NeuRec - Automatic Playlist Continuation and Generation using Deep Learning

Shrey Agarwal
University of California, Los Angeles
Los Angeles, California
shrey1995@cs.ucla.edu

Varun Saboo
University of California, Los Angeles
Los Angeles, California
v18saboo@cs.ucla.edu

ABSTRACT

Recommendation is the key driving force behind every customer facing business model. With the growing popularity of Netflix, Amazon Prime, and other streaming services, it has become a widely popular area of research in recent times. Recent development in deep learning has allowed us to implement complicated neural network architectures to outperform the previous state of the art recommendation systems which were based on Matrix Factorization. In this project, we explore techniques such as Neural Collaborative Filtering, Neural Collaborative Autoencoders, and Deep Attention based Encoder-Decoder. We used the Spotify Million Playlist dataset which is released by ACM as part of the Recsys Challenge 2018. We also look at novel playlist embedding techniques to address scalability issues.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; • **Computing methodologies** → **Neural networks**; *Batch learning*; Non-negative matrix factorization;

KEYWORDS

Neural Networks, Recommender Systems, Deep Learning, Implicit Feedback, Collaborative Filtering

1 INTRODUCTION

In the era of information explosion, recommender systems play a pivotal role in alleviating information overload, having been widely adopted by many online services, including E-commerce, online news and social media sites. The key to a personalized recommender system is in modeling users' preference on items based on their past interactions (e.g., ratings and clicks), known as collaborative filtering [14], [17]. Among the various collaborative filtering techniques, matrix factorization (MF) [5], [8] is the most popular one, which projects users and items into a shared latent space, using a vector of latent features to represent a user or an item. Thereafter a user's interaction on an item is modeled as the inner product of their latent vectors. Popularized by the Netflix Prize, MF has become the de facto approach to latent factor model-based recommendation. Much research effort has been devoted to enhancing MF, such as integrating it with neighbor-based models [8], combining it with topic models of item content [16], and extending it to factorization machines [13] for a generic modeling of features. Despite the effectiveness of MF for collaborative filtering, it is well-known that its performance can be hindered by the simple choice of the interaction function-inner product. For example, for the task of rating prediction on explicit feedback, it is well known that the performance of the MF model can be improved by incorporating user and item bias terms into the interaction function. While it seems to be just a trivial tweak for the inner product operator [5], it points to the positive effect of designing a better, dedicated interaction function for modeling the latent feature interactions between users and items. The inner product, which simply combines the multiplication of latent features linearly, may not be sufficient to capture the complex structure of user interaction data. This paper explores the use of deep neural networks for learning the interaction function from data, rather than a handcraft that has been done by many previous work [7],[8]. The neural network has been proven to be capable of approximating any continuous function [6], and more recently deep neural networks (DNNs) have been found to be effective in several domains, ranging from computer vision, speech recognition, to text processing [2], [3], [18]. However, there is relatively little work on employing DNNs for recommendation in contrast to the vast amount of literature on MF methods. Although some recent advances [12], [15] have applied DNNs to recommendation tasks and shown promising results, they mostly used DNNs to model auxiliary information, such as textual description of items, audio features of musics, and visual content of images. With regards to modeling the key collaborative filtering effect, they still resorted to MF, combining user and item latent features using an inner product. This work addresses the aforementioned research problems by formalizing a neural network modeling approach for collaborative filtering. We focus on implicit feedback, which indirectly reflects users' preference through behaviours like watching videos, purchasing products and clicking items. Compared to explicit feedback (i.e., ratings and reviews), implicit feedback can be tracked automatically and is thus much easier to collect for content providers. However, it is more challenging to utilize, since user satisfaction is not observed and there is a natural scarcity of negative feedback. In this paper, we explore the central theme of how to utilize DNNs to model noisy implicit feedback signals. The main contributions of this work are as follows. 1. We present a neural network architecture to model latent features of users and items and devise a general framework NCF for collaborative filtering based on neural networks which is inspired by the NCF paper [4]. 2. We try a new approach employed by NVIDIA using Deep Autoencoders for the same task by making certain subtle changes to handle the scalability issues. 3. We make sure that our results are qualitative by constantly submitting our test reports to the challenge portal.

(a) userâĂŞitem matrix

(b) user latent space

Figure 1: An example illustrates MF's limitation. From data matrix (a), u_4 is most similar to u_1, followed by u_3, and lastly u_2. However in the latent space (b), placing p_4 closest to p_1 makes p_4 closer to p_2 than p_3, incurring a large ranking loss.

2 PRELIMINARIES

2.1 Implicit Feedback

M and N denote the number of users and items, respectively. We define the playlist-song interaction matrix $Y \in \mathcal{R}^{M \times N}$ from users' implicit feedback as,

$$y_{ui} = \begin{cases} 1, & \text{if interaction (user } u\text{, item } i\text{) is observed;} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Here a value of 1 for y_{ui} indicates that there is an interaction between user u and item s; however, it does not mean u actually likes i. Similarly, a value of 0 does not necessarily mean u does not like i, it can be that the user is not aware of the item. This poses challenges in learning from implicit data, since it provides only noisy signals about users' preference. While observed entries at least reflect users' interest on item, the unobserved entries can be just missing data and there is a natural scarcity of negative feedback.

The recommendation problem with implicit feedback is formulated as the problem of estimating the scores of unobserved entries in Y, which are used for ranking the items. Model-based approaches assume that data can be generated (or described) by an underlying model. Formally, they can be abstracted as learning $\widehat{y}_{ui} = f(u, i|\theta)$, where $\widehat{y}_{ui}$ denotes the predicted score of interaction y_{ui}, θ denotes model parameters, and f denotes the function that maps model parameters to the predicted score (which we term as an interaction function).

2.2 Matrix Factorization

MF associates each user and item with a real-valued vector of latent features. Let $\mathbf{p}_u$ and $\mathbf{q}_i$ denote the latent vector for user u and item i, respectively; MF estimates an interaction y_{ui} as the inner product of $\mathbf{p}_u$ and $\mathbf{q}_i$:

$$\widehat{y}_{ui} = f(u, i|\mathbf{p}_u, \mathbf{q}_i) = \mathbf{p}_u^T \mathbf{q}_i = \sum_{k=1}^{K} p_{uk} q_{ik} \quad (2)$$

where K denotes the dimension of the latent space. As we can see, MF models the two-way interaction of user and item latent factors, assuming each dimension of the latent space is independent of each other and linearly combining them with the same weight. As such, MF can be deemed as a linear model of latent factors.

2.2.1 Drawback of MF. 1 illustrates how the inner product function can limit the expressiveness of MF. There are two settings to be stated clearly beforehand to understand the example well. First, since MF maps users and items to the same latent space, the similarity between two users can also be measured with an inner product, or equivalently , the cosine of the angle between their latent vectors. Second, without loss of generality, we use the Jaccard coefficient as the groundtruth similarity of two users that MF needs to recover. Let us first focus on the first three rows (users) in Figure 1a. It is easy to have $s_{23}(0.66) > s_{12}(0.5) > s_{13}(0.4)$. As such, the geometric relations of $_1$, $_2$, and $_3$ in the latent space can be plotted as in Figure 1b. Now, let us consider a new user u_4, whose input is given as the dashed line in Figure 1a. We can have $s_{41}(0.6) > s_{43}(0.4) > s_{42}(0.2)$, meaning that u_4 is most similar to u_1, followed by u_3, and lastly u_2. However, if a MF model places p4 closest to p_1 (the two options are shown in Figure 1b with dashed lines), it will result in p_4 closer to p_2 than p_3, which unfortunately will incur a large ranking loss. The above example shows the possible limitation of MF caused by the use of a simple and fixed inner product to estimate complex userâĂŞitem interactions in the low-dimensional latent space. We note that one way to resolve the issue is to use a large number of latent factors K. However, it may adversely hurt the generalization of the model (e.g., overfitting the data), especially in sparse settings [13]. In this work, we address the limitation by learning the interaction function using Deep Neural Networks from data.

2.3 Approximate Nearest Neighbour Oh Yeah (ANNOY)

There are many ways to search through this corpus of word-embedding pairs for the nearest neighbors to a query. One way to guarantee that you find the optimal vector is to iterate through your corpus and compare how similar each pair is to the query. This however is incredibly time consuming and not often used. Vectorized cosine distance is notably faster than the iterative method but still may be too slow. This is where approximate nearest neighbors shines: returning approximate results but blazingly quickly [10]. The search involves creating a large number of indexed trees - larger the number of trees, higher the accuracy of search but slower speed of search. Upon a query, the algorithm searches over those trees to find the nearest neughbours. This technique is approximated by searching some nodes in the trees rather than all of them. For our project purpose, we made use of the popular ANNOY library developed by Spotify. We used a total of 250 trees to index our 1 million playlists.

3 METHODOLOGY

We explain our model designs briefly in the following sections.

First we discuss Neural Collaborative Filtering method that was inspired by [4]. The main issue of scalability will be addressed in section **3.1**.

We then explore the second approach which is inspired by [9]. The architecture was primarily developed to handle Netflix dataset and hence needed severe modifications to incorporate the 1 Million Playlist Dataset. We describe the model in detail in section **3.2**

3.1 *Neural* Collaborative Filtering

To permit a full neural treatment of collaborative filtering, we adopt a multi-layer representation to model a playlist-song (i.e. userâĂŞitem) interaction y_{ui} as shown in Figure 2, where the output of one layer serves as the input of the next one. All notations in our paper address playlist-song interaction even though we regularly use user-item wordings to address some of the basic recommendation concepts. Instead of using a pure neural netwrok approach, we also retain the Matrix Factorization technique by learning linear transformations in the GMF (General Matrix Factorization). The bottom input layer consists of two feature vectors v_u^U and v_i^I that describe user u and item i, respectively. These vectors are generally one-hot encoded, but can also be initialized with context aware vectors when working with content-based recommendation. Above the input layer is the embedding layer. There are two sets of embedding layers for each user and item. One embedding layer will learn the latent space for GMF, whereas the other embedding layer learns the embedding space for the multilayer perceptron (MLP). The user embedding and item embedding are then fed into a multi-layer neural architecture to map the latent vectors to prediction scores. Each layer of the neural CF layers can be customized to discover certain latent structures of userâĂŞitem interactions. The dimension of the last hidden layer X determines the modelâĂŹs capability. The final output layer is the predicted score $\widehat{y}_{ui}$, and training is performed by minimizing the pointwise loss between $\widehat{y}_{ui}$ and its target value y_{ui}. The formulation of the is given as follows:

$$\begin{aligned} \phi^{GMF} &= p_u^G \bullet q_i^G, \\ \phi^{MLP} &= a_L\left(W_L^T\left(a_{L-1}(..a_2(W_2^T\begin{bmatrix} p_u^M \\ q_i^M \end{bmatrix} + b_2)...) + b_L\right), \\ \widehat{y}_{ui} &= \sigma(h^T\begin{bmatrix} \phi^{GMF} \\ \phi^{MLP} \end{bmatrix}) \end{aligned} \tag{3}$$

where p_u^G and p_u^M denote the user embedding for GMF and MLP parts, respectively; and similar notations of q_i^G and q_i^M for item embeddings. We use ReLU as the activation function of MLP layers, which is more biologically plausible and proven to be non-saturated; moreover, it encourages sparse activations, being well-suited for sparse data and making the model less likely to be overfitting. This model combines the linearity of MF and non-linearity of DNNs for modelling userâĂŞitem latent structures. During training, the loss function is a simple binary cross entropy loss as the target is either 1 or 0. Mathematically, it can be represented as:

$$L = -\sum_{(u,i)\in Y\cup Y'} y_{ui}\log\widehat{y}_{ui} + (1-y_{ui})\log(1-\widehat{y}_{ui}) \tag{4}$$

3.1.1 Shortcomings. This model seems simle to understand, however, there are grave issues when dealing with large scale data. Because we're training only binary entries, i.e. user u and item i, we are only learning to predict positive values. To handle the implicit feedback, we introduce negative sampling. For every 1 positive input to our neural network, we create 3 negative entries. Thus quadrupling the training set size instantly. In our dataset, we had almost 60 Million entries. Thus, quadrupling this results in a training set of 240M entries. Such large amounts of trainnig data will take a very very long time to complete even one epoch. Another drawback of this approach was the one-hot encoding. Given that we have 1 million unique users and 2.3M unique songs, we cannot represent so much data in one-hot encodings in a standard 16GB RAM. Takin into account the number of model parameters and weight matrices, we could require upto 64GB in Memory alone. This works well for large well established companies, but however is unsuitable for research. We can address the latter issue by introducing song embeddings instead of one-hot encodings and we will address that in future work.

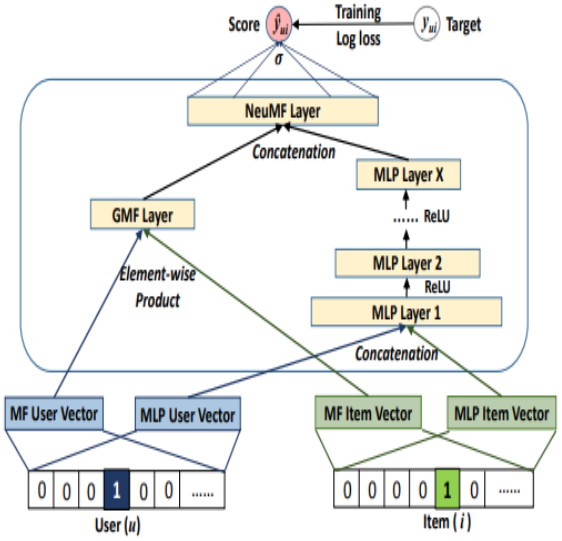

Figure 2: Neural Collaborative Filtering [4]

3.2 Deep Autoencoders

Following the work of NVIDIA [9], we used the architecture of deep autoencoder to generate recommendations. The input to this model was a NxS matrix, where N is the number of playlists and S is the total number of unique songs in the dataset. Each row in the matrix is a vector of 0s and 1s as the paper originally uses where 1s indicate that the song at index i is present in the playlist.

This brings us to the next challenge. The dataset we are looking at has 1 Million playlists with 2.3 million unique songs. We are now looking at a 1 million x 2.3 million matrix. This is a huge matrix which is highly sparse in nature. The model learns to over-fit the output vectors to all 0s with a constant minimum loss. The next logical thing was to reduce the number of unique songs based on the frequency and after filtering songs which have a frequency of less than 10, we were left with 220k unique songs which was still really huge and the model did not learn to recommend anything useful. Reducing the songs based on a higher frequency would not make sense anymore because we will be constraining our recommended to pick songs from a very limited dataset.

We thought of a novel approach of making use of song embeddings. We train a song2vec model on our playlists consisting of

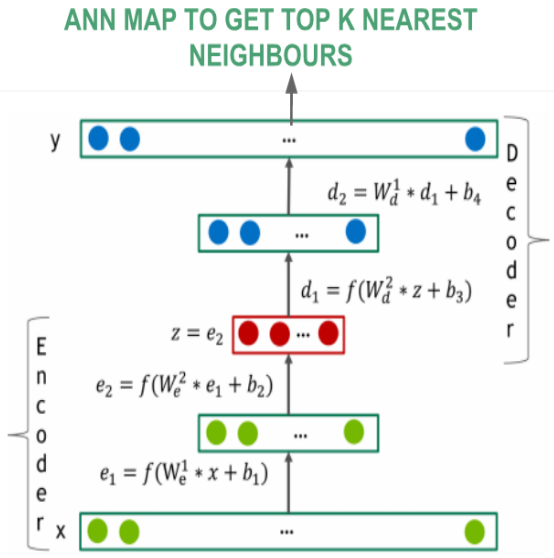

Figure 3: Final Architecture using Deep Autoencoder and Approximate Nearest Neighbours [9]

songIDs with the help of gensim's word2vec. The embedding dimension is set to 256 and min_count is 5. This way each playlist now is a list of song embeddings. Next step is to convert list of song embeddings into a playlist embedding. We discuss two approaches of this in next subsection **2.2.1** Once we obtain the playlist embedding, we train our autoencoder with this playlist embedding for all 1 million playlists and the output is exactly the same as input. We train our autoencoder with 4 encoder layers and 4 decoder layers of units (256, 128, 128, 64). The loss function used by NVIDIA [9] was MSE loss which makes sense if your output is a vector of 0s and 1s and the last layer is a softmax. But we are dealing with embeddings which are float values. We therefore considered Cosine Embedding loss which would give us the distance between two embeddings (predicted, expected). Optimizing this loss will train the model in the best way possible for embedding generation. The Cosine Embedding Loss is:

$$L(x_1, x_2, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y == 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y == -1 \end{cases} \quad (5)$$

Another loss function that we found suitable to predicting embeddings was Von Mises Fisher Loss. This loss function was recently popularized in <insert ref>. It is a combination of Gaussian and Cosine loss and provides a better, more meaningful loss when predicting embeddings. The Cosine loss finds the difference in angle between the embeddings and the Gaussian loss finds thed deviation, thus combining them into a single function should work better. We hope to implement this in the future.

The model takes approximately 1 hour to train. We now perform an ANNOY (Approximate Nearest Neighbor) technique on the playlist embeddings. We find the 100 most nearest playlists to the predicted playlists and extract the songs of these playlists. We then rank the songs based on the similarity score of the playlists and the frequency of the songs in these 100 playlists. We pick the first 500 songs for each playlist and submit as a CSV to the ACM Recsys Challenge. We were able to get a rank of 76 by this approach.

We also tried to generate an ANNOY map on song embeddings and find 500 nearest songs to a playlist embedding which surprisingly works better than the previous approach and brought us to a rank of 75.

At the time, we didn't realize a glaring logic error in our idea. How can the autoencoder model learn to recommend anything meaningful if we don't allow it to try to learn a new playlist embedding? We thought of completely scraping the autoencoder model and just used the playlist embeddings we get from subsection **2.2.1** to find the 500 nearest songs using the Approximate Nearest Neighbors algorithm. We were able to boost up our rank to 50th position by finding 500 nearest song embeddings given a playlist embedding and to a rank of 57 when we searched nearest playlists to a given playlist.

3.2.1 Playlist Embedding Generation. The problem of generating playlist embedding from a list of song embeddings can be thought of as similar to the problem of generating a sentence embedding given a list of word embeddings or generating document embedding given a list of sentence embeddings. We read <insert link> so and so papers to represent a playlist with a proper embedding. The first idea was to simply take average of all the song embeddings in the playlist to generate the playlist embedding. This simple idea though quite flawed, as some might argue works quite decently.

The next method we tried was to use a machine learning model to learn the playlist embedding from a list of song embeddings. The moment we talk about sequence of input, the first idea that works quite well in this domain is LSTMs. We use an encoder-decoder model [1] to learn the playlist embeddings and hope to generate meaningful representations. The test on this model is yet

to be performed. Please note that each submission takes about 1-2 days to update on the Recsys Leaderboard and hence we were constrained in trying multiple approaches.

4 DATASET

The Million Playlist Dataset provided by Spotify consists of one million playlists with varying number of tracks in each. There were a total of 65 million tracks in all playlists and more than 2 million unique tracks. Each track had additional metadata such as artist, album and duration. Each playlist was identified by a unique ID and a name. The challenge dataset provided by Spotify considered the following cases:

- Playlist with no seed tracks
- Playlist with one track
- Playlist with 2-5 tracks
- Playlist with multiple tracks
- Playlist with no name but some tracks

4.1 Data Preprocessing

Each track and artist in the dataset was represented by a unique Spotify URI of almost 40 characters. First, we created an indexed mapping for all tracks, artists and between songs and artists. While analyzing dataset we observed that almost 1.6 million tracks occur only 5 times! When we're dealing with over 60 million tracks, such a less number doesn't affect our results given that over 100K tracks repeat at least 40 times. Thus we filtered the dataset to include only tracks that occur more than 20 times, reducing the number of unique tracks from 2.3M to 230K.

4.2 Handling Empty Playlists

This was an interesting case and we designed our model to handle empty playlists in the following way: If the playlist is empty, we take its name and apply fuzzy string matching against all the other unique playlist names. After we find the top 10 similar playlist names, we get all the songs from these playlists and generate a playlist embedding from the song embeddings. This playlist embedding is treated like any other playlist embedding which goes through the usual channel of autoencoder and then ANNOY. (or just ANNOY after scrapping autoencoder model).

5 EVALUATION

We used two test metrics as suggested by ACM Recsys Challenge: We denote the ground truth set of tracks by G, and the ordered list of recommended tracks by R

R-Precision

R-precision is the number of retrieved relevant tracks divided by the number of known relevant tracks (i.e., the number of withheld tracks):

$$R-precision = \frac{|G \cap R_{1:|G|}|}{|G|} \tag{6}$$

Normalized discounted cumulative gain (NDCG)

Discounted cumulative gain (DCG) measures the ranking quality of the recommended tracks, increasing when relevant tracks are placed higher in the list. Normalized DCG (NDCG) is determined by calculating the DCG and dividing it by the ideal DCG in which the recommended tracks are perfectly ranked:

$$DCG = rel_1 + \sum_{i=2}^{|R|} \frac{rel_i}{\log_2(i+1)} \tag{7}$$

The ideal DCG or IDCG is, on our case, equal to:

$$IDCG = 1 + \sum_{i=2}^{|G|} \frac{1}{\log_2(i+1)} \tag{8}$$

If the size of the set intersection of G and R, is empty, then the DCG is equal to 0. The NDCG metric is now calculated as:

$$NDCG = \frac{IDCG}{DCG} \tag{9}$$

6 RESULTS

The Table 1 summarizes our results on various approaches that we took. One thing to note before looking at the ranks is that the ranks are generated relative to other teams and hence hold no meaning. In fact, models at rank 50 and 57 had almost similar metrics and would have differed by a rank of 1 if they had been ranked on same day.

From the results, its quite clear that with each improvement in the architecture of the model, we are getting closer to a better recommendation. The results only prove our flawed logic of using embeddings in Autoencoders. We need to find a way to use autoencoder architecture for recommendation (More on this in **Future Work**). NCF model shows quite promising results but lacks in scalability. If we can somehow scale the issues mentioned in section **3.1**, we can really improve our test metrics further.

7 CONCLUSIONS

The idea of recommending while making use of Deep Neural Networks is a rich idea which has exploited in this paper for the ACM Recsys Challenge. Our increase in rank for different improvements over the base model is an indicator that DNN have the potential to form recommenders of the future.

By making use of the concept of embeddings, we were able to scale the model for the 1 Million playlists and more importantly 2.3 Million unique songs which would have otherwise required 10x times RAM and GPU computational power. Even though are rank is not high, we believe we tried a new architecture which can be transferred to any other approach to scale large datasets.

8 FUTURE WORK

We have some ideas to train our encoder-decoder model to improve the playlist embeddings. We plan on making use of clustering algorithms on the songs to generate cluster labels for each song. The encoder-decoder model will now try to predict the cluster labels of each song given song embeddings at each time-step. By extracting the hidden representation from the last layer of this model will give us the playlist embeddings.

This paper [11] inspired us to rethink our architecture. The paper discusses the use of mini-batch to generate a vocabulary based on some rules which is specific to that mini-batch and considerably

Table 1: Models Comparison

Model	Leaderboard Rank	R-Precision	NDCG
Team1	1	0.22313	0.393859
Team2	2	0.216227	0.374434
NCF	46	0.140597	0.280991
$ANNOY_{songemb}w/AutoEnc$	-	-	-
$ANNOY_{songemb}w/oAutoEnc$	50	0.116227	0.24434
$ANNOY_{avg-playlistemb}w/AutoEnc$	76	0.044656	0.0846
$ANNOY_{avg-playlistemb}w/oAutoEnc$	57	0.059304	0.20546
$ANNOY_{enc-decplaylistemb}w/AutoEnc$	-	-	-
$ANNOY_{enc-decplaylistemb}w/oAutoEnc$	-	-	-

lesser in size and easy to represent in vector and hence easily scalable on Machine Learning models. If we somehow incorporate this technique on our encoder-decoder model as well as NCF model, we will be able to take advantage of 1 million playlists and scale our models to get higher accuracies.

ACKNOWLEDGMENTS

We would like to thank Prof Miodrag for permitting us to do this project under his guidance. We would also like to thank ACM and Spotify for providing us with the dataset.

REFERENCES

[1] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. 2017. Massive Exploration of Neural Machine Translation Architectures. *CoRR* abs/1703.03906 (2017). arXiv:1703.03906 http://arxiv.org/abs/1703.03906

[2] Ronan Collobert and Jason Weston. 2008. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*. ACM, New York, NY, USA, 160–167. https://doi.org/10.1145/1390156.1390177

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[4] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 173–182. https://doi.org/10.1145/3038912.3052569

[5] Xiangnan He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. 2016. Fast Matrix Factorization for Online Recommendation with Implicit Feedback. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '16)*. ACM, New York, NY, USA, 549–558. https://doi.org/10.1145/2911451.2911489

[6] K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.* 2, 5 (July 1989), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[7] Longke Hu, Aixin Sun, and Yong Liu. 2014. Your Neighbors Affect Your Ratings: On Geographical Neighborhood Influence to Rating Prediction. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR '14)*. ACM, New York, NY, USA, 345–354. https://doi.org/10.1145/2600428.2609593

[8] Yehuda Koren. 2008. Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*. ACM, New York, NY, USA, 426–434. https://doi.org/10.1145/1401890.1401944

[9] O. Kuchaiev and B. Ginsburg. 2017. Training Deep AutoEncoders for Collaborative Filtering. *ArXiv e-prints* (Aug. 2017). arXiv:stat.ML/1708.01715

[10] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2016. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement (v1.0). *CoRR* abs/1610.02455 (2016). arXiv:1610.02455 http://arxiv.org/abs/1610.02455

[11] Haitao Mi, Zhiguo Wang, and Abe Ittycheriah. 2016. Vocabulary Manipulation for Neural Machine Translation. *CoRR* abs/1605.03209 (2016). arXiv:1605.03209 http://arxiv.org/abs/1605.03209

[12] Aäron van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep Content-based Music Recommendation. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13)*. Curran Associates Inc., USA, 2643–2651. http://dl.acm.org/citation.cfm?id=2999792.2999907

[13] Steffen Rendle. 2010. Factorization Machines. In *Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM '10)*. IEEE Computer Society, Washington, DC, USA, 995–1000. https://doi.org/10.1109/ICDM.2010.127

[14] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the 10th International Conference on World Wide Web (WWW '01)*. ACM, New York, NY, USA, 285–295. https://doi.org/10.1145/371920.372071

[15] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2014. Collaborative Deep Learning for Recommender Systems. *CoRR* abs/1409.2944 (2014). arXiv:1409.2944 http://arxiv.org/abs/1409.2944

[16] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2015. Collaborative Deep Learning for Recommender Systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. ACM, New York, NY, USA, 1235–1244. https://doi.org/10.1145/2783258.2783273

[17] Hanwang Zhang, Fumin Shen, Wei Liu, Xiangnan He, Huanbo Luan, and Tat-Seng Chua. 2016. Discrete Collaborative Filtering. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '16)*. ACM, New York, NY, USA, 325–334. https://doi.org/10.1145/2911451.2911502

[18] Hanwang Zhang, Yang Yang, Huanbo Luan, Shuicheng Yang, and Tat-Seng Chua. 2014. Start from Scratch: Towards Automatically Identifying, Modeling, and Naming Visual Attributes. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2647868.2654915