Collaborative Filtering, Report

Homework 2: Collaborative Filtering

Contact info

GitHub link: https://github.com/kirilldaniakin/AML-DS-2021

DockerHub link: https://hub.docker.com/u/kirlon

Kirill Daniakin: k.daniakin@innopolis.university

Motivation

Motivation behind this project is to compare matrix factorization and neural network approaches to collaborative filtering on the example of the task of predicting users' ratings for movies.

Task definition and data description

Dataset is MovieLens 20M Dataset truncated down to the first million ratings and with filtered out users and movies rated less than 20 times. Eventually, there is 6687 users and 5064 movies. The dataset is presented as two .csv files (one for train, one for test) with three columns each: "userId", "movieId", and "rating". The task is to create a model able to predict user's ratings for movies that were not rated by that particular user, with the help of collaborative filtering.

Data Preprocessing

The dataset contains user and movie Ids, but they are not consecutive numbers, meaning that, for example, if we sort movie Ids or user Ids by decreasing order we get the following consecutive cells:

userId	movieId	rating
5763	118696	4.5
6646	118696	2.5
768	116823	3

By counting number of unique user and movie Ids the two following numbers are achieved correspondingly: 6687 users and 5064 movies, but maximum Id for movies is 118696 and for users – 6743. This confirms that there are missing Ids in the dataset.

The idea for matrix factorization is to factorize rating matrix (with user Ids being rows, and movie Ids being columns, ratings – values) to two matrices: user-feature matrix and feature-movie matrix. However, if missing Ids were accounted for, rating matrix would have size of 6743x118696, but in fact, the dataset contains only 6687x5064 matrix. Therefore, mapping is needed to bring the data to ordinal scale.

After applying such mapping (it is reversible, as map is stored in variable, so after training it is possible to reverse and get original Ids), the original data becomes ready to be presented as rating matrix R with the help of pandas "pivot" function, NaNs are filled with 0.

After getting rating matrix, the mask M is calculated by rule: if rating matrix in certain position is more than zero, then replace that value with 1.

In order to load data in batches, custom data loader is defined in separate script file called Loader.py.

Please note that described data preprocessing steps are conducted using pandas and take some time to complete. Workflow execution on github may take a while.

Justification for design decisions

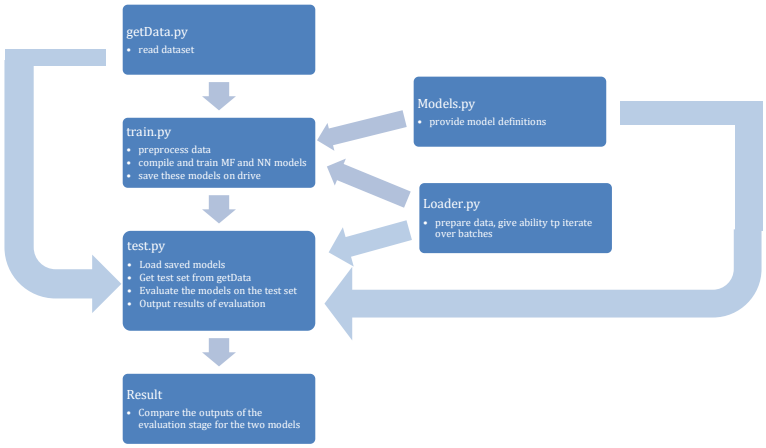
As platform for implementation for both matrix factorization and neural network approach, pytorch was used.
For the first approach, the idea was to use the similarity between pytorch and numpy, but to take advantage of pytorch's ability of being sent to GPU.
For the second approach, as high level APIs are prohibited, the choice was between pytorch and tensorflow low-level APIs. As first part (matrix factorization) was to be implemented in pytorch, it was decided to continue with pytorch for the second part.

For the sake of simplicity and time efficiency, complicated preprocessing steps (such as sparse matrices, which would require rebuilding loss function and gradient calculations), were left out of the scope of this project. Also for simplicity, number of features for factorization is set to 10.

Note: for training and evaluation of matrix factorization part, pytorch Ignite Engine is used in order to avoid writing long "for" loops, but the training and evaluation steps are specified manually, therefore, use of this tool cannot be considered as use of high level API.

This project workflow

First, the dataset is read by getData.py script. Then it is preprocessed in either train or test script with accordance to their needs. Models script defines models, it is necessary that train and test share the same model architectures, otherwise, it would be impossible to load models with pytorch. Loader script is used for dividing data to batches and providing iterator. After that, train script trains the two models and saves them. Test script reads saved models and evaluates them on the test set. The results of evaluation are printed.



Comparison of Matrix Factorization and Neural Network approaches

Matrix Factorization

1. This model is built as a python class inherited from pytorch nn.Module;

2. It has two parameters as tensors: user-feature $P_{n_{user} \times k}$ and feature-item $Q_{k \times n_{item}}$ matrices;

3. During the training process weights of those two parameters are adjusted to converge to minima according to Alternating Least Squares algorithm:

$$P = P - \frac{\alpha}{n_{user} \cdot n_{item}} \left(\lambda P - \left((PQ^T - R) * M\right)Q\right),$$

$$Q = Q - \frac{\alpha}{n_{user} \cdot n_{item}} \left(\lambda Q - \left((PQ^T - R) * M \right)^T P \right),$$

where α – learning rate;

λ – regularization constant;

$R_{n_{user} \times n_{item}}$ – rating matrix;

$M_{n_{user} \times n_{item}}$ – mask, where 1 correspond to existing in the train data pair userId-movieId, 0 – to not existent (user didn't rate this particular movie). Mask is also used in loss calculation – particular loss value for pair userId-movieId is accounted for only if the corresponding value in R matrix is not zero.

* - element-wise multiplication of matrices;

n_{user} – number of users;

n_{item} – number of items (movies);

$n_{user} \cdot n_{item}$ – number of ratings.

The loss function is MSE with regularization L2 by P and Q.

Note: as MSE is used for loss, here gradients are divided by number of ratings.

4. Two pytorch SGD optimizers are used to update each of the matrices after manual gradient calculation as described above. Optimizers only serve to update weights automatically by optimizer.step() function. This is done to add elegance to the code, shorten and optimize it.

Note: Automatic weights update was not prohibited by the task unlike automatic gradient calculation.

5. $Learning\ rate\ \alpha = 0.01;\ k = 10;\ \lambda = 10^{-6}$.

NN architecture

1. Two Embeddings for users and movies;

2. One `Dropout` for the output of the embeddings;

3. The hidden layers:

- 3 layers (128, 256 and 128 neurons),
- Dropout after every layer with 20% probability,
- Relu as activation function for all 3 hidden layers;

4. Output layer.

Optmizer is Adam, learning rate = 0.001, scheduler of learning rate on plateau is also used.

The summary is presented in Table 1.

Table 1 NN model summary

```
RecommenderNet(
  (u): Embedding(610, 10)
  (m): Embedding(193609, 10)
  (drop): Dropout(p=0.02, inplace=False)
  (hidden): Sequential(
    (0): Linear(in_features=20, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=128, out_features=256, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=256, out_features=128, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.2, inplace=False)
  )
  (fc): Linear(in_features=128, out_features=1, bias=True)
)
```

Comparison

After model training, resulting trained models are saved with the help of pytorch. This is where the training process stops, and testing starts. First, saved models are read from drive by test script, then they are evaluated on the test dataset. Metric scores gathered during train and evaluation stage are reflected in Table 2.

Table 2 Resulting model comparison

	MF	NN
Train set last loss	0.12	0.3597085475921631
Test set loss	0.1177297979593277	0.7910948991775513

Conclusion and future work

In this project, neural network and matrix factorization models were compared on the task of predicting user ratings. NN does not perform much better in terms of loss, but it is faster than matrix factorization approach due to matrix (tensor) multiplication. From the other hand, matrix factorization showed better results taking less epochs for achieving minimal loss in training than NN. For future work, it is worth it to consider different NN architectures and using sparse matrices for matrix factorization.

Bonus

As a bonus, there was implemented a method taking user Id as input and giving top five recommended movie Ids as output. The method is based on matrix factorization model.
First, during the training of matrix factorization model, the following data was saved as files:

- model parameters, namely user-feature and feature-movie matrices;
- the original rating matrix R;
- mappings described in data preprocessing steps.

Algorithm is the following:

- The method reads these files into corresponding models/variables (via pytorch and numpy);
- user Id is taken as input from std;
- mapping is applied to user Id (see “Data preprocessing” section);
- row from user-feature matrix P is selected corresponding to the supplied user Id, this row is multiplied with the whole feature-movie matrix Q;
- the resulting vector (tensor) is sorted, original indices are kept thanks to torch.sort();
- movie Ids which were already rated by the user are filtered out (here is where matrix R is used);
- top five of the resulting vector are printed.

The docker image was already built and pushed to DockerHub, all user needs to do is pull and run it with the following commands:

```
docker pull kirlon/cf:recommender
docker run -it kirlon/cf:recommender
```

Then, the method will ask user to input user Id and will output five recommended movies, but please keep in mind, that if user Id was not in the original train dataset, the method won’t work, giving only “No such user!” message.

In case user wants to rebuild the image, they can find Dockerfile, source code (recommender.py) and saved data (R.pt, map_u.npy, map_m.npy, mf.pt) in GitHub repository.

Please keep in mind, that the image pulled from DockerHub is somewhat heavy due to pytorch usage and stored data, therefore, it make take some time to pull it.

About repository

There are separate files for training and testing the models in the GitHub repository, namely train.py and test.py. In the same “src” folder, Models.py defines models, loader.py is used for dividing data to batches and providing iterator.

Script getData.py, which is located in scripts folder, is used for data loading. It returns two datasets (train and test).

Training script train.py saves the two models: MF and NN.

Testing script test.py loads these models and evaluates them on the test data.

In order to run train and test procedures in repository, one would need to go to “actions” tab of GitHub from the repository page and run “NN Model Test” workflow, which will install all dependencies, run train.py script, which internally uses getData.py and loader.py scripts (as well as Models.py), then, after training, run test.py script, outputting the loss on the test set of the 2 models: MF and NN.

It is also worth mentioning that in order to save time, MF models trains for less epochs in the GitHub repository, as the model achieves loss minimum early, and doesn’t improve significantly (after third sign) afterwards.