Comparing Collaborative Filtering and Neural Network in Movie Recommender System

Anonymous Submission

1 Introduction

Recommender systems are widely used in our daily life. Whenever we open a shopping website like Amazon, or a video website like YouTube, we would see a bunch of items in the front page. Those items are generated by a recommender system, which infers our preferences based on our activities, such as browsing history, ratings and purchases. A good recommender system would suggest good contents that we would like to see, and thus will make our life more convenient.

This project report will briefly introduce how to build a recommender system for movies. Section 2 presents how the problem is formulated. Section 3 elaborates on two methods, namely collaborative filtering and neural networks. Section 4 shows the results from experiments. Finally, Section 5 draws a conclusion and summarizes the whole report.

2 Dataset and Problem Description

According to the textbook [5], a machine learning problem consists of three parts: data, model and loss. Now we will discuss the data part here and leave model and loss to Section 3. This methods in this report will be built with the Movie Lens Dataset [7] by Grouplens Research. This dataset contains 100835 ratings of 9724 movies from 610 people, and the ratings range from 0 to 5 with a step at 0.5. The following table shows the first three rows of the entire dataset.

Table 1: Entries in Movie Lens Dataset

	User ID	Movie ID	Rating
1	1	3	4.0
2	1	6	4.0
3	1	47	5.0
⋮	⋮	⋮	⋮

In this problem, a **data point** is a row in the table, which is a tuple with elements User ID, Movie ID and Rating. User ID and Movie ID serve as **features**, and Rating will be considered as **label**. Specifically in this dataset, User ID is an integer ranging from 1 to 610 without, but Movie ID ranges from 1 to 193609 and is not continuous. Concretely, the task of a recommender system is to predict a user's rating (unknown) of a movie, given a set of records (User ID, Movie ID and Rating). And we will see how to build such a system in the following section.

3 Methods

In this section, we discuss two methods for a recommender system. The first one is called Collaborative Filtering, which was introduced in Round 3 by Prof. Stephan Sigg [8]. The second is a feed forward neural network, which is implemented with PyTorch. Notably, this network contains an embedding layer [6] and the network structure is inspired by Lecture 4 in CS-E4890 Deep Learning [4], as well as the corresponding assignment.

In the Collaborative Filtering model, the estimated rating $\hat{r}_{ui}$ of movie i from user u is a biased inner product of the item (movie) vector q_i and user vector p_u:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u, \tag{1}$$

where μ denotes the average rating of all movies, and b_u and b_i stands for user and item biases. Meanwhile, the vector representations of users and items as well as their biases can be learned jointly, by minimizing a **regularized mean square error** function:

$$\sum_{r_{ui} \in Trainset} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2 \right), \tag{2}$$

where r_{ui} means the true rating of movie i given by user u. As stated in the lecture [8], it's hard to optimize q_i and p_u simultaneously, but we can optimize them alternatively using gradient descent. Notably, this report implements collaborative filtering with the python package `Surprise` [3].

The embedding neural network is implemented with the following codes. The two embedding layers map the original user ID and item ID to vectors with length `n_embed`, and then these embedded features will be fed to the `hidden_block`, which uses the `Tanh` function for activation. Finally, the hidden features will be processed by a linear output layer. Similar to collaborative filtering, we use a **mean square error** for this neural network, which is just the first term in Equation (2). The loss function will be implemented with `torch.nn.MSELoss`, and we apply `Dropout` [2] for regularization.

```python
import torch
import torch.nn as nn
class RecommenderSystem(nn.Module):
    def __init__(self, n_users, n_items, n_embed=40,
                dropout_rate=0.2, hidden_neurons=[10]):
        super(RecommenderSystem, self).__init__()
        self.user_emb = nn.Embedding(n_users, n_embed)
        self.item_emb = nn.Embedding(n_items, n_embed)
        hidden_block = []
        last_output = n_embed*2
        for n_hidden in hidden_neurons:
            hidden_block.append(nn.Linear(last_output, n_hidden))
            hidden_block.append(nn.Tanh())
            hidden_block.append(nn.Dropout(p=dropout_rate))
            last_output = n_hidden
        self.hidden_block = nn.Sequential(*hidden_block)
        del hidden_block
        self.output_layer = nn.Linear(last_output, 1)

    def forward(self, user_ids, item_ids):
        user_feat = self.user_emb(user_ids-1)
        item_feat = self.item_emb(item_ids-1)
        x = torch.cat([user_feat, item_feat], dim=1)
        x = self.hidden_block(x)
        x = self.output_layer(x)
        return torch.squeeze(x)
```

There are two reason for using (regularized) MSE Loss for both of the methods. First, MSE Loss is a continuous function, and therefore easy to apply gradient based optimization. Second, MSE Loss penalize high deviation heavily, this allows us to obtain results that are close to true value.

4 Experiments and Results

Before any of the following experiments, we need to reserve a part of data for the final test, and use the other data for training and validation. Specifically, the size of training, validation and testing data are 60%, 20% and 20%. To do this, we need to load the original data table, shuffle the data and then save 20% to a file for final testing. After that, we save the rest 80% and proceed with "training-testing" scheme.

4.1 Collaborative Filter

To construct a suitable dataset for the algorithm, we need to use the built in classes `Dataset` and `Reader` in the `Surprise` package.

```
# timestamps are recorded but we don't need them
cols = ['user_ids', 'item_ids', 'ratings', 'timestamps']
data = pd.read_csv(data_dir + "/" +"ratings_train_valid.csv", names=cols, header=1)
dataset = Dataset.load_from_df(data[['user_ids', 'item_ids', 'ratings']], Reader())
```

After that, the implementation of Collaborative Filter can be done in a few lines of codes with the `SVD` method [1] . The following code block briefly shows the key computations done for the evaluation, the related `import` statements are omitted here.

```
def compute_mse(pred, label):
    pred, ref = np.array(pred), np.array(label)
    return np.mean((pred-ref)**2)

def pred_results(algo, testset):
    output = algo.test(testset)
    preds = [x.est for x in output]
    labels = [x.r_ui for x in output]
    return preds, labels
# test_size is 0.25 because we need 20% data for validation, but the file
# contains 80% of the whole data, 0.8*0.25 = 0.2
trainset, validset = train_test_split(dataset, test_size=0.25, shuffle=True)
train_preds, train_labels = pred_results(svd.fit(trainset), trainset.build_testset())
print("MSE on Train Set {}".format(compute_mse(train_preds, train_labels)))
valid_preds, valid_labels = pred_results(svd, validset)
print("MSE on Validation Set {}".format(compute_mse(valid_preds, valid_labels)))
```

The training set and validation set are obtained by a single random split, where the validation set takes 20% of the whole dataset. This split is done with the `train_test_split` function in `Surpirse` package, and it works like the split function in `sklearn`. Here, the training and validation process could be simply done by `fit` and `test`. After some hyperparameter tuning, the hyperparameter set that gives the lowest MSE error on `testset` has been adopted to produce the final result. In this case, the MSE loss on `trainset` is 0.64, and that for `testset` is 0.77. Since the training loss is not significantly lower than the test loss, it's reasonable to say the model has not over-fit too much.

4.2 Embedding Net

The Embedding Net `RecommenderSystem` is implemented with PyTorch, and the optimizer for the training process is the `Adam` optimizer in `torch.optim`. Since a single data point is small, we can feed a large batch each time and choose a larger learning rate. For this report, the batch size is chosen at 1024 and learning rate 0.01. The whole training process consists of 30 epochs, and in each epoch, we record the training loss and validation loss, which are presented in the figure below.

While the implementation of collaborative filter needs just a few lines of codes, training a neural net actually requires more work. The complete code cannot be displayed because of limited report length, and the main missing part is the dataset. For that, the general idea is to compose a `TensorDataset` in `PyTorch` with the data, after loading and shuffling data.

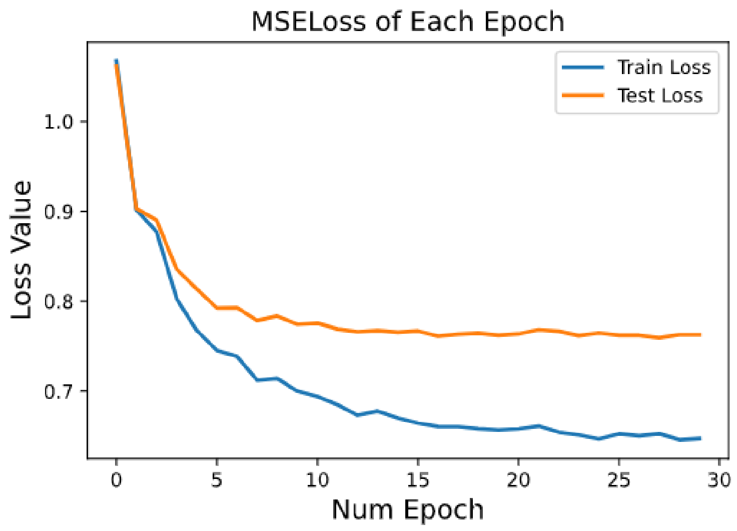

Figure 1: Add captions here

From the figure above, we can see the training loss and validation loss (noted as Test Loss in the figure) decreases as the training process goes on. Concretely, this neural network achieves a best validation loss at 0.76, which is slightly lower than the collaborative filter implementation. Therefore, for this movie recommender system, we would (slightly) prefer the Embedding net solution. Further, we test the trained neural net with the initially reserved test set, and final test error turns out to be 0.77, which is almost the same as the validation loss.

5 Conclusions

In this project, we've seen how to build a recommender system by collaborative filtering and embedding neural network. According to the experiment results, the validation loss of collaborative filter is 0.77, compared to 0.76 of the embedding net. Although we prefer the neural net solution, we can not yet certainly conclude it is better than the filtering solution. The validation loss and final testing loss of the neural net are fairly close, and both of them are approximately 0.1 higher than training loss. The neural net is prone to over-fit, and therefore a good direction for future work is to test these two kinds of solutions on larger dataset.

References

[1] Simon Funk. Svd algorithm. https://sifter.org/~simon/journal/20061211.html. Accessed: 24 Mar. 2021.

[2] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[3] Nicolas Hug. Scikit-learn surprise package. https://github.com/NicolasHug/Surprise. Accessed: 24 Mar. 2021.

[4] Alexander Ilin. Cs-e4890 deep learning lecture 4. https://mycourses.aalto.fi/pluginfile.php/1436564/mod_resource/content/4/reg_all_slides.pdf. Accessed: 24 Mar. 2021.

[5] Alexander Jung. Machine learning basics book. https://github.com/alexjungaalto/MachineLearningTheBasics/blob/master/MLBasicsBook.pdf. Accessed: 24 Mar. 2021.

[6] PyTorch. Embedding layer. https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html. Accessed: 24 Mar. 2021.

[7] GroupLens Research. Movie rating dataset. https://grouplens.org/datasets/movielens/latest/. Accessed: 24 Mar. 2021.

[8] Stephan Sigg. Lecture slides of cs-c3240 machine learning. https://mycourses.aalto.fi/pluginfile.php/1415791/mod_page/content/23/CS-C3240_Round3_AnomalyDetection-OnlineLeaning-RecommenderSystems.pdf. Accessed: 24 Mar. 2021.