ICLR Paper Reproducibility Report

Lei He, Zhixiong Hu, Congwei Wu

Decemeber 15, 2017

1 Introduction

This is a report for reviewing the reproducibility of a recently proposed recommender system algorithm in ICLR paper called “Training Deep AutoEncoders For Recommender System”. Due to the double blind peer review process, the names of autors are anonymous. It was submitted to 2018 International Conference on Learning Representation(ICLR) on November 3, 2017. The main idea of the paper is that the authors proposed a new model based on autoencoders for the rate prediction task in the recommender systems and claimed that their model is able to outperform significantly previous state-of-art models. In addition, they also proposed a new training algorithm to speed up training and improve their model performance. Our goal is to assess if the new methods are reproducible, and to determine if the conclusions of the paper are supported by our findings.

2 Summary

2.1 Methodology

An autoencoder is a type of neural network usually used for unsupervised learning task. The aim of autoencoder is to learn a representation for a set of data(encoding) and then reconstruct them(decoding). In this paper, the authors trained a deep autoencoder for the rating prediction on the Netflix data set, which is a supervised task. Specifically, the model contains 6 layers: one input layer, two encoding layers, two decoding layers and one output layer. Both encoders and decoders consist of feed-forward neural networks. The Netflix data set was first cleaned and made as a large sparse rating matrix where each row represents each individual users and columns are the movies. In most of the experiments, SELU was used as the activation function but the last layer is linear if the range of the activation function is smaller than that of data; They also use batch size of 128, with momentum optimizer of 0.9. The learning rate is 0.001. The xvaier initialization was chosen to initialize the weights. The loss function optimized is the square root of Masked Mean Square Error Loss(RMSE), which is computing Mean Square Error(MSE) over all the predicted values, except for those elements whose corresponding true values equal to the masked value(in this case the masked value is 1 for non-zero rating in the true value).

To tackle the issue that the rating for all items of a user is usually sparse but the model output is dense, the paper proposed a technique which augments every optimization iteration with an iterative dense re-feeding steps to enforce fixed-point constraint and to be able to perform dense training updates.

To avoid model over-fitting, the authors tested different numbers of hidden units, different numbers of layers and various dropout values. Their experiments indicated that small number of hidden units(such as 128), 3 or less layers in both encoders and decoders and very high value of dropout(such as 0.8) turned to improve the model the most.

2.2 Implementation Details

The recommender system built in this article mainly includes four parts–preprocessing the data, constructing the autoencoder structure, training the model and evaluating the accuracy. In order to implement the whole system, we checked the correction of each part and re-ran them step by step. Notice that due to the facility and time limitations, the implementation only focused on 3-month Netflix data set instead of the full data set.

The data preprocessing is quite important because model training requires standard data form. The raw Netflix data set consists of three features: user ID, movie ID and rating. Each row represents an user's rating to a movie, which was stored as the string type. In order to transform the raw data into a numeric tensor which is a standard input form for autoencoder in pytorch, each row was separated into a user ID index, a movie ID index and the corresponding rating value. Since one user is able to rate more than one movie and a movie can be viewed by several users, a major ID should be defined to reconstruct the data. In this case, user ID was set as the major ID. As a result, the data was stored efficiently as a m by n m by n tensor, where m is the number of users, n is the number of movies, the row represented user ID, the column represented movie ID and the element corresponding to i-th row j-th column was the rating of i-th user to j-th movie. Finally, for every epoch, both the training and validation data were evenly divided into 128 batches.

The model implementation was done using the Pytorch framework. It consists of three pieces: the activation function object, the MSE loss function and the definition of an autoencoders class object. The structure of the model implementation is similar to most neural network implements with Pytorch. There is another script to perform the training task. In there it reads all the parameters such as layer size, dropout value, batch size, etc. During the training, the author wrote the codes in the GPU environment and some codes to generate the plots and logs with Tensorflow. In our implementation, due to some limitations we rewrote some parts of the codes to make it run in the CPU environment only and did not use Tensorflow to generate the plots.

While training this autoencoder, the most influential part is choosing the learning rate because it has huge impact on the model result. If the learning rate is too large, the training will stop early so that only NaN appears in the following loss (perhaps this case is due to the property of Pytorch). As a result, the optimizer will update the model with NaN, making the training fail. On the other hand, if the learning rate is too small, it will take too long to train precisely within limited epoch size. Unfortunately, no clue about what learning rate should be used appears in the article. In the implementation, learning rate 1e-4,2e-4 and 5e-4 were selected through practice. It is also worthy to mention that our implementation used the same "momentum" optimizer as the author's paper.

Three aspects of author's result were tested in our implementation. The performances of eight popular activation types were compared to prove that SELU activation type is the most efficient one in this autoencoder. Then based on SELU, we also compared the performance of different drop probability, showing that the model could over-fits without regularization. Finally, based on author's model architecture, the 3-month test RMSE was re-calculated, which turned out to be similar to the author's result.

3 Analysis

Availability of datasets, partitioning information
The training/test dataset is provided by Netflix. We followed the data cleaning code provided by the author, to split data evenly according to 3/6/12 months and follow-up one month for testing.
Availability of code, names and version numbers of dependencies
The code is available on Github, created by the author. All code can be ran in Python 3.
Availability of random seed and all hyperparameters
Random seed and hyperparameters were in the data cleaning scripts provided by the author, which will produce the exactly same dataset of 3/6/12 months.
Alignment between the paper and the code
All the methods and algorithms mentioned in the paper were basically implemented in authors' codes. There are four major components: `run.py`, `infer.py`, `input_layer.py` and `model.py`. The `model.py` is the neutral network defined in the paper and `input_layer.py` defines how we construct the user-item mapping relationships. Other two modules were used to obtain readable reports.
Clarity of code and paper (ease of understanding, language)
All code was written in Python. There is a clear one-to-one corresponding between each module and one part of the paper, and it is also easy to modify the code to have a different implementation for other purposes.
Details of computing infrastructure used
The difference in training framework between us and the author is Pytorch. We used Pytorch Linux conda Python 3.5 without CUDA. The author used unknown platform Pytorch Python 3.6 GPU with CUDA (8.0). We deployed a Jupyter notebook on three virtual instances hosted Google Cloud Computation (8 vCPUs, 8 GB memory) with CPU platform, Intel Sandy Bridge.
Computation requirements (time, memory, number/type of machines)
Jupyter provided a shared development environment for all members in our team. We didn't have GPU installed because there wasn't enough expense and we experienced installation issues on CUDA driver.
Reimplementation effort
It is effortless to refactor and build all our code for a reproduction purpose based on originals.
Number and complexity of interactions with the authors
To have a concise software architecture, we put everything in a single Jupyter Notebook combining every code. This approach is proven to be consistent with the author.

4 Discussion of Findings

As mentioned above, we only used 3-month Netflix data to verify the results and conclusion made by the authors. The layer size we used is the same as which the author suggested for 3-month training data(m is the number of rows in training data set): encoder two layers m x 128 and 128 x 256, decoder two layers 256 x 128, 128 x m. Batch size is 128. One issue we found is that the paper mentioned that they used learning rate 0.001 for most of the experiments, but in our experiments we found that this learning rate might be too large for 3-month data because during the training process, we had training RMSE being 'NaN' after a few epochs and then the training process failed. One possible reason is that 0.001 was used for the full Netflix data, but not a good choice for 3-month data. Therefore we had to investigate and verify the range of appropriate learning rate for the 3-month data. Below table 1 shows the results of some of our investigation. Considering the running time, we only used 40 epochs to train the model and get the RMSE.

Table 1: RMSE Comparison

Learning Rate	Dataset	RMSE
0.001	Train	NaN
	Validation	NaN
	Test	NaN
0.0002	Train	0.967
	Validation	0.976
	Test	0.978
0.0001	Train	0.995
	Validation	0.995
	Test	0.997
0.00005	Train	1.027
	Validation	1.027
	Test	1.029

According to table 1, if the learning rate used for 3-month data is 0.001, which is consistent with author's choice for most of their experiments, then the training process would fail and yield 'NaN' for the RMSE. If we decreased it to 0.0002, then it helped finish the training and the RMSE results are close to the results in the paper. However, we further decreased it to 0.0001 and 0.00005, the RMSE would start to increase. So as the learning rate becomes smaller, the epoch number should increase accordingly. Therefore for 40 epochs, a good range of the learning rate for 3-month data should be 0.0002 or slightly larger.

Just like what the author did, we also implemented the comparison between activation functions and verified the fix of overfitting by adding drop probability to autoencoder. Figure 1 and figure 2 below show the results.

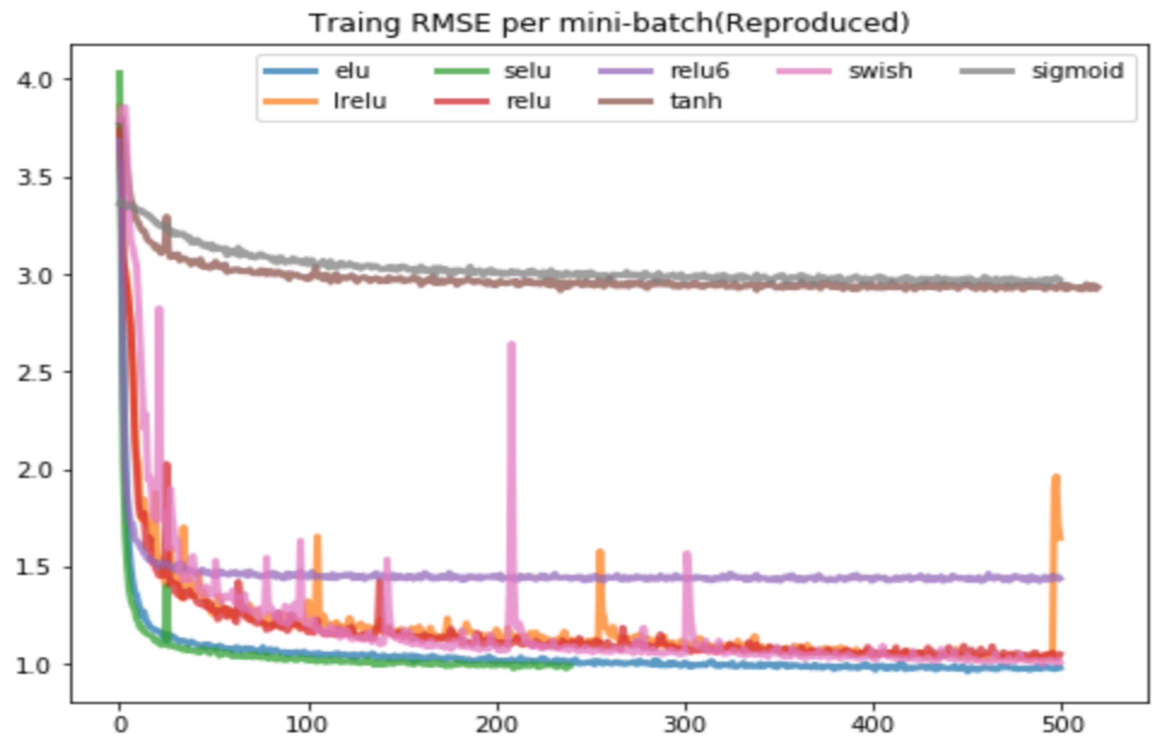

Figure 1: Training RMSE per mini-batch. This is a reproduction on Figure 2 of paper. The difference is that we used 3-month data and the author used full data. We achieved similar results in terms of RMSE across different activation functions: TANH and SIGMOID lines are very similar as well as lines for ELU and SELU. The best performing activation functions are ELU and SELU

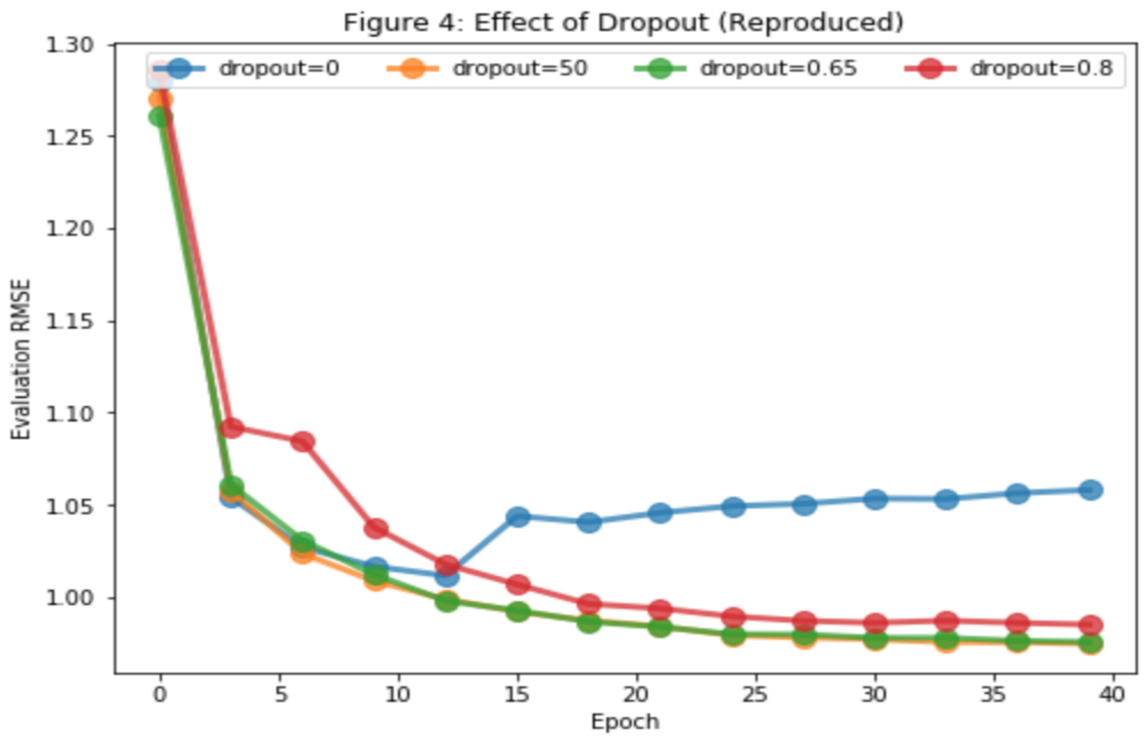

Figure 2: Effect of dropout rate on auto-encoder model. This is reproduction on Figure 4 of paper. The difference is that we used 3-month data and the author used full data. We noticed a similar renounced when dropout rate is zero, compared to other dropout options, which is consistent with the author. The model does overfit if trained without regularization. However, the issue could be fixed by adding high dropout probability.

5 Conclusion

In general, we successfully reproduce the experiments and results. The test RMSE for 3-month data in author's paper is 0.937, while 0.976 in our implementation, which are close. If more epoch is ran, we are quite possible to get closer to 0.94. The patterns of our figures show that SELU activation function is the best and high drop probability could be used to avoid overfitting, which are consistent to the results in author's paper.

There are several limitations in our work. First, we used just 3-month data set. So we can not get exactly the same results as the author. Second, because the author didn't mention how much learning rate should be used, we selected the learning rate ourselves, which may not be the same one the author used.

Appendix

Our code is available at:
https://github.com/y5yeyey/CS273-Project.
Author's code is available at:
https://github.com/NVIDIA/DeepRecommender.
Netflix Data
http://academictorrents.com/details/9b13183dc4d60676b773c9e2cd6de5e5542cee9a.