CS230: Deep Learning-Based Collaborative Filtering

Omar Alhadlaq & Arjun Kunna

22 March 2018

Abstract

In this paper, we use an autoencoder model to predict users' ratings on movies they have not yet watched in the Netflix dataset. After constructing a baseline model consisting of a single-layer autoencoder, we experimented with a) constructing a deeper network b) using constrained autoencoders c) adding dropout and d) dense refeeding. Our final model consisted of 6 layers with [128, 256, 256] units respectively in both the encoder and the decoder, and uses a dropout probability of 0.2. We found that constrained autoencoders and dense refeeding did not improve the model. The model obtained an RMSE of 0.943 on the test dataset, which is comparable to other state-of-the-art deep-learning based recommendation system models.

1 Introduction

Deep learning has shown great success in the fields of computer vision and NLP. However, there are other less-studied areas where deep learning holds much potential. One such area that we are interested in is recommendation systems.

Recommendation systems are of great interest to online services like Amazon, Netflix, and Spotify, as they derive a significant amount of revenue by accurately suggesting products that users might enjoy.

In this project, we aim to improve predictions for user-submitted ratings in recommendation systems, by implementing a deep autoencoder model. The input is a sparse column vector representing a user, with each entry representing the user's ratings for a movie. The output is dense a vector of the same size, with entries representing our predictions for their ratings of the movies. This is detailed in section 3.2.

2 Related Work

The literature shows that non-deep learning methods have faced problems in dealing with matrix sparsity [Zhang et al., 2017]. Moreover, it shows that deep learning has much to contribute to this area, because it is able to effectively capture non-linear and complex user-item relationships. [Li et al., 2017].

Previous deep learning-based efforts include Wu et al. [2017] which has shown promising resutls using RNNs by capturing temporal aspects of the data. In addition, Kuchaiev and Ginsburg [2017] achieves state-of-the-art results using the autoencoder approach, and we are attempting to implement a similar model. We will explain the details of the model, as well as what an autoencoder is, in Section 4.

3 Dataset

We are using the Netflix Dataset [Net]. This is a publicly available dataset with about $480k$ users and $100m$ ratings over 17770 movies.

3.1 Data Exploration

We began by doing an exploration of the data, to get some insight on its structure. The figures referenced in this section are located in the appendix.

First, we plotted a histogram of how many ratings had been submitted per month (Figure 9). This was interesting, as the resulting distribution is skewed with the majority of ratings submitted after 2004. Our explanation for this is that Netflix's popularity increased significantly after 2004.

We also made a distribution of the number of ratings submitted per user. Most users have submitted less than 100 ratings, but a non-negligible number submitted between 100-500 (Figure 10).

Lastly, we plotted a histogram of number of ratings submitted for each movie. As you can see, the majority of movies have between 0-1000 ratings (Figure 11).

3.2 Data Wrangling

3.2.1 From dataset to input matrix

Next, we had to adapt the data's original form into a form that was suitable to train a model on. The dataset was provided as a repository of 17770 sections, one per movie. The first line of each section contained the movie id with each subsequent line in the section corresponds to a rating from a user and its date in the following format: (UserID,Rating,Date).

For our model, we decided to represent each user by a vector, with each vector entry corresponding to a rating for a particular movie. Thus, each vector is in $\mathbb{R}^{17770}$. We wrote some scripts to parse the data files into this form and then wrote it back to disk in a format such that instead of being categorized by movie, it was categorized by user.

Thus, we ended up with a dataset of dimensions $num\_movies \times num\_users = 17,770 \times 480,000$. This is depicted in figure 1.

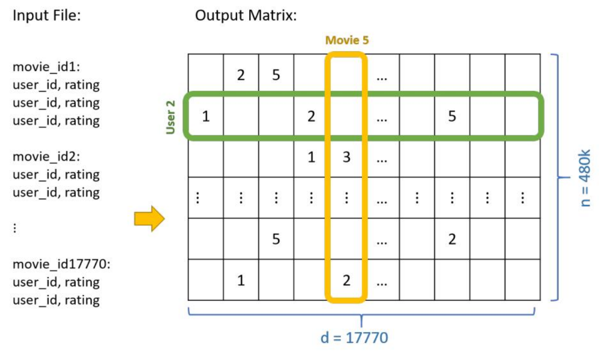

Figure 1: Schematic of data pipeline from input file to output matrix

3.2.2 Training and Test Split

After our data wrangling step, we had 480,000 users each represented as a vector in $\mathbb{R}^{17770}$.

We split the dataset into training and test sets, as shown in Table 1. The training set consists of all the users, with their ratings From Dec 1999 up until Nov 2005. The testing set includes only the users who have recorded ratings in Dec 2005, but includes all of their ratings, dating back to Dec 1999. In short, we train the model using data from Dec 1999 to Nov 2005, and test its predictions on ratings made in Dec 2005 based on previous ratings. We further split the 'testing' set into Test and Dev sets randomly, with almost equal amounts of data points in both.

When running our experiments on different model types, we extracted a mini-dataset of size 50,000 users. Then, we split the mini-dataset in a similar fashion. In the final testing, we ran the model on the entire dataset.

	Full-dataset	Mini-dataset
Training	12/99 - 11/05	12/99 - 11/05
Users	477, 412	49, 698
Development	12/05	12/05
Users	86, 847	9, 056
Testing	12/05	12/05
Users	86, 847	9, 234

Table 1: Training, Development, and Test Split

4 Model

4.1 Architecture

There are two main classifications of recommendation systems:

1. Collaborative Filtering:

   This makes recommendations by learning from historical interactions between the user and items, without considering the nature of the items. The assumption is that two people that have similar tastes, will likely have a similar opinion on a randomly chosen new item.

2. Content Based:

   This uses data inherent in the items, such as frequencies in song data.

We decided against a content-based method because modeling video (or audio) content is difficult due to the structural complexity present in the content. In addition, this method tends to be more domain specific.

4.2 Autoencoders

There are several different types of collaborative filtering methods Many non-deep learning approaches use matrix factorization, which can be thought of as a form of dimensionality reduction. As autoencoders also have a similar property to them, they are a natural method to try for a deep learning model.

An autoencoder is a network that consists of two transformations: $encode(x) : \mathbb{R}^d \rightarrow \mathbb{R}^e$, and $decode(x) : \mathbb{R}^e \rightarrow \mathbb{R}^d$. The goal of the autoencoder is to obtain an e-dimensional representation of the data, such that the error between x and $f(x) = decode(encode(x))$ is minimized.

Both the encoder and decoder parts of our model consist of a fully connected neural networks which has 128 neurons. computing. We first pass the input through this, computing $A = g(W * X + b)$. It then

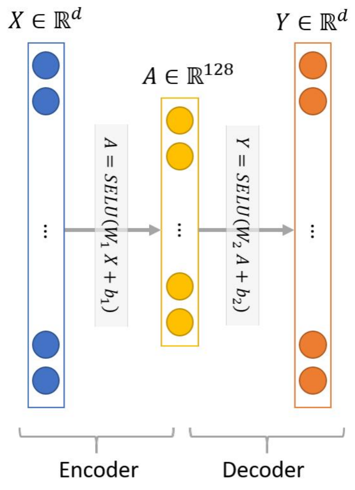

Figure 2: A constrained autoencoder with two neural networks, each with a single layer.

goes through a SELU activation function, and then our decoder function is applied. This is depicted in figure 2.

The forward propagation equations are outlined in (eq. 1). The input vector $X \in \mathbb{R}^{d \times b}$, where d is the number of movies and b is the batch size.

$$\begin{aligned} Z_1 &= WX + b_1 \\ A_1 &= SELU(Z_1) \\ Z_2 &= W^T A_1 + b_2 \\ Y &= SELU(Z_2) \end{aligned} \tag{1}$$

4.3 Loss Function

We used a Masked Mean Squared Error loss function.

$$MMSE = \frac{m_i * (r_i - y_i)^2}{\sum_{i=1}^{i=n} m_i}$$

r_i is the actual rating, y_i is the predicted rating, and m_i is the mask function: $m_i = 1$ if $r_i \neq 0$, and $m_i = 0$ otherwise. Thus, we are only computing the loss on examples where we have the actual rating.

Note that there is a direct relation between $RMSE$ loss and $MMSE$ loss as $RMSE = \sqrt{MMSE}$.

4.4 Activation Function

According to Kuchaiev and Ginsburg [2017], activation functions with nonzero negative part and unbounded positive part work best for autoencoders. Of these, scaled exponential linear units (SeLU) performed better than leaky rectified linear units (LReLU) or exponential linear units (ELU). Thus, we decided to use SeLU as our activation function.

We also used a momentum gradient descent with learning rate 0.005 and a batch size of 32.

5 Experiments and Results

5.1 Baseline Model

We first trained a baseline model, which was a single-layer autoencoder as depicted in section 4. We tried using 128, 256, and 512 units per layer. The training set results and the dev det results are depicted in figures 3 and 4 respectively. Note that although increasing units improves training set performance, we observe that there is the opposite effect on the dev set: increasing the number of units up to 512 units decreases the performance on the dev set significantly as the model starts to overfit. The model with 128 neurons was able to achieve an RMSE of 1.084 on the mini-dev set.

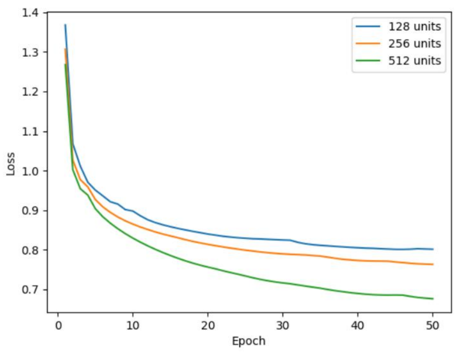

Figure 3: Baseline model performance on the training set.

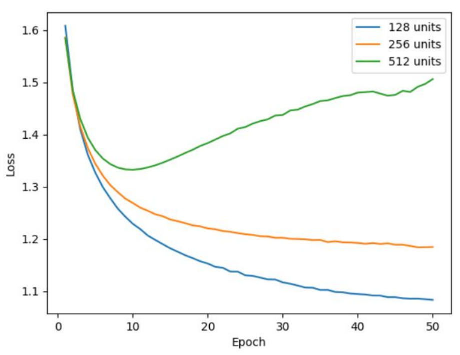

Figure 4: Baseline model performance on the dev set.

After we had the basic model set up, we experimented with some extensions in an attempt to increase its performance.

5.2 Going Deeper

The first thing we did was to increase the number of layers, as this would allow us to train a more complex model. We built a model with 3 layers in each of the encoder and decoder. We tried different structures with different number of units in each layer. In the first, we had $[128, 256, 256]$ units in the first, second, and third layers respectively. We also tried $[256, 256, 512]$ and $[512, 512, 1024]$. Out of these structures, we found $[128, 256, 256]$ to have performed the best as can be seen in figure 5.

Going deeper tremendously improved performance and we were able to decrease the RMSE loss down to 0.951 on the mini-dev set.

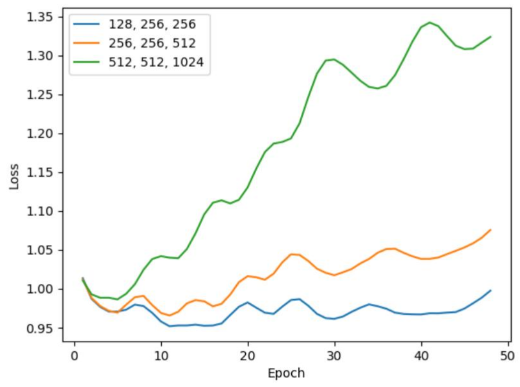

Figure 5: 3-layer model performance on the dev set.

5.3 Constraining the Weight Matrices

After adding more layers, we saw room to improve the model further by reducing overfitting. One way to address overfitting is using a constrained autoencoders.

In the basic model, we trained the weights of the encoder separately from that of the decoder. However, as the decoder is theoretically the inverse of the encoder, it is reasonable to constrain the decoder's weights W^d to be equal the transpose of the encoders weights, W^e. That is, $W^e = (W^d)^T$. This method was mentioned by Kuchaiev and Ginsburg [2017]

This has the effect of effectively halving the number of parameters, and would be expected to reduce overfitting. However, in practice it appeared to have negative effects on both the training and dev sets, and increased the RMSE loss to 0.984. Figure 6 shows how a constrained model compares to our best unconstrained model. This was somewhat surprising to us, so we reached out to the authors of Kuchaiev and Ginsburg [2017] to enquire if they had similar results. It turned out that they too had not obtained promising results using it, and had omitted it from their final model. Thus, perhaps a constrained autoencoder is not the best way to address overfitting in this case.

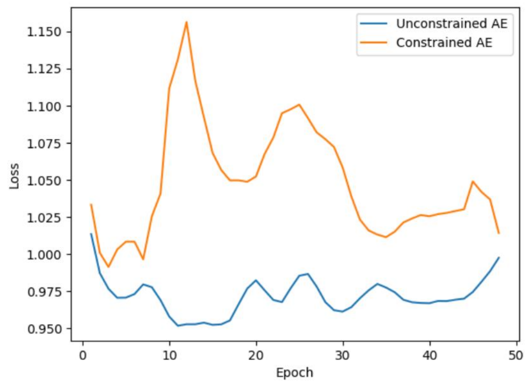

Figure 6: Effect of constraining the weight matrices.

5.4 Dropout

We also tried to mitigate overfitting by adding dropout instead of constraining the weights. This acts as a form of regularization. We only applied dropout on the encoder, and used dropout probabilities between 0.2 and 0.8. We notice that loss decreases as we decreased the dropout, although low dropout rates performed better than the model with no dropout. With a 0.2 dropout we were able to get an RMSE loss of 0.939. This is seen in figure 7.

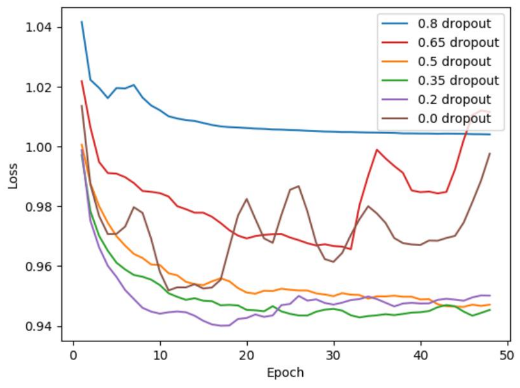

Figure 7: Effect of adding different dropout rates.

5.5 Dense Refeeding

One method in the literature used to improve performance is called dense refeeding. This was adopted by Kuchaiev and Ginsburg [2017].

Note that the user's input $x \in R^d$ is very sparse because users only watch a small percentage of movies. However, ideally we would want the output of the autoencoder, $f(x)$, to be dense as we want predicted ratings of all the movies.

Also note that with a perfect autoencoder $f(x)$,

(i) $f(x)_i = x_i \; \forall x_i \neq 0$

(ii) For any current $x_k = 0$ that receives a new rating x'_k, $f(x)_k$ should be equal to x'_k.

Thus, $f(x) = y$ should be a fixed point: $f(y) = y$. To explicitly enforce both of these constraints, we augment every optimization iteration with a 'dense refeeding' step, which gives the name of the method.

The steps are as follows:

1. Given a sparse input x, compute $f(x)$ using forward propagation.
2. Compute gradients and perform the weight update using backward propagation.
3. Treat $f(x)$ as a new dense example and compute $f(f(x))$.
4. Compute gradients and perform a second weight update.

We implemented dense refeeding on the model selected with 0.2 dropout and $[128, 256, 256]$ layers, however it did not seem to have improved the results as the dev loss was about 0.947. Figure 8 depicts the effect of applying dense refeeding.

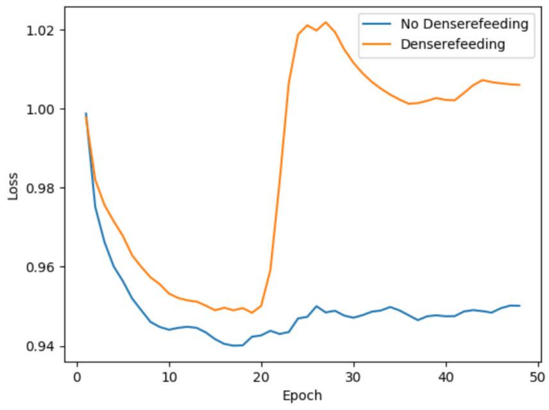

Figure 8: Effect of applying dense refeeding.

5.6 Results on the Full Dataset

After conducting all of the experiments on the mini dataset of 50,000 users, we selected the best model - a 3-layer, [128, 256, 256] autoencoder with 0.2 dropout. We trained this model on the full training set (recall Table 1) , and then tested it on both the full dev and test sets. The final RMSE on the test set was 0.944.

6 Conclusion and Future Work

In this project, we have found that deep learning is able to aid recommendation systems in a meaningful way. We built the system from scratch, which was a fantastic learning experience as we were forced to think about data-wrangling and pipeline questions as well.

Some of the surprising results were that constraining the weight matrices did not improve overfitting that much, and that dense re-feeding was not as effective as suggested in the literature. Nevertheless, it was encouraging to note that going 'deeper' and adding dropout worked as the theory predicted.

It was also encouraging that we obtained respectable results compared to other state-of-the-art models. This is reflected in Table 2. As this was achieved with relatively low resources, we are optimistic that deep learning has much more to add to recommendation systems.

I-AR	U-AR	RNN	DeepRec	Our model
0. 936	0. 965	0. 922	0. 910	0.943

Table 2: Test RMSE of different models on the Netflix dataset.

For a description of what these models are, please refer to the appendix.

Contributions

Omar worked on the dataset preprocessing and the data pipeline. Arjun worked on the dataset exploration. All other parts of the project were equally divided efforts by all members.

Code Repository

https://github.com/hadlaq/AECF

References

Netflix Prize Dataset. https://www.kaggle.com/netflix-inc/netflix-prize-data/data.

O. Kuchaiev and B. Ginsburg. Training Deep AutoEncoders for Collaborative Filtering. *ArXiv e-prints*, Aug. 2017.

Q. Li, X. Zheng, and X. Wu. Collaborative Autoencoder for Recommender Systems. *ArXiv e-prints*, Dec. 2017.

S. Sedhain, A. K. Menon, S. Sanner, and L. Xie. Autorec: Autoencoders meet collaborative filtering. In *WWW*, 2015.

C.-Y. Wu, A. Ahmed, A. Beutel, A. J. Smola, and H. Jing. Recurrent recommender networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 495–503, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4675-7. doi: 10.1145/3018661.3018689. URL http://doi.acm.org/10.1145/3018661.3018689.

S. Zhang, L. Yao, and A. Sun. Deep learning based recommender system: A survey and new perspectives. *CoRR*, abs/1707.07435, 2017. URL http://arxiv.org/abs/1707.07435.

Appendix

Figures for Section 2: Data Exploration

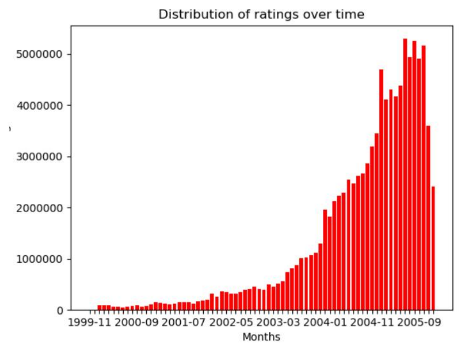

Figure 9: Histogram of ratings submitted over time

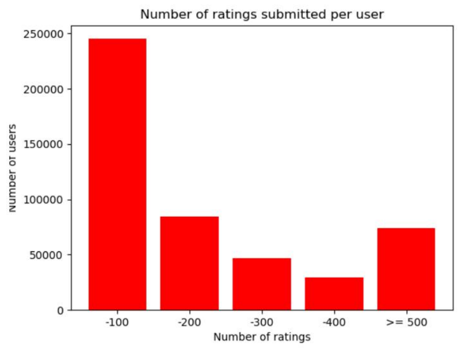

Figure 10: Number of ratings submitted per user.

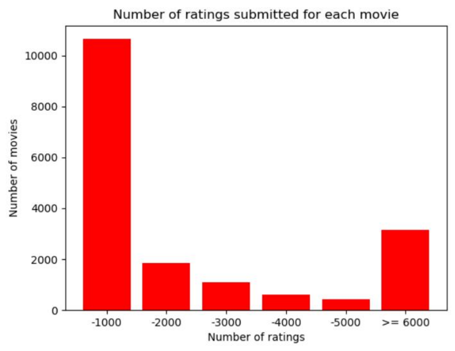

Figure 11: Histogram of number of ratings submitted for each movie.

Legend for Table 2: Other Deep Learning-Based Recommendation System Models

In Table 2, we referred to several other state-of-the-art models. These models are as follows: I-AR is an item-item based recommendation system that is built using autoencoders by Sedhain et al. [2015]. U-AR is the same model, but it's user-user based. U = user-user based. RNN is the RNN model by Wu et al. [2017] which was described in Section 2. Finally, DeepRec is the state-of-the-art autoencoder model by Kuchaiev and Ginsburg [2017].