Neural Collaborative Filtering

Giovanni Burbi
Damiano Giani
Dipartimento di Ingegneria Informatica

Report of the data mining project
Exam: Data and Documents Mining
Professor: Simone Marinai

12th August, 2022

Abstract
We tried to reproduce the work of the paper "Neural Collaborative Filtering" written by Dr. Xiangnan He et all and compared the results of the experiments we carried out with theirs. The novel idea of the original work is to apply techniques based on neural networks in the context of the recommendation problem, in particular collaborative filtering, on the basis of implicit feedback.The result of the research conducted by the authors led to the birth of a general framework named NCF which can be used to express, generalize and extend Matrix Factorization thanks to its neural architecture. This framework can also model the interaction between user and item, the key factor in collaborative filtering, using a Multi-Layer Perceptron that allows the framework to learn high level non-linear features from the data. We carried out experiments on two real-world dataset, MovieLens and Pinterest, following the guidelines given by the original paper. The empirical results showed that fusing the linear capabilities of Matrix Factorization and the non-linear ones given by the Multi-Layer Perceptron allow the instance of the framework to better learn from the data and modelling the interaction function offering better recommendation performance.

1 Introduction

The purpose of this project is to reproduce the results obtained in the paper titled "**Neural Collaborative Filtering**" written by Dr Xiangnan He. The idea behind this research article is to apply **deep neural networks** on recommendation systems on the basis of **implicit feedback** datasets. In general **recommender systems** are algorithms aimed at suggesting relevant items to users. There are different approaches to design recommender systems; in this paper we will consider **Collaborative Filtering** methods.
The key factor in collaborative filtering is the modelling of the **interaction** between user and item based on their past interactions, like reviews, ratings or clicks. These interactions are stored in a matrix called **user-item interaction matrix**. The data that we can work with in collaborative filtering methods can be in explicit or implicit form. In this work, we will consider only **implicit feedback,** which indirectly reflects users preferences through behaviours like watching videos, purchasing products and clicking items.

One of the advantages of this approach is that we don't require information about users or items but only their interactions. The more interactions we have for a certain set of users and items, the more accurate the recommendations will become. Implicit feedback can also be **tracked automatically** and is thus much easier to collect.
The downside of using this type of feedback is in its utilization, since user satisfaction is not observed and there is a **lack of negative feedback**. In the following sections, we will see how we can make up for this aspect.
Among collaborative filtering techniques, **Matrix Factorization** (MF) was the most popular one at the time of the writing of the paper we want to reproduce. This technique puts users and items into a shared latent space, using two vectors of features to represent a user and an item. Then it uses the **inner product** as the interaction function. The performance of this technique depends on this interaction function. Since the inner product may not be enough to capture the complex structure of user interaction data the authors of the original paper decided to use **deep neural networks** for learning the interaction function from data directly. The result of this research is a **neural network framework** (NCF) for collaborative filtering that models latent features of users and items.
It's possible to derive MF as a specialization of NCF and it's also possible to utilize a multi-layer perceptron to endow NCF modelling with a high level of non-linearities.

2 The Problem

The recommendation problem with implicit feedback is the problem of **estimating the scores of unrecorded entries** in the interaction matrix and creating a **ranking of items for each user**.
Given a dataset of implicit feedback containing a certain number of distinct users and items, we can define the user-item interaction matrix.
The interaction matrix will have value 1 in the position (i,j) if an interaction recorded for user *i* and item *j* exists, zero otherwise. In this context, a value of zero doesn't necessarily mean that user *i* doesn't like *j* but it may be that the user **doesn't know the existence of that item yet**. This is why implicit feedback naturally doesn't have negative feedbacks.
Generally, the aim of model-based techniques, which is the main focus of this work, for solving the recommendation problem is to **learn a function to predict the score of the interaction** between a user and an item. The score can be seen as the **likelihood** that a user likes a certain item that he hasn't interacted with yet.

3 Neural Collaborative Filtering Framework

Assuming implicit data, the authors of the original paper created a **probabilistic model** to exploit the binary property of implicit feedback that is the basis of the NCF framework. This framework can generalize the popular model Matrix Factorization (MF) and it can also be instantiate using a **Multi-Layer Perceptron** (MLP) to learn the user-item interaction function. Futhermore, it is also possible to instantiate a new neural model that **combines MF and MLP**, unifing the **linear proprieties** of MF and the **non-linear proprieties** of MLP for modelling the user-item latent structures. The neural part of the NCF framework consists of a multi-layer layout to model a user-item interaction.
The bottom layer consists of two vectors of features that describe a user *i* and an item *j*. Then, there is the **embedding layer** that generates dense vectors in a certain latent space

from the sparse representation of the inputs. Above this layer, there is the **multi-layer neural architecture** that maps the latent vectors to prediction scores. The dimension of the uppermost hidden layer denotes the **model's capability**. The final layer is the output that produces the predicted score. The score returned by the output layer is constrained in the range of [0,1] by using a probabilistic function as the activation function for the layer. Lastly, the training is performed by minimizing the **binary cross-entropy loss** between the predicted score and its target value. Figure 1 shows the general layout of the NCF framework.

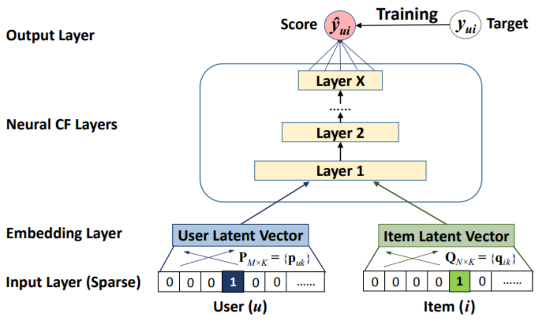

Fig. 1. Neural collaborative filtering framework

3.1 General Matrix Factorization (GMF)

The first model we see is Matrix Factorization (MF) that we can generalize (**GMF**) using the NCF framework. We will utilize the MF model as a sort of baseline for our experiments later on. MF uses the **inner product** on the latent vectors of a user and an item to estimate their interaction. Specifically, this model associates each user and item in a dataset with a vector of latent features in the same latent space by performing the embedding and then it **linearly combines** their latent factors with the same weight. For this reason, we can say that MF is a **linear model of latent factors**. Lastly, we project the combined vector to the output layer to obtain the predicted score.
With the NCF framework **we can derive the standard MF model** by using in the output layer the identity function for the activation function and setting all its weights to 1.
We can take a step further thanks to the NCF framework; we can **extend** the model allowing the weights in the output layer to be **learned from data** directly using the log loss, also called **binary cross-entropy loss.** This way we allow the model to attribute different importance to latent dimensions. Futhermore, we can use a non-linear activation function, the **sigmoid**, to give more expressiveness to the linear model MF.

3.2 Multi-Layer Perceptron (MLP)

In order to learn the **non-linearity structure** of the interaction between user and item latent features we can use a simple **Multi-Layer Perceptron**.
Conversely of GMF, which uses only a fixed element-wise product to learn the interaction function, using MLP endows the model with a **large level of flexibility and non-linearity**.
For the design of the network's hidden layers structure, we use the **tower pattern** where the bottom layer is the widest and the size of each subsequent layer is halved compared to the upper layer. This way, the higher layers with a small number of hidden units can learn more **high-level features** of data.
For activation functions of MLP layers, we use the **Relu function**, which is well-suited for

sparse data and makes the model less prone to overfitting compared to the sigmoid or the tanh.

3.3 Neural Matrix Factorization (NeuMF)

The innovation brought by the original paper consist in the **fusion** of the two previous models under the NCF framework which combine the linear kernel of GMF and the non-linear kernel of MLP in order to **mutually reinforce each other** and learn to **better model** the complex user-item interactions.
Despite GMF being a non-linear model due to its output function, in NeuMF models, only its linear part composed of the hidden layers is used.

Figure 2 shows the **Neural Matrix Factorization model** (NeuMF) which is the fusion of GMF and MLP obtained by concatenating their last hidden layer. An important aspect to highlight in the NeuMF model is that the model allows GMF and MLP to learn **separate embeddings** instead of sharing the same embeddings that would potentially limit the performance of NeuMF. This is because, for some datasets, the optimal embedding size of the two models may vary a lot. The activation function for the MLP layers is still the **ReLU function**.

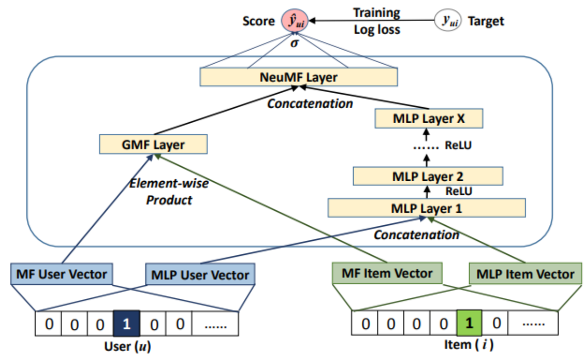

Fig. 2. Neural matrix factorization model

3.4 Pre-training

To **optimize** the objective function of NeuMF we use **gradient-based methods**. These methods only find locally optimal solution for non-convex functions, therefore the **initialization is really important** for the performance of deep learning models. Since NeuMF is made up of the fusion of GMF and MLP, we can initialize NeuMF with the **pre-trained models of GMF and MLP**. Therefore, after we initially train GMF and MLP until convergence, we use their weights to initialize the corresponding parts of NeuMF.
In the article the authors proposed to initialize the last layer of NeuMF with the **concatenation of the weights** of the last layer of the two models weighted by an hyper-parameter that determine the **trade-off** between the two pre-trained models. In our experiments, we will see how different values of the trade-off parameter influence the performance of NeuMF. Finally, after the initialization with pre-training of NeuMF, we proceed with **fine-tuning** the model.
The fine-tuning process is done only on the output layer of NeuMF, so that NeuMF is able to learn how to best utilize and combine the features learned by GMF and MLP.

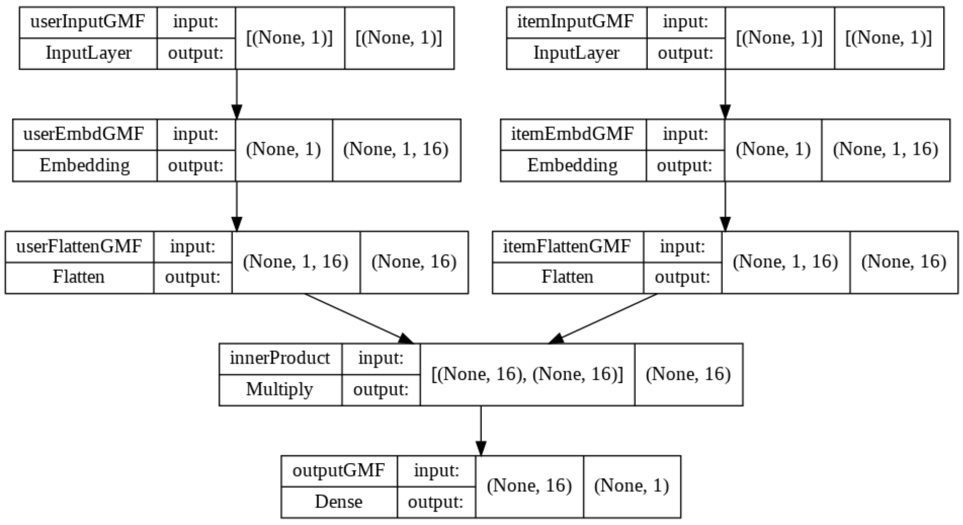

Fig. 3. Keras visual representation of our implementation of the GMF model

4 Implementation

For the implementation of this project has been used **Keras**, a deep learning API written in Python, running on top of the machine learning platform TensorFlow. The training process from scratch of the three models is done by using Adaptive Moment Estimation (**Adam**) optimizer that yields faster convergence respect to SGD and relieves the pain of tuning the learning rate. We saved the weights of each model during the training process when we obtained the best metrics.

4.1 GMF

Figure 3 is show the GMF model implemented in Keras. This model takes as inputs two 1-dimensional Keras tensors representing the userId and the itemId. We then created two embedding layers that turn each input tensor into a **dense vector** with the dimension of the selected latent factor. We proceeded to **flatten** these vectors and then we performed the **element-wise product** between them. Finally, we defined the output layer as a 1-dimensional dense layer with a sigmoid activation function. The weights of the embedding layers and output layer are randomly initialized with a **normal distribution** (mean 0 and standard deviation 0.01).

4.2 MLP

The MLP model, just like GMF, takes as inputs two 1-dimensional Keras tensors representing the userId and the itemId. Then, based on the **predictive factor** selected, we computed the dimension of the embedding layers with:

$$embeddingDim = \frac{predictiveFactor * 2^{numHiddenLayers-1}}{2} \tag{1}$$

where *numHiddenLayers* is the total number of hidden dense layers plus the embedding layer.

After the creation of the input's tensors, we defined the embedding layer for the inputs with the dimension obtained from the equation above and then we flattened the embedding vectors. We then combined the features of the two input's embedding with a Concatenate layer and defined the dense hidden layers of the MLP model. The first dense hidden layer has dimension

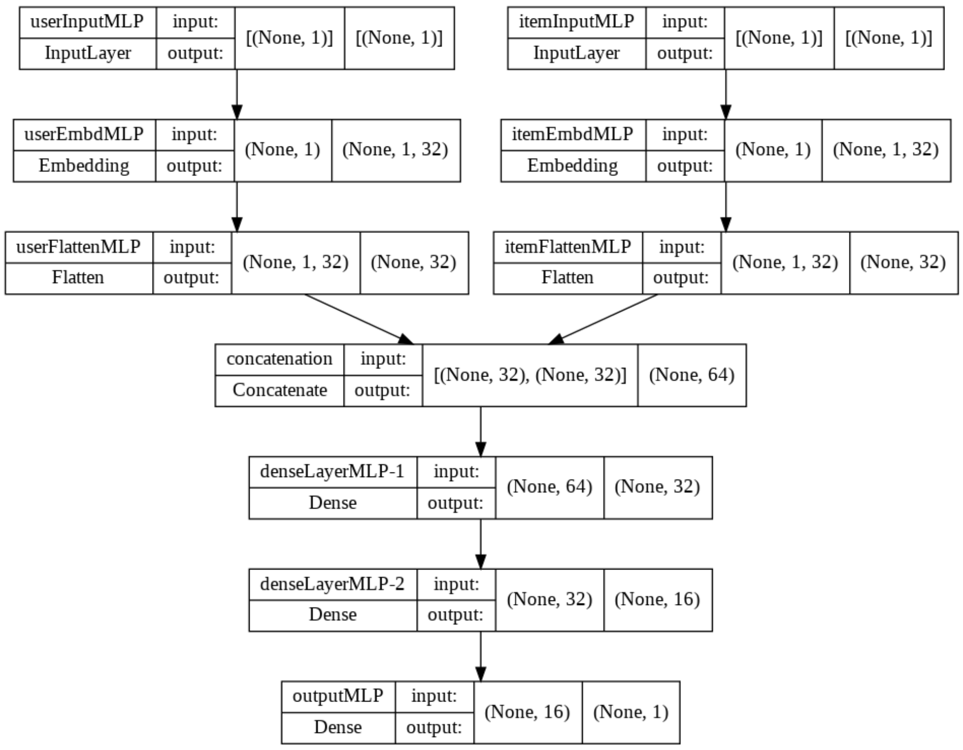

Fig. 4. Keras visual representation of our implementation of the MLP model

equal to half the dimension of the embedding. The last dense layer instead has dimension equal to the predictive factor. All the dense layers use the **Relu activation function**. Finally, the last layer of the model is the output layer that use the **sigmoid activation function**.

4.3 NeuMF

As we have previously seen, Neural Matrix Factorization model also takes as inputs two keras tensors representing a userId and an itemId. As we can see in figure 13, under the NCF framework, we provide flexibility to the model by allowing the two build-in models to learn **separate embeddings**, and then we combine GMF and MLP by **concatenating their last hidden layer** with a Concatenate layer. Once again, the output layer is a dense layer with a sigmoid activation function.

4.3.1 Pre-Training

Given that the **initialization plays an important role** for the performance of deep learning models, especially when using gradient-based optimization methods like SGD or Adam, we can initialize NeuMF using the **pre-trained models of GMF and MLP**. To do this with Keras, we first train until convergence the GMF and MLP using Adam. Then we load the weights of the hidden layers in NeuMF using the name of the layers to match the ones in the two pre-trained models. Lastly we set up the output layer of NeuMF with the concatenation of the weights and bias of the last hidden layer of the two models. After we initialized NeuMF with the pre-trained models, we can perform a **fine-tuning** for few epochs to improve the performance of the network by allowing the model to adjust the weights of the output layer to the data used for training. To perform the fine-tuning, we froze all layers except for the output by setting to False the attribute 'trainable' of the layers. We also used a **lower learning rate** than the one we used to pretrain GMF and MLP to avoid overfitting from the start of the fine-tuning process. We followed the original paper in using the **SGD optimizer** instead of

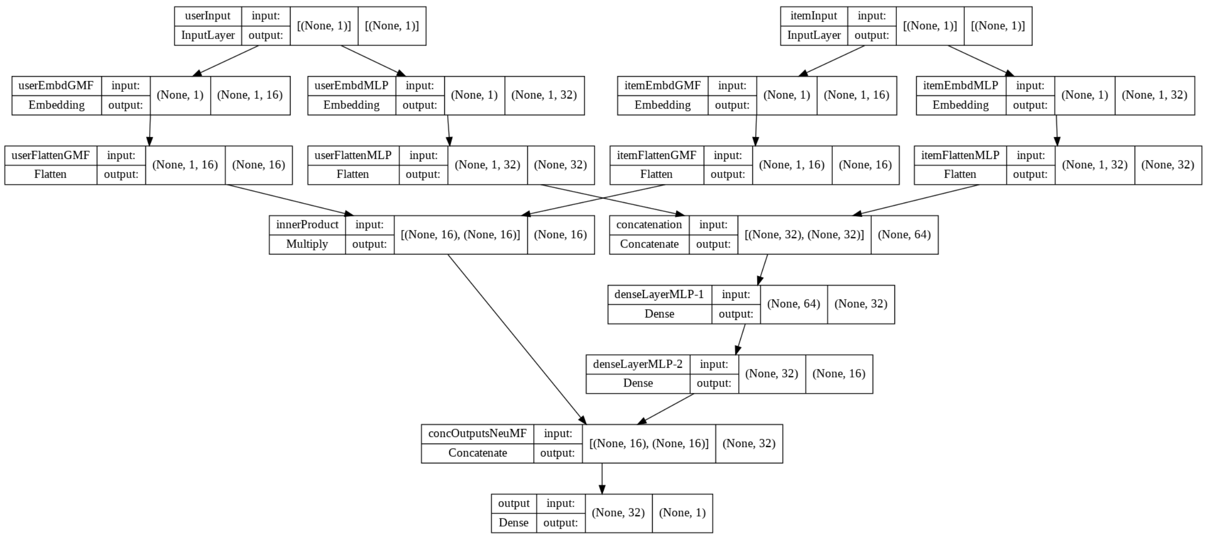
userInput InputLayer input: output: [(None, 1)] [(None, 1)]
itemInput InputLayer input: output: [(None, 1)] [(None, 1)]
userEmbdGMF Embedding input: output: (None, 1) (None, 1, 16)
userEmbdMLP Embedding input: output: (None, 1) (None, 1, 32)
itemEmbdGMF Embedding input: output: (None, 1) (None, 1, 16)
itemEmbdMLP Embedding input: output: (None, 1) (None, 1, 32)
userFlattenGMF Flatten input: output: (None, 1, 16) (None, 16)
userFlattenMLP Flatten input: output: (None, 1, 32) (None, 32)
itemFlattenGMF Flatten input: output: (None, 1, 16) (None, 16)
itemFlattenMLP Flatten input: output: (None, 1, 32) (None, 32)
innerProduct Multiply input: output: [(None, 16), (None, 16)] (None, 16)
concatenation Concatenate input: output: [(None, 32), (None, 32)] (None, 64)
denseLayerMLP-1 Dense input: output: (None, 64) (None, 32)
denseLayerMLP-2 Dense input: output: (None, 32) (None, 16)
concOutputsNeuMF Concatenate input: output: [(None, 16), (None, 16)] (None, 32)
output Dense input: output: (None, 32) (None, 1)

Fig. 5. Keras visual representation of our implementation of the NeuMF model

Adam to perform the fine-tuning. The fine-tuning on the output layer of NeuMF the network is done to allow the network to learn the best configuration of the weights of this layer after the initialization.

5 Datasets

In the last part of this report, we will compare our experiment's results with the original paper. To carry out these experiments, we have used two datasets: **MovieLens** and **Pinterest**.

Dataset	Interactions	Users	Items	Sparsity
MovieLens	1000209	6040	3706	95.53%
Pinterest	1500809	55.187	9.916	99.73%

Table 1
Details and statistics of the datasets used

5.1 Movielens

MovieLens dataset has been used often to evaluate recommendation systems. It contains the rating that a number of users have given to movies. There are many versions of this dataset; we used the version containing **one million ratings**, where each user has at least 20 ratings. The details of this dataset is shown in table 1. This dataset is composed of explicit feedback data, but we can learn from the implicit signal of data. In fact, we transformed the entries of the dataset into **implicit feedbacks** by marking with a label of value 1 the interaction between a movie and a user that has rated that movie, 0 otherwise. In the original paper, the values of the items have been changed without giving any explanation, effectively changing the distribution of the interactions. We used the original dataset values, so there will be some discrepancy between our experiment's results and the original paper. We will see more in the experiments section.

5.2 Pinterest

Pinterest dataset has been downloaded from the github repository of the original article's authors. Originally the data contained in this dataset are used for evaluating **content-based image recommendations**. We grouped all the data together and then we retained only users with at least 20 interactions. Each interaction means that the user has pinned an image in his own board, and therefore we have a label with value 1. The details of this dataset is shown in figure 1.

6 Datasets Pre-processing and Evaluation Protocol

From the datasets we have introduced, we need to create the input data that will be fed to the NCF framework models. After we fetched one of the datasets and extracted the number of distinct users (nUsers) and items (nItems) from the data, we created the **interaction matrix** of dimension nUsers x nItems. It was initialized as a zero-matrix, then we have set 1 in the position (userId, itemId) in the matrix, where the couple (userId, itemId) represents an interaction recorded in the dataset between a certain user id and item id. We decided to use SciPy's dok_matrix to implement the interaction matrix for performance reasons.

The next step was to **split the data** into two sets: **Train set** and **Test set**. The split rule depends on the evaluation protocol used; in our case, we used the **leave-one-out evaluation**. Following this protocol, we held-out, for each user, the latest interaction as the test set and utilized the remaining data for training. In the MovieLens dataset, each review has a timestamp associated, so we looked up to this field's value to find out the latest interaction. In the Pinterest dataset, on the other hand, we don't have the timestamp, so we decided to take for each user, their last recorded review in the original layout of the dataset. The result is two sets of data, the initial form of train and test sets, each of these containing the reviews in explicit format.

The next step was to transform these sets of data in **implicit feedbacks** by attaching a label to each couple (userId, itemId). This way, we have all the recorded interactions between users and items with labels set to 1. We **lack negative interactions**; this is the common problem of implicit feedback data. To fix this aspect we may add, for each user, all the items that the user hasn't interact with and set the respective label to 0, but it would generate sets **too big to work with**. Therefore, we uniformly **sampled** a certain number of negative samples from unobserved interactions.

Following the original paper setup, for each user, we randomly sampled 99 items from the unobserved interactions of the user and added them in the test set together with the latest observed interaction of that user. We used the interaction matrix to make sure that the negative samples generated was not part of the interactions of the user by verifying that the position (userId, randomItemId) in the interaction matrix was not 1.

We also made sure that the generated negative samples were all distinct from each other using a bucket with all item ids. We randomly picked one of these item ids from the bucket verifying that it was not interacted with the user by looking at the interaction matrix and then we added it to the test set and deleted it from the bucket. Otherwise, we only removed it from the bucket and repeated the process with a new item id. After we completed the test set by adding the negative samples, we saved it as a pickle file. To **objectively compare** the performance of all models, we used this same test set to carry out all the experiments that we will show in the experiment section.

We used a similar procedure for the training set in each training epoch using 4 negative samples for each positive instance like in the original paper. Therefore in each training epoch, we generated a new training set always with the same positive samples but with different negative instances. In the experiments section, we will show the impact on the performances using a different number of negative samples per positive instance in the training set.

7 Evaluation Method

To evaluate the performance of a model on a dataset, we need to **measure** how well the predictions made by the model match the observed data. To measure how well the predictions made by the model are, we **ranked the test items** based on the output, or score, of the network (the score represents the likelihood of appreciation between a user and an item) and then we used the **Hit ratio** (HR) and **Normalized Discounted Cumulative Gain** (NDCG) metrics to analyze the ranked list.

HR measures the percentage of times where the positive sample item is present on the top-K list. NDCG assigns different scores based on the position of the positive item giving the higher score if the positive item is in the first position.

The evaluation method we implemented takes an entry of the test set, composed by a userId, the positive sample and 99 negative samples, and creates a dictionary with a test item id as the key and its score, the output predicted by the network, as the value. It sorts the dictionary based on the score of the items and trunks it to the top-K. We will see the effect of truncating the list in different places in the experiments section. Finally, it computes the metrics for each entry of the test set and stores them in a list. The procedure ends by returning the mean of the metric lists, that is the overall Hit ratio and NDCG of the evaluation.

8 Experiments

In this section we will compare the results we obtained from our experiments with the same ones done in the original paper.

We trained our models **from scratch** by performing a certain number of epochs based on the model and its settings. Bigger models like NeuMF need fewer training epochs to reach convergence than GMF, for example. We also observed that for each model, using a big predictive factor causes **overfitting** sooner than when using small factors, but this also means that the model reach **convergence** in fewer training epochs for that factor.

In the experiments, we mainly use the **Hit Ratio** and the **NDCG** metrics to evaluate the performance of the models.

Since evaluating the metrics in each training iteration turned out to be very time-consuming with Google Colaboratory's free machines, especially with Pinterest dataset, we chose to evaluate our models every certain number of training epochs. The frequency of the evaluation depends on the dataset: using MovieLens we evaluate the models every 3 epochs, while using Pinterest we performed the evaluation with lower frequency since it took 1 hour to complete one evaluation given the large number of test entries. The consequence is that we may miss the best epoch in terms of metrics and this is the reason why the results with the Pinterest dataset are a little worse compared to the ones in the original paper.

Comparing the two datasets in the following experiments, we can notice that the values of the metrics are higher with the Pinterest dataset compared to MovieLens. In fact, in Pinterest dataset we have **more interaction** data between users and item and that allows the neural network to **better model** the interaction function and obtain **better performance** in terms of metrics.

The original paper compares the models under the NCF framework with other methods used as baselines. In our project, we didn't use them but they are reported in the plots of the original paper that we will show in the following parts. We refer further explanations on these baseline methods on the original work.

8.1 Experiments Setup

We trained all NCF models by optimizing the log loss (or **binary cross-entropy loss**) and sampled four negative instances per positive instance in each training iteration. The parameters of the models trained from scratch have been randomly initialized with a **gaussian distribution** using a mean of 0 and standard deviation of 0.01. We used batch size of 256, **learning rate** of 0.001 and **Adam** as the model optimizer. Defining **predictive factor** the dimension of the last hidden layer in the NCF framework, we evaluated the predictive factors of 8,16,32,64 in our experiments. Large factors may cause overfitting and degrade the performance of the model. In the MLP model,we have used **three hidden layers**, including the embedding layer. We will see how the number of these hidden layers affect the performance of the models.

In the original paper, NeuMF with pre-training is shown only with the trade-off between the two build-in models, the alpha parameter, set to 0.5, which allows the pre-trained GMF and MLP to contribute equally to the model's initialization. We have tried different values of the trade-off parameter, the results of this experiment will be shown later on.

8.2 Experiments Results

To understand how the different parameters of the models affect the performance of the models under the NCF framework, we replicated the experiments of the original paper on both datasets and we compared the results.

8.2.1 Metrics Varying Factor

In the first experiment, we **varied the factors** testing each model with factors [8, 16, 32, 64] . For GMF we call the factor as the **latent factor** and for MLP we call it the **predictive factor**. The latent factor refers to the dimension of the embedding space. The predictive factor, instead, is the dimension of the last hidden layer of MLP, which determines the model's capability.
As we can see in figure 6, the performance of NeuMF on the MovieLens dataset **greatly improves** by increasing the factor. On the other hand, the performance of GMF and MLP improves up to factor 32 and it remains stable with factor 64. The reason for this behavior could be that GMF and MLP suffer from **overfitting** for large factor. In figure 7 NeuMF gets worse on Pinterest dataset at factor 64 and GMF reach the max at factor 16 and than declines. On this dataset MLP is the only model that gradually improves with every factor. Our results on the MovieLens dataset are slightly **better** respect the article's results and the shape of the curves are similar. The only difference in our results is related to GMF. In the original paper's, results the hit ratio get worse after factor 32 while ours don't. Our results on pinterest dataset are slightly worse than the article's results but the trend of the curves is **similar**. The reason of this could be that, due of the dimension of the dataset, we trained our model for too few epochs.

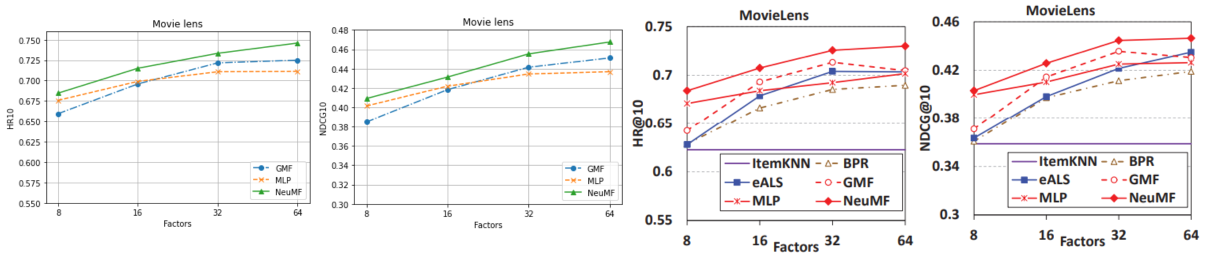

Fig. 6. Performance of HR and NDCG w.r.t. the number of predictive factors on MovieLens Dataset

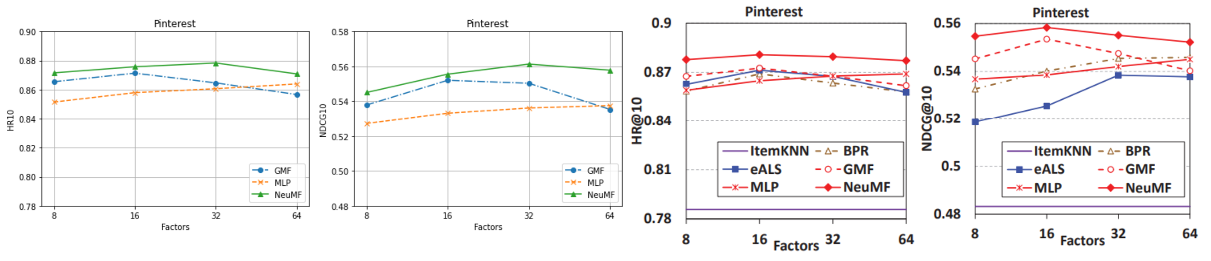

Fig. 7. Performance of HR and NDCG w.r.t. the number of predictive factors on Pinterest Dataset

8.2.2 Varying K in top K

In this set of experiments we tried to compare the performance varying the value on which we trunk the ranked list and evaluate the metrics on. Obviously, the smaller the parameter K is, the smaller the probability to have our positive instance in the top K. At the limit, with K equal to 100, the number of test instances per user, we would have probability 1 to find the positive sample in the ranked list but the NDCG metric may become really low. Therefore, it is reasonable to see the curves in figure 8 and in figure 9 have a **increasing monotonous trend**. In the article they performed this type of experiment only on NeuMF and the other baseline methods. Whereas, we performed this experiement also on GMF and MLP, and compared their results with the ones of NeuMF. Our results on MovieLens dataset are really close to the article's, while the ones on Pinterest dataset are slightly worse.

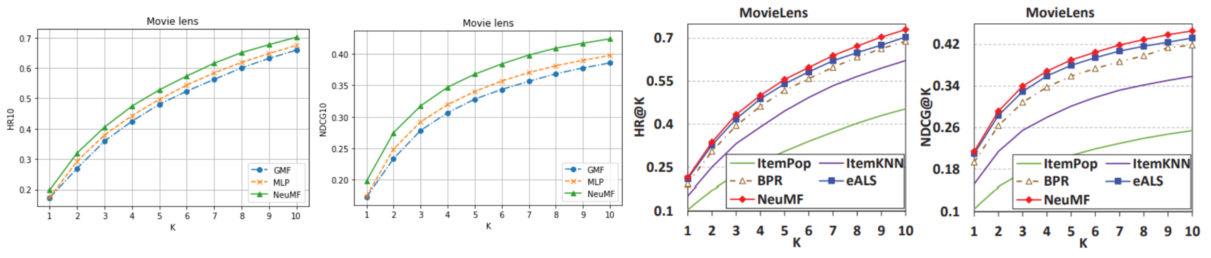

Fig. 8. Evaluation of Top-K item recommendation where K ranges from 1 to 10 on MovieLens Dataset

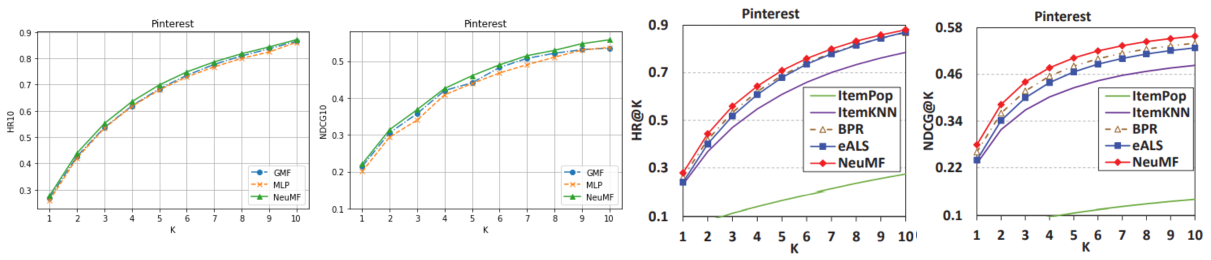

Fig. 9. Evaluation of Top-K item recommendation where K ranges from 1 to 10 on Pinterest Dataset

8.2.3 Varying the Number of Negative Samples in the Train Set

In this set of experiments we compared the impact of **negative sampling** by showing the performance of our NCF models with respect to different negative sampling ratio with the original paper. In figure 10 and in figure 11 are shown the results of HR and NDCG obtained varying the number of negative samples per positive instance in the train set. On MovieLens dataset, the lowest results for each model are obtained with one negative sample per positive instance. This may be caused by the fact that the train set doesn't give enough **information** about user-item interactions to the network to allow it to make a good modelling of the interaction function and obtain optimal performance.

Looking at figure 10. it is possible to see that on MovieLens dataset, NeuMF reach its **best result** with the value of 4 negative instances and bigger values cause the performance to drop slightly. This happens to both our results and those of the article. This is the reason why the standard value of negative samples used in the other experiments is 4. We can also see a difference between our results and the article's results over MLP's performance. In fact, in our results, MLP reaches his maximum value at 4 negative instances, while in the article's results it does it at 10 instances.

The Pinterest results we obtained have a similar trend with the original paper though our values are a bit lower.

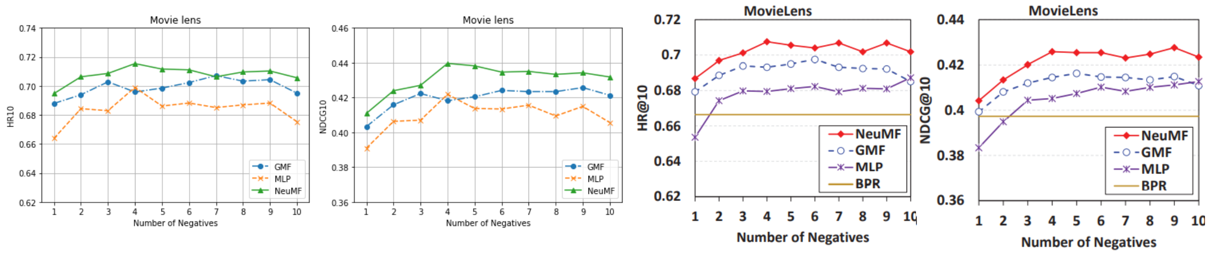

Fig. 10. Performance of HR and NDCG w.r.t. the number of predictive factors on MovieLens Dataset

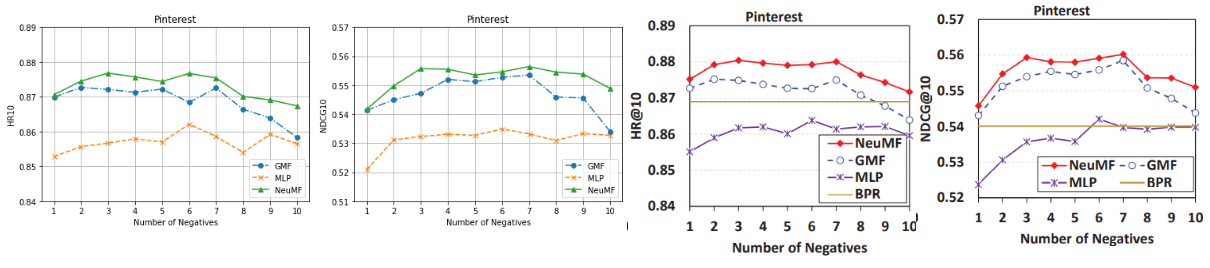

Fig. 11. Performance of HR and NDCG w.r.t. the number of predictive factors on Pinterest Dataset

8.3 Performance over the training epochs

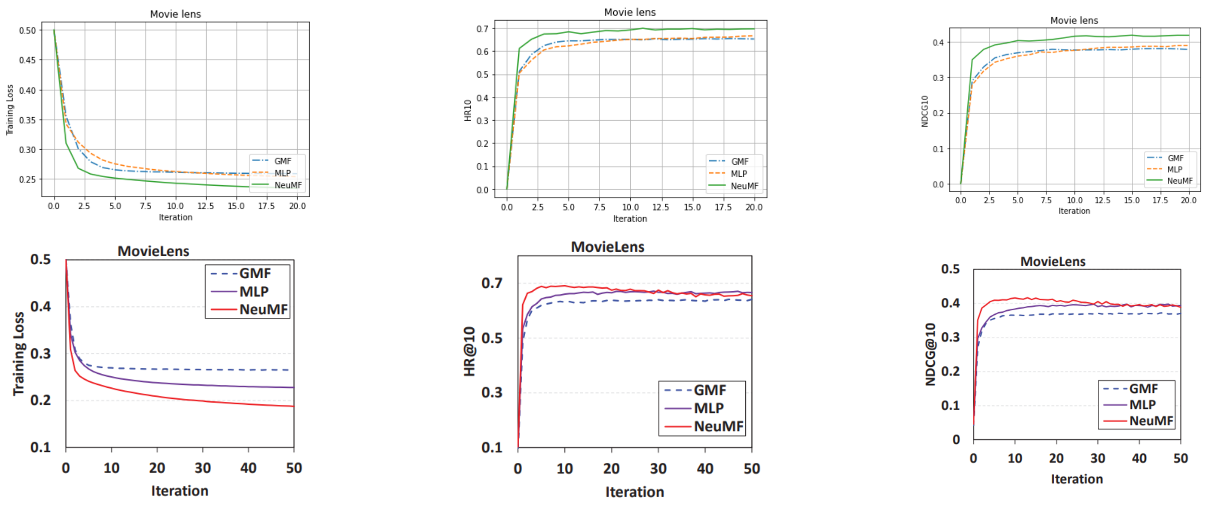

Fig. 12. Training loss and recommendation performance of NCF methods w.r.t. the number of iterations on MovieLens (Factor = 9

In figure 12 we show the training loss and the metrics of NCF models for **every training epoch** on MovieLens dataset. We show fewer iteration than the original paper, but they are enough to compare the trends. We can see that in both the cases, with more iterations, the loss **gradually decreases** and the performance in terms of metrics **improves**. This is especially true for the first 10 or so iterations, where we observe the most effective updates. More iteration may **overfit** a model. NeuMF's training loss is the lowest, then MLP and lastly GMF. The metrics follow the same trend, with the best ones are from NeuMF, then MLP and lastly GMF. These findings prove the **effectiveness** of viewing NCF as a probabilistic model and using the log loss to optimizing it for learning from implicit data.

8.3.1 MLP with different layers table

To investigate whether learning user-item interaction function using a deep networks architecture is **beneficial** to the recommendation task, we experimented using MLP with different number of hidden layers. Table 2 and 3 show the results we obtained compared to the original paper. The notation MLP-x indicates that MLP has x hidden layers, beside the embedding layer. As the original paper observes, stacking more layers is **beneficial** to the overall performance, even for models with the same capability. This improvement is caused by the **high non-linearities** brought by stacking more non-linear layers. MLP-0 has no hidden layers; simply concatenating user and item latent vectors is **insufficient** for modelling the interaction function as we can see from the results. Obviously, using more hidden layer make the training process much slower due to the bigger model.

Factors	MLP-0	MLP-1	MLP-2	MLP-3	MLP-4
MovieLens					
8	0.469 (0.452)	0.596 (0.628)	0.676 (0.655)	0.691 (0.671)	**0.704** (**0.678**)
16	0.469 (0.454)	0.663 (0.663)	0.689 (0.674)	0.698 (0.684)	**0.708** (**0.690**)
32	0.469 (0.453)	0.692 (0.682)	0.701 (0.687)	0.706 (0.692)	**0.711** (**0.699**)
64	0.469 (0.453)	0.701 (0.687)	0.703 (0.696)	0.712 (0.702)	**0.715** (**0.707**)
Pinterest					
8	0.275 (0.275)	0.848 (0.848)	0.851 (0.855)	0.859 (0.859)	**0.862** (**0.862**)
16	0.274 (0.274)	0.850 (0.855)	0.858 (0.861)	0.862 (0.865)	**0.864** (**0.867**)
32	0274 (0.273)	0.859 (0.861)	0.861 (0.863)	**0.863** (**0.868**)	**0.863** (0.867)
64	0.274 (0.274)	0.863 (0.864)	0.864 (0.867)	0.864 (0.869)	**0.866** (**0.873**)

Table 2
Hit Ratio top-10 of MLP with different hidden layers. Values in round brackets are from the original paper

Factors	MLP-0	MLP-1	MLP-2	MLP-3	MLP-4
MovieLens					
8	0.261 (0.253)	0.340 (0.359)	0.401 (0.383)	0.416 (0.399)	**0.429** (**0.406**)
16	0.261 (0.252)	0.391 (0.391)	0.422 (0.402)	0.423 (0.410)	**0.436** (**0.415**)
32	0.260 (0.252)	0.413 (0.406)	0.434 (0.410)	0.437 (**0.425**)	**0.442** (0.423)
64	0.261 (0.251)	0.432 (0.409)	0.437 (0.417)	0.440 (0.426)	**0.446** (**0.432**)
Pinterest					
8	0.129 (0.141)	0.519 (0.526)	0.527 (0.534)	0.528 (0.536)	**0.529** (**0.539**)
16	0.130 (0.141)	0.524 (0.532)	0.533 (0.536)	0.535 (0.538)	**0.537** (**0.544**)
32	0.129 (0.142)	0.530 (0.537)	0.536 (0.538)	0.537 (0.542)	**0.539** (**0.546**)
64	0.129 (0.141)	0.530 (0.538)	0.538 (0.542)	0.538 (0.545)	**0.542** (**0.550**)

Table 3
NDCG top-10 of MLP with different hidden layers. Values in round brackets are from the original paper

8.3.2 Pre-Train performance

Previously, we had already emphasized the importance and utility of **pre-training** for NeuMF. We will now compare our two versions of NeuMF and the ones reported in the original paper. Like the original authors, we used **Adam** to learn from scratch NeuMF without pre-training, randomly initializing the model's parameters. The fine-tuning process was not explained in great detail in the original work, so we followed the standard procedure for doing it, like explained in section 4.3.1.

The results and comparison between our work and the original paper regarding the pre-training is shown in table 4. Our results turned out to be in line with the original paper, on MovieLens they are even **better**, while on Pinterest they are slightly worse. The reason of the declining in performance can be found in the big dimension of the Pinterest dataset and

	With Pre-training		Without Pre-training	
Factors	Hit Ratio	NDCG	Hit Ratio	NDCG
MovieLens				
8	0.685 (0.684)	0.409 (0.403)	**0.704** (**0.688**)	**0.425** (**0.410**)
16	**0.712** (**0.707**)	**0.441** (**0.426**)	0.708 (0.696)	0.440 (0.420)
32	**0.733** (**0.726**)	**0.455** (**0.445**)	0.720 (0.701)	0.449 (0.425)
64	**0.746** (**0.730**)	**0.468** (**0.447**)	0.728 (0.705)	0.450 (0.426)
Pinterest				
8	**0.872** (**0.878**)	**0.545** (**0.555**)	0.867 (0.869)	0.543 (0.546)
16	**0.876** (**0.880**)	**0.556** (**0.558**)	0.868 (0.871)	0.547 (0.547)
32	**0.878** (**0.879**)	**0.561** (**0.555**)	0.868 (0.870)	0.548 (0.549)
64	**0.871** (**0.877**)	**0.558** (**0.552**)	0.869 (0.872)	0.548 (0.551)

Table 4
Performance of NeuMF with and without pre-training

our available working machines, as discussed in a earlier section. The improvements in the MovieLens, instead, may be originated by how we conducted the pre-processing of the dataset.

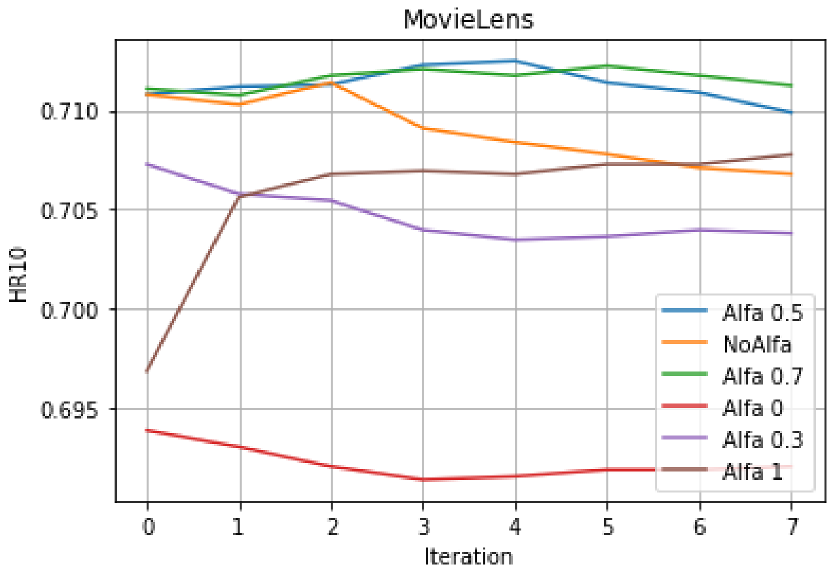

Fig. 13. Hit Ratio top-10 NeuMF with different trade-off values (Factor = 16). Values in round brackets are from the original paper

Depending on the dataset we work with, we need to **tune** the values of the trade-off parameter in a way that allows to better learn from the available data. Differently from what has been reported in the original paper, we tried **different values** of the trade-off parameter in the initialization of NeuMF with pre-training to see its effect on the performance. Figure **??** shows the trend of the fine-tuning over few iterations with different trade-offs on the MovieLens dataset. The value of *alpha* refers to the weight given to the the linear part of the model, while *1-alpha* refers to the weight of its non-linear part. Looking at the performance in the plot with only the initialization of the weights (iteration zero), we can see that it's important to have alpha **set to 0.5 or higher**. This makes us understand that, on MovieLens, the **linear capabilities** of GMF contained in NeuMF are important to obtain good results.

After performing the fine-tuning, the configuration that gives the best performance is to set alpha to **0.5**, giving the same weight to the linear and to the non-linear parts of NeuMF, confirming the original authors choice as the **best choice**. Setting alpha to 0.7, giving a little more importance to the GMF compared to MLP, results in **good performance** even if slightly worse that 0.5. We also tried to not use the trade-off parameter (no alpha) and we saw that the initialization performance was the same than setting alpha to 0.5 but the fine-tuning process is unable to improve the performance like for the latter value. The other values of alpha made us understand that giving less importance to the linear parts of the model compared to the non-linear parts or using only one part of it, makes the performance **fall drastically**, especially if we give set the trade-off parameter (alpha 0) all in favor to the non linear part.

9 Conclusion

In this project, we reproduced the work done by the authors in the original paper building a neural network architecture for collaborative filtering. We implemented with **Keras** the three instances of the **NCF framework** proposed in the original article -**GMF**, **MLP** and **NeuMF**- and we compared the results of the experiments. In particular, we saw how it is possible, with NeuMF, to **unify** the strength of linearity of MF and non-linearity of MLP to achieve **more flexibility** and **performance** in learning from data and modelling the user-item latent structures. We also saw how tuning some parameters can affect the overall performance of the network. The performance of the experiments we carried out on all the models are similar to the ones reported by the authors, sometimes we even **exceed them**.