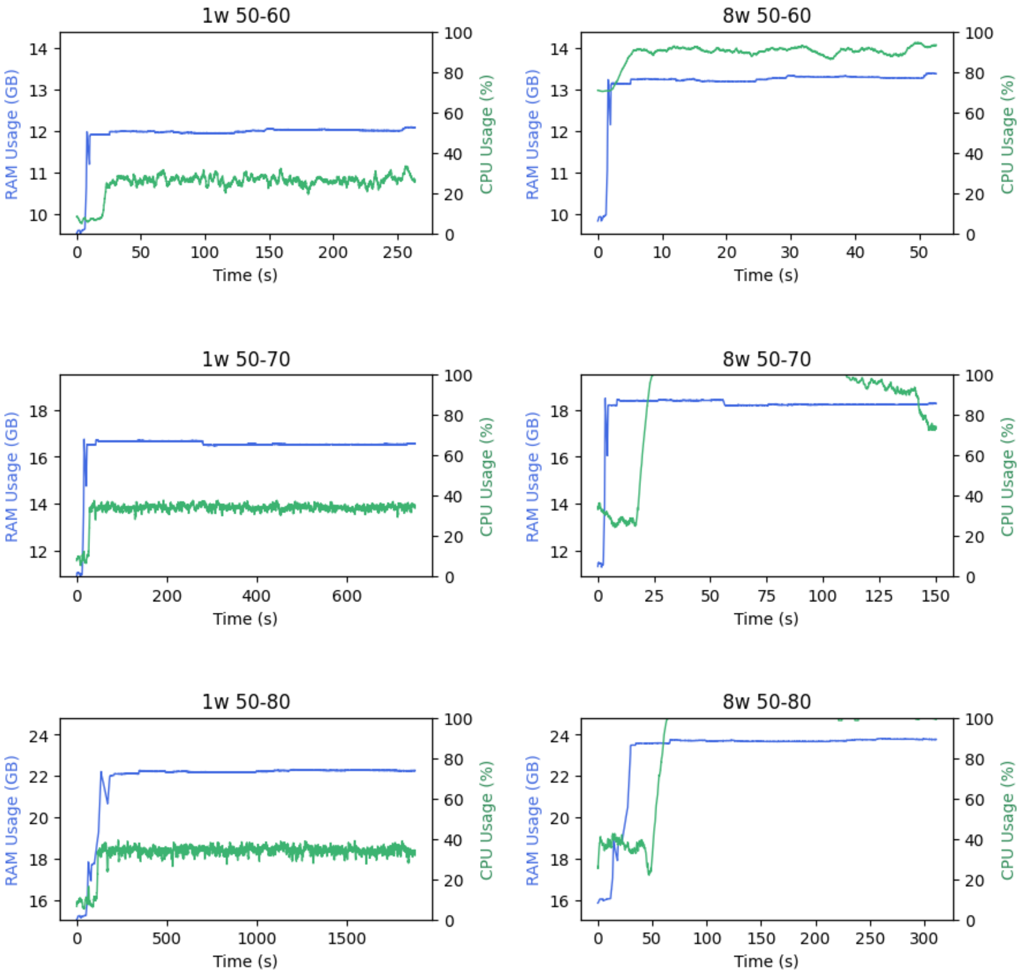


Figure 7: Model training resources usage.

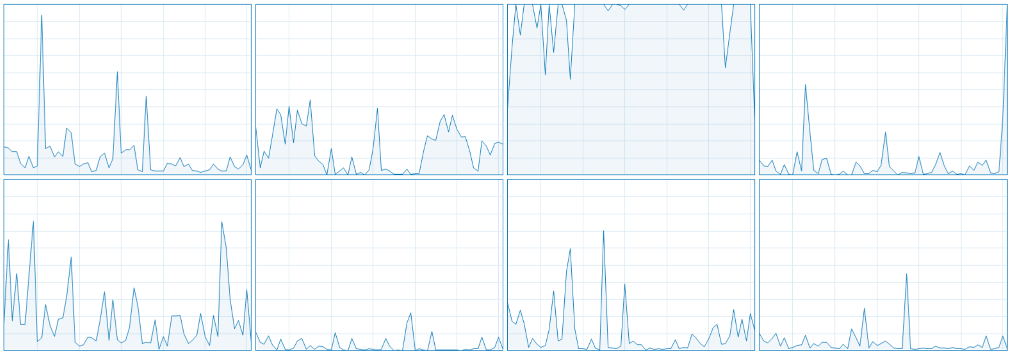


Figure 8: CPU cores usage in 1W configuration. Only 1 core working.

to analyze larger datasets. Once the RDD of the train set has been transformed into the format (input, target), the training of the model starts by running the Elephas instructions.

From the graphs(fig.7) we see how training is a rather stable process: there are no particular changes in memory or CPU usage in any of the configurations. The RAM remains almost constant in any case, at values proportionate to the subset used. In fact, during this phase there are no data transformations that imply elimination or creation of new information and it is logical that the memory used remains almost constant. One difference worth mentioning is that the levels of memory usage differ between the same subset. For the same dataset, it must be considered that distributed training must manage many more resources at play. It includes 8 different models allocation plus the central aggregation one, unlike 1W which only has one. Furthermore, the training iterates over 8 training batches simultaneously, 1 per worker. In 1W, having only one model, only one batch runs at a time. Finally, both the cluster and the distributed training process must be managed, which certainly require additional resources.

The CPU is forced to use a single core in the 1W case while it is used almost 100% throughout the 8W training phase. Speaking of execution times, the training phase is the one that most highlights the effectiveness of parallelization. In the 50-60 and 50-80 cases the distributed training is completed approximately 5 times faster than in the classic mode, while in the 50-70 case approximately 4 times faster.

Image 8 shows that in the 1W case only one CPU core works. The screenshot was taken from the task manager during model training, but is representative of all single-core tasks performed.

5.8 Model evaluation

In the fig. 9 charts, the train loss curve is shown in blue while the validation loss curve is shown in red. A shortcoming of Elephas is that once training

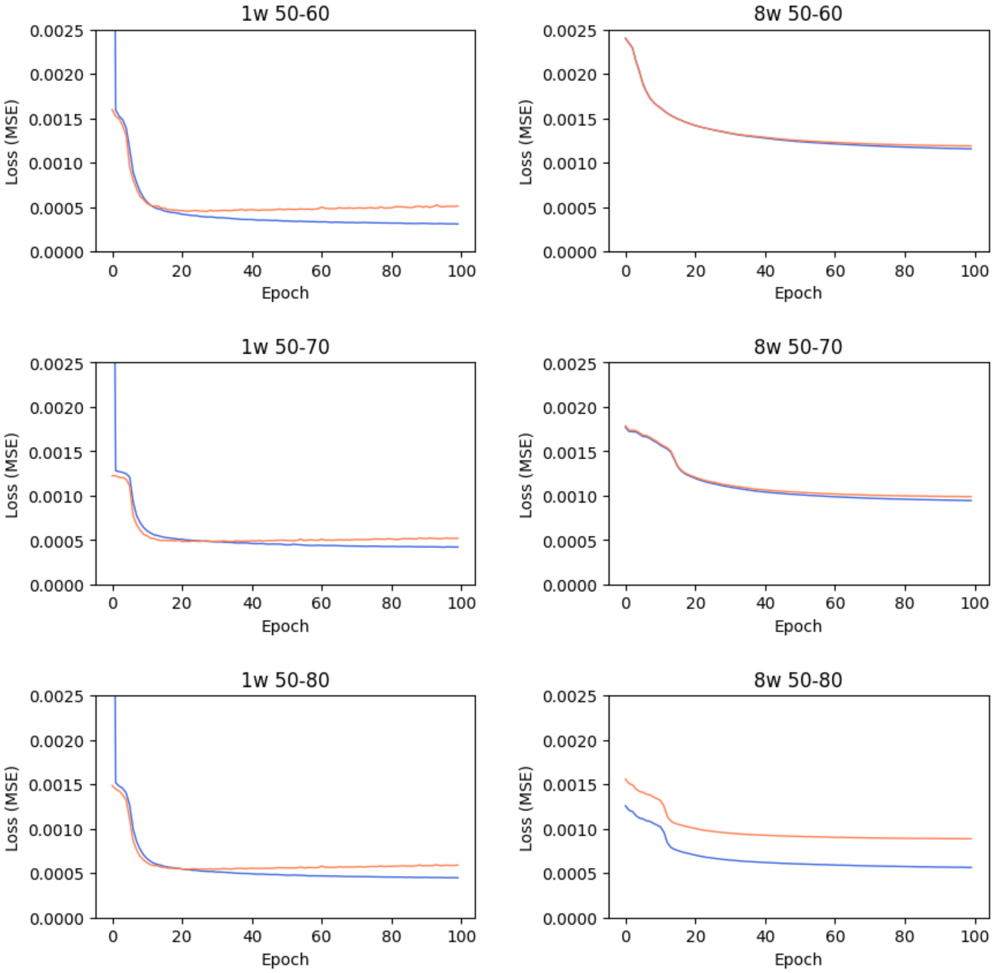


Figure 9: Model training losses.

config	**train loss**	**test loss**	**loss divergence**
1W 50-60	0.000345	0.000471	36.83 %
1W 50-70	0.000451	0.000501	11.00 %
1W 50-80	0.000479	0.000559	16.69 %
8W 50-60	0.001241	0.001254	1.02 %
8W 50-70	0.001014	0.001040	2.61 %
8W 50-80	0.000604	0.000914	51.27 %

Table 2: Train and Test sets losses.

is completed it does not provide the loss values for each epoch as happens with Keras, so I had to implement a script that made the training proceed from epoch to epoch by calculating at the end of each train and validation loss(`evaluable_training.py`).

As a first evaluation I focus on the loss curves. It is noted that classical training generally has a much steeper learning curve than distributed training. Furthermore, the curves settle very quickly, already before the twentieth epoch. This is certainly due to the complete availability of the samples to the model. In fact, given the same dataset, a model trained locally has all the samples, while a model trained on a node that works in parallel with others has an eighth of the samples available as these must be divided among the workers. Since the batch_size is 64 in both cases, the classic model has more iterations of weight adjustment for each epoch, thus faster curves settling. I decided to stop the training at the 50th epoch as you can notice that, around the 30th epoch in 1W and the 50th in 8W, the training and validation loss curves start to "detach", so the model is starting to overtit on the training set, lacking generalization. Since we can observe, for the same dataset, models trained locally always have loss curves that reach lower values than their distributed counterparts.

Let's now move on to the analysis of the behavior of the models at the 50th epoch, the training stopping point, looking at the table 2. As regards the 1W models, as the size of the dataset increases, despite more samples being available, the performance worsens. Loss values increase as the dataset size increases. This is due to the fact that the model struggles to replicate so many features. From table 1 we see how in the smaller subset there are 2.8k films with 77k ratings arrays(users) available, while in the larger case there are "only" 136k users available for more than 10k movies. As the dataset grows, the samples/features ratio drops and consequently the model struggles with learning.

In the case of distributed training, however, we notice the opposite behavior:

as the size of the dataset (and number of features) increases, the model performs better. This partly depends on the fact that as the number of users increases, each worker has more samples available, so the workers' local models are trained more accurately and consequently the aggregated model also benefits.

Despite this, it is clear that for the same dataset, the models trained locally perform better than those trained on the cluster, both on the train set and on the test set.

Another benefit found with distributed training is less overfitting. Loss divergence was calculated as $(test\_loss - train\_loss) * 100$. The 1W models have a divergence between trining loss and test loss that is on average higher than those trained on the cluster. In the 8W 50-60 and 70 cases the overfitting stands at only 2%. The 8W 50-80 case is very anomalous, in fact its divergence is 51.27%. However, it must be considered that its loss values are very close to those of its 1W counterpart and also that its training time is 5 times less, as we have seen from the previous section.

6 Recommendations

Here I explain the last part of the project's code lines, the ones relating to how perform recommendations.

First of all, the model to be used must be loaded, which was produced and saved during the training phase. I chose to use Keras as there is no need for distributed computing here.

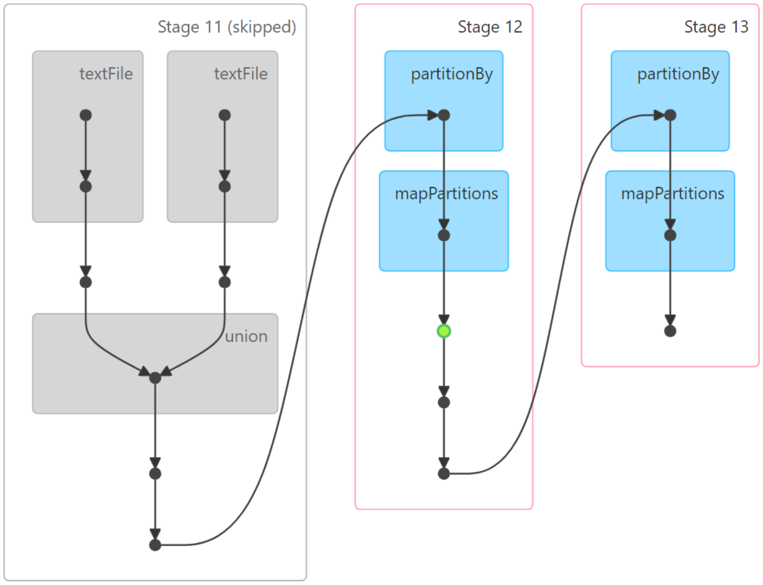
Subsequently a user must be selected to do the test. The RDD of all user ratings arrays is used to pick one at random. `rdd.takeSample()` method allows you to specify how many samples to randomly take from an RDD. Picked an array, this is processed by the autoencoder and its features are reconstructed.

`np.stack()` creates a "merged" array by stacking the original array, the reconstructed array, and a list of indexes as long as the number of movies in the subset. This is used to align the information relating to films' index, ratings and corresponding reconstructions in a single *np.array*. Immediately afterwards, "merged" is filtered maintaining only those indexes that have a rating of zero in the original array, therefore those films that the user has never seen.

From this filtering, one of the indexes aligned with the maximum rating available is selected. The extracted index is used to retrieve the information relating to the film, which is stored in the dictionary produced during phase 5 (statistical analysis of the subset).

7 DAG visualization

In this section I don't present results but I want to comment on a Directed Acyclic Graph (DAG) that emphasizes Spark properties. The configuration is 8W, therefore on cluster to have parallelization. The DAG shown is for the `aggregated_user_ratings.count()` job. This refers to the `no_monitor` python notebook which does not include code to log resource usage to display Spark related code more cleanly.



Jobs are sequences of *stages*, which can be seen as sequences of narrow operations. By ”narrow” tasks we mean those that manage to complete an operation without having to interact with other machines in the cluster. ”Wide” tasks, on the other hand, need to interact with the entire RDD and therefore require communications between workers and consequently the use of the network. Jobs are therefore sequences of stages, which are delimited by wide operations.

The Job taken as an example performs the count of the ratings arrays but first it is necessary to complete the generation process of the latter.

This job is composed by 3 stages. In the first we note the loading of ratings and movies *.csv* files which are treated with narrow operations, therefore transformations dependent on the single block of the RDD. We can confirm this because they are contained in a single stage. The results then enter the ”union” block, which refers to the join() operation that aggregates the two RDDs.

At this point we notice how this operation involves the interaction of different machines(wide task) since all the data of the RDD is required in order complete the task. Consequently, the first stage ends and a new stage begins creating the resulting RDD of the join operation.

This stage continues until the groupByKey() is performed, which collects the ratings of each user. Clearly a user’s ratings are spread across multiple machines, since these being in an RDD. The last stage concerns the creation and normalization of the ratings arrays.

The first stage (stage 11) is ”skipped” because spark had already performed those tasks for previous jobs, in particular to collect the subset statistics. So spark does not do all the operations from scratch but it does optimizations on the execution plan to minimize unnecessary computations.

8 Conclusions

Elephas: I begin the conclusions of the project with a small account of the use of Elephas. As far as ease of use is concerned, it is excellent as with just a few lines of code you can start distributed training on a cluster, but it has some shortcomings. First of all, the impossibility of having the loss history once the training was completed. I had to create a script from scratch by proceeding the training from epoch to epoch and calculating the losses at the end of each one. Furthermore, in the case of using an autoencoder, the RDD (input, target) format is certainly an inefficient solution as it requires wasting large amounts of memory that could have been used to analyze a larger portion of MovieLens.

Distributed vs classic model training: Speaking purely of performance, the results obtained show how classic training is more effective. As expected, for the same dataset classical training allows for better refinement of the model given that all samples are available on a single node. On the other hand, it is interesting to note the opposite behavior of the two cases: in the classic mode, as the dataset and features grow, the performance worsens, while in the 8W case the performance improves. It would have been interesting when there would have been a trade-off between the two modes, i.e. understanding at what size of the dataset (and increasing number of features) the distributed mode guarantees the best performance. However, it must be considered that cluster computation allows training to be approximately 5 times faster and also less overfitting.

Cluster vs single worker: Regarding data manipulation performance, I would have expected faster parallel computation than what I achieved. I set 8 workers since the total logical cores are 8, even if the physical cores are actually 4. From the results obtained, parallel computation takes approximately 3 to 5 times less than single-worker computing. One could expect orders of higher speed, around 8 times as the number of workers, but probably being a simulated cluster on a single machine, the system was limited by the only 4 physical cores, a number that aligns with the speed ratio factor found. Clearly 8 logical cores have lower performance than 8 physical cores. If we take into account the workers-driver communication times and the aggregations of the results, we should obtain that distributed computing is something less than 4 times faster and in fact in some experiments this is the case. It is also true that the results are slightly altered by calling `rdd.count()` at the end of each phase, which could be faster in single computation since it only returns the number of indices in the partition without having to aggregate the counts as occurs for parallel computation.

Recommendation system: I used the goal of creating a recommender system to serve a purpose for this project which clearly aims primarily at analyzing the performance of different technologies such as Spark and distributed training. I therefore did not focus on the creation of the system itself.

In fact, the system I propose has limitations. It assumes that zero ratings are films that the user has never seen when in reality it could be that a review simply wasn't left after viewing. Filtering should be integrated to include only unseen films in the recommendations.

Furthermore, if new items are inserted into the catalog they cannot be included in the model in use as it has been trained on the movies previously present. Also the input array size is fixed. The same problem arises if items are removed from the catalog.

Regarding the memory used, the entire pipeline is very demanding of it. In the largest subset analyzed there were only 10K movies out of 62K and 2.7M ratings compared to the total 25M. So we were using around a ninth of the total dataset and despite this the peak RAM used is around 23 GB. To analyze the entire dataset, several times more memory would have been required.

Final conclusions: I took the excuse of creating a recommendation system to integrate several concepts that I thought were interesting. First of all, Spark. I was able to experiment with its use and test its properties. Observe how different data sizes can affect machine performance and what advantages a cluster can bring compared to classical computing. I learned how data can be integrated between tools, like Spark and Elephas and which are the most important aspects to consider when handling with such large datasets. Furthermore, I was able to experiment under which conditions a Deep Learning model trained on cluster can lead to advantages or disadvantages.