FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Deep Latent Factor Models for Recommender Systems
Student:	Bc. Radek Bartyzal
Supervisor:	Ing. Tomáš Řehořek
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2019/20

Instructions

Survey latent factor models based on neural networks used in recommendation systems.

Implement several of the described models using modern deep learning frameworks.

Design and implement a new architecture able to produce vector representations of both users and items while supporting input of different attribute information.

Evaluate and compare all the implemented models on several standard datasets using multiple metrics.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 24, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Deep Latent Factor Models for Recommender Systems

Bc. Radek Bartyzal

Department of Applied Mathematics
Supervisor: Ing. Tomáš Řehořek

April 25, 2019

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Řehořek for his professional guidance and various insights into the recommender systems. I would also like to thank my family and friends for their support throughout my whole studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on April 25, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Bartyzal, Radek. *Deep Latent Factor Models for Recommender Systems.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Doporučovací systémy nám napomáhají objevit zajímavé produkty v široké nabídce. Jedním z typů algoritmů generujících doporučení jsou faktorizační modely. V této práci popisujeme moderní faktorizační modely založené na neuronových sítích. Čtyři z nich také implementujeme. Dále představujeme nový faktorizační model Hybrid cSDAE založený na neuronových sítích, který dokáže zpracovat, jak interakční informace, tak různé druhy atributů. Všechny implementované modely jsou porovnány na standardních datasetech za stejných podmínek.

Klíčová slova Doporučovací systémy, Umělé neuronové sítě, Faktorizační modely

Abstract

Recommendation systems help users discover relevant items. One of the types of models used to generate the recommendations are latent factor models. We survey the state of the art neural network based latent factor

models and implement four of them. We also design and implement a novel architecture of a deep latent factor model called Hybrid cSDAE that is able to process both the rating and attribute information. We comprehensively evaluate the implemented models on standard datasets.

Keywords Recommender systems, Artificial Neural Networks, Latent Factor Models

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The amount of digital information available in the world is increasing. This trend has been going for many years and it does not show any signs of slowing down. For example just Amazon offers over 1 000 000 books [2] and over 800 000 movies [3], way more than it is possible to consume. The true problem arrives when there are so many choices that we cannot even go through all of them. That is where the recommendation systems come into play.

Recommendation (recommender) systems allow for a personalized experience to be enabled by any site frequented by users looking for some item, for example a book, an article, a job posting, anything really. The system is comprised of one or a combination (ensemble) of algorithms.

There are several types of these recommendation algorithms that are described in Section 2.1. Almost all of them deal with some representation of the users visiting the site and the items offered on the site. Latent factor models is one of the groups of these algorithms able to predict items a specific user would like but also producing low dimensional vector representations of both users and items called user (item) embeddings. The term comes from the act of *embedding* a user/item in a lower dimensional space.

Using embeddings has multiple advantages over using the original representations. Firstly it is simply more efficient to work with lower dimensional vectors in case of the original representation being just a single high dimensional vector. However in many cases the original representation can comprise of multiple differently shaped vectors or even matrices. For example a product can have an image, a text description, price, multiple tags and so on. And these are just the attribute information, each item also

has a rating vector specifying how it was rated by users. Combining all this information into a single low dimensional vector is extremely useful because it allows to train the standard recommendation algorithms on these embeddings without the need to redesign them to support all the different kinds of attribute information available in the different domains.

There are many ways to embed a high dimensional input to a lower dimensional space. This work focuses on methods using neural networks, especially the ones with multiple hidden layers. Such networks are usually called "deep" hence the title of the thesis.

We will present several of the state of the art latent factor models including our proposed architecture, describe their implementation and compare them on multiple datasets under equal conditions.

1.2 Goals

Main goals of this thesis are:

- Survey state of the art latent factor models used in recommendation systems.
- Implement several of the described models using modern deep learning frameworks.
- Design and implement a new deep neural network architecture able to process the rating and attribute information of both users and items.
- Evaluate and compare the implemented models under the same conditions on several standard databases and discuss their performance.
- Examine the effects of chosen hyperparameters on the performance of the proposed model.

1.3 Outline

We start by briefly presenting the basics of recommendation systems and neural networks in Chapter 2. The following Chapter 3 describes the state of the art latent factor models and presents our novel architecture. It also explains our implementation of the models chosen for evaluation. Chapter 4 goes over the evaluation setup, used datasets and the exact way of how we train the models. It ends with a description of all the experiments along with the discussion of their results. The whole work is concluded by the last

Chapter 5 giving a summary of the findings and outlining possible future work.

Chapter 2

Related work

This chapter goes over the basics of recommendation systems, defines notation used in their algorithms and describes the most popular evaluation metrics in Section 2.1. It also introduces neural networks and explains how to train them in Section 2.2. Special attention is paid to the autoencoder architecture in Section 2.3 that will be used frequently by the presented latent factor models.

2.1 Recommendation systems

As has been already hinted at, the popularity of recommender systems has greatly increased in the 21st century with the rapid adoption of the Internet around the world.

One of the ways to define a modern recommendation systems is as: *"Any system that produces individualized recommendations as output or has the effect of guiding the user in a personalized way to interesting or useful objects in a large space of possible options."* [4].

We will be focusing on systems producing personalized recommendations which in case of algorithms presented in this work means solving a Top-N recommendation task. **Top-N recommendation** task refers to producing a list of N items that should be the most relevant for a given user [5].

A possible way of solving this task is to use available data to predict the unknown rating a certain user would give to the items he did not encounter. The predicted ratings are then filtered, sorted and the top N highest rated remaining items are recommended. This high level description applies to all of the algorithms discussed in the Chapter 3.

There are four basic types of recommender systems: [6]:

- **Collaborative filtering** uses information about user preferences, usually represented by their ratings to find similar users or items that can be then used to predict the unknown ratings [7].

- **Content based filtering** is similar to the collaborative filtering except instead of using the rating information it tries to match the user attributes with the item attributes such as the description of an item, user preferences, location of the user and so on [8].

- **Knowledge based** recommender systems require the user to interact with the system by providing clues to what is he interested in while the system guides him towards the desired items. These systems are not as frequent as the previous ones and very different from the algorithms discussed in this work [6].

- **Hybrid** systems are simply systems that combine one or more of the types explained above [4].

Advantage of collaborative filtering is that it is able to recommend novel items different from the ones already seen by the user. However the downside is that it requires enough rating information about the given user to be able to recommend reasonably. The problem of not having enough rating information for certain users or items is called **cold start** and can be solved for example by returning content based recommendations before we gather enough ratings [9].

Another problem plaguing the collaborative filtering is the **long tail** [10]. It refers to the fact that the majority of ratings are distributed among the minority of items meaning that the majority of items has very little to none ratings. This can be again solved by including content based recommendations that do no suffer from this problem, because they do not have to take the ratings into consideration at all. On the other hand, the rating information is incredibly rich and therefore important for generating quality recommendations which points to a clear conclusion that the best way is to combine both methods.

All the algorithms presented in this work use the rating information which makes them collaborative filtering models. Some of the advanced algorithms also support adding the attribute information of both users and items which means they could be considered hybrid models.

2.1.1 Rating matrix

With the basic functioning of the recommendation systems explained, we can delve deeper into the description of the algorithms themselves. At the core of all the evaluated algorithms are the ratings. They can be either explicit or implicit:

- **Explicit ratings** are given by the user and express his opinion. It can be a simple like or dislike, number of stars or other typical rating mechanisms.
- **Implicit ratings** are inferred from the user's behaviour. For example whether the user clicked on the item, added it to his cart or bought it. The interpretation of these user actions and their translation to a numeric value is up to the designer of the recommendation system.

Both types of ratings are usually represented by discrete or binary numbers and are stored in a rating matrix called R. The matrix R, visualized in Figure 2.1, is a $n \times m$ matrix which consists of rows representing users and columns representing items, r_{12} therefore represents the rating given to item 2 by user 1 [1].

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} & \dots & r_{1m} \\ r_{21} & r_{22} & r_{23} & \dots & r_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & r_{n3} & \dots & r_{nm} \end{pmatrix}$$

Figure 2.1: User-item matrix of rating values [1].

The rating matrix is usually preprocessed before being used in training of a recommendation algorithm. Frequently used preprocessing techniques include:

- **Removing user bias** by subtracting the average rating of each user from his ratings. This is useful if we want to work with the explicit ratings because different users perceive the rating levels differently.
- **Rescaling the ratings** to a range of $[0, 1]$ or $[-1, 1]$. Effectively turning them from discrete to floating point representation by for example min-max normalization [11].

- **Binning the ratings** to either binary values $\{0, 1\}$ or to discrete $\{-1, 0, 1\}$ which is closely connected to the encoding of the unknown values discussed below.

R is expected to be very sparse. That stems naturally from the fact that having too many items for each user to go through is usually the very premise of deploying a recommender system. Meaning that each user has usually only couple of ratings.

The sparsity of the rating matrix brings two problems connected to the unknown rating values:

1. **How to encode them:** We can either consider them as negative ratings or we can assign them some value between a negative and positive rating value.

   If the training uses explicit ratings, the ratings are usually divided into positive and negative by some arbitrary cut off point. The positive ratings are assigned a value of 1 while the negative ratings are assigned -1, which naturaly leaves 0 to be assigned to the unknown ratings. Example of such preprocessing is the original AutoRec article [12].

   In case of using implicit ratings it is typical to assign a value of 1 to all positive user item interactions and to leave all the other ratings, either negative or unknown, to a value of 0. This approach is used in a majority of current publications on this topic because the implicit ratings are easier to obtain in the real world domains [13] [14] [15] [16] [17].

2. **Their role during training:** The explicit ratings can be directly used to train the model to predict the unknown ratings. However it is not so easy with the implicit ratings. If we trained a model only on the positive ratings it would just predict positive for every user-item combination and achieve 0 error. There are multiple ways to prevent this behavior which are discussed along with our choice in the Section 4.3.

With the rating matrix prepared and all the decisions about unknown ratings made, we just need to select an algorithm to train.

2.1.2 Taxonomy of recommendation algorithms

Many recommendation algorithms have been created since the start of the field. They can be divided into three high level groups:

- **Non-personalized models:** This is the simplest approach that recommends the same items to all of the users. Typical choice is the most popular (most rated, purchased, viewed etc.) items in which case the model is called *Top Popular* or *Bestseller*. Another way is to recommend the items with the best average ratings [5].

- **Traditional Data Mining:** The field of Knowledge Discovery in Databases (KDD) has been long interested in extracting information from transactions stored in large databases. Transaction can be for example a set of items purchased by a certain user.

  One of the ways to use these transactions is to mine Association Rules (AR) out of them by an algorithm presented in [18]. Quick explanation of what an AR is follows. A set of all available items is called $I = \{I_1, I_2 \cdots I_n\}$. Each transaction $T \subseteq I$ consists of items purchased together by a certain user. An association rule $X \implies Y$ means that if a set of items $X \subseteq I$ has been purchased there is a high probability that the items in set $Y \subseteq I$, $Y \cap X = \emptyset$ will be purchased as well.

  Using these mined association rules to recommend new items to a user is straightforward. Create a set of items Z purchased by the user. Find all rules $X \implies Y$ such that the user purchased all items in X ($X \subseteq Z$). Now sort the rules according to their confidence and recommend items from Y of the top rules. Confidence of a rule is just a conditional probability of seeing Y, given that we have seen X [19].

- **Collaborative Filtering (CF):** CF is one of the most popular methods and is used in some form by a majority of currently deployed recommender systems [19]. The basic idea is that the system recommends items to a certain user based on other users opinion [20]. The difference between CF and association rules is that CF uses the rating matrix to generate recommendations whereas AR just care about what has been purchased together, not by whom. Also AR typically do not support different rating levels. That gives CF an advantage in ability to learn from what users both like a lot and not like at all.

The collaborative filtering group can be further divided into the following subgroups based on how exactly it uses the interaction information:

- **Neighborhood-based Collaborative Filtering** methods work with the full rating matrix which means that user u is represented by a row $R_{u,\bullet}$ and the item i by a column $R_{\bullet,i}$. They are either user or item based. Both incorporate the k-Nearest Neighbors algorithm and are therefore called *Item-KNN* and *User-KNN*. Both support the same similarity measures such as *Cosine Similarity*, *Adjusted Cosine Similarity* or *Pearson's Correlation Coefficient* [5].

  Item-KNN generates recommendations for user u by looking at the items he rated and approximating the unknown ratings using item-item similarities. More exactly, the unknown rating r_{ui} is calculated from the known ratings given by user u to items similar to the item i. The calculation can be a simple similarity weighted sum of the ratings or it can be a more complex regression model [21].

  User-KNN works almost just like the Item-KNN. To predict the unknown rating r_{ui} it finds the most similar users to u that have rated the item i and combines their ratings [5].

- **Model based algorithms** take the rating matrix and possibly other information as an input and try to model the relationships between items and users using different techniques. This category is described in greater detail in the Section 2.1.3.

2.1.3 Model based algorithms

All model based algorithms for recommendation can be described by a following equation:

$$\hat{y}_{ui} = f_\phi(u, i) \tag{2.1}$$

The model f_ϕ with trainable parameters ϕ is trying to predict the user u's opinion about the item i called y_{ui}. We intentionally call it an opinion or preference instead of rating because some models do not have to predict the exact ratings they can just try to predict the order in which the user u prefers the items.

There are several types of models that reflect different relationships between users and items [22]:

- **Similarity Model (SM):** The SM described in the Equation 2.2 has a lot in common with the previously mentioned *Item-KNN* but instead of using a preset similarity function it learns its own similarity matrix S. The similarity is used as a weight in a weighted sum over the

user u's known ratings Ω_u. The item i is explicitly excluded from the sum however if we are trying to predict an unknown rating, it would not be among the known ratings anyway. The issue of this model is that the size of the similarity matrix S is quadratic in the number of items and therefore quickly becomes infeasible to both store and calculate [23].

$$\hat{y}_{ui} = f_S^{SM}(u,i) = \sum_{j \in \Omega_u \setminus \{i\}} r_{uj} \cdot S_{ij} \tag{2.2}$$

- **Factorized Similarity Model (FSM):** The FSM depicted in the Equation 2.3 follows naturally from the SM problems. Factorizing the large similarity matrix S into two latent matrices P and Q allows the model to be easily stored and to also extract interesting patters [24].

$$\hat{y}_{ui} = f_{P,Q}^{FSM}(u,i) = (\sum_{j \in \Omega_u \setminus \{i\}} r_{uj} \cdot P_j)^T \cdot Q_i \tag{2.3}$$

- **Latent Factor Models (LFM):** The LFM approximate the preference y_{ui} by a dot product between a user embedding U_u and an item embedding V_i. These embeddings can be calculated by a classic matrix factorization of a rating matrix R. They can also be a result of more complex models that compress both the rating information and additional attribute information about the items and users [25].

$$\hat{y}_{ui} = f_{U,V}^{LFM}(u,i) = U_u \cdot V_i^T \tag{2.4}$$

- **Latent Factor Similarity Models (LFSM):** The LFSM described in Equation 2.5 are a straightforward combination of LFM and FSM allowing for compression of both similarity and rating information separately and then combining them together. This leads for example to algorithm *SVD++* which was successful during the famous Netflix price competition that invigorated the recommendation systems research [24].

$$\hat{y}_{ui} = f_{P,Q}^{LFSM}(u,i) = (\sum_{j \in \Omega_u \setminus \{i\}} r_{uj} \cdot P_j + P_u)^T \cdot Q_i \tag{2.5}$$

The main reason why we chose the Latent Factor Models for our research is its ability to incorporate different types of additional information and combine it with the ratings to create multi-purpose embeddings. Meaning

we can not only get the predicted rating (we are going to be predicting ratings with all the evaluated models) as a product of the embeddings but we can also use these embeddings in other algorithms to further improve the whole recommendation system.

2.1.4 Evaluation metrics

With such a large number of possible algorithms to choose from, we need a way to find out which one is the best for our needs. There are several evaluation metrics just for that. They are calculated using the output of a trained model for the testing users T.

The simplest metric is the well known *Root Mean Squared Error* (RMSE) comparing each of the predicted ratings to its known value and calculating the square of the difference. The Ω_u represents the set of items rated by the user u.

$$\text{RMSE} = \sqrt{\sum_{u \in T} \sum_{i \in \Omega_u} (\hat{r}_{ui} - r_{ui})^2} \tag{2.6}$$

Other metrics come from the field of information retrieval (IR). The task of IR is to retrieve relevant documents from a large set which is remarkably similar to the problem of recommendation. All the IR metrics are tied to the confusion matrix shown in Table 2.1. It clearly divides the items (documents) into four categories: *True Positive* (TP), *False Positive* (FP), *True Negative* (TN) and *False Negative* (FN) [26].

items that were:	actually relevant	actually not relevant
predicted relevant	TP	FP
predicted not relevant	FN	TN

Table 2.1: Categorization of items according to their predicted and actual relevance. Predicted relevant items are the recommended ones or in other words, the ones in the top K items.

Among the IR metrics commonly used in recommendation are [27]:

- **Precision** is a fraction of recommended items that were actually relevant:

$$Precision = \frac{|TP|}{|TP|+|FP|} \tag{2.7}$$

- **Recall** also called *True Positive Rate* or *Sensitivity* is the fraction of all relevant items that were recommended:

$$Recall \ = \frac{|TP|}{|TP|+|FN|} \tag{2.8}$$

Both of these metrics are usually calculated using the top N items recommended by the algorithm. It is however standard in the literature to use K instead of N in case of these metrics which is why they are then called *Precision@K* and *Recall@K*.

The so far mentioned metrics have one thing in common: they do not take the order of the returned items into consideration. In case of recommending lists of items, the order is of great interest because the users are more likely to see the items at the top. It is therefore important to have the best possible items at high positions. A metric designed to measure the ranking success is called *Normalized Discounted Cumulative Gain* (NDCG) [28].

$$\mathrm{NDCG_K} = \frac{DCG_K}{IDCG_K} \tag{2.9}$$

The $\mathrm{DCG_K}$ looks at the top K places and calculates the sum of the ith item relevance (rel_i) divided by a smoothly increasing logarithm value. If we consider only binary ratings {1, 0} the following equation 2.10 holds, if not then only the first equality holds.

$$\mathrm{DCG_K} = \sum_{i=1}^{K} \frac{2^{rel_i} - 1}{\log_2(i+1)} = \sum_{i=1}^{K} \frac{rel_i}{\log_2(i+1)} = rel_1 + \sum_{i=2}^{K} \frac{rel_i}{\log_2(i+1)} \tag{2.10}$$

If we normalize the DCG by the best possible DCG called *Ideal DCG* (IDCG) we get the Normalized DCG. The $\mathrm{IDCG_K}$ is the standard $\mathrm{DCG_K}$ calculated for the list of the relevant items (REL) ordered by their relevance.

$$\mathrm{IDCG_K} = \sum_{i=1}^{|\mathrm{REL}|} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{2.11}$$

Both the IR metrics and the $\mathrm{NDCG_K}$ are calculated for each user in T and then averaged to get the final score.

Last commonly used metric is *Coverage*, it is simply the relative amount of items that were recommended compared to the number of all items.

$$\mathrm{Coverage} = \frac{|\text{set of unique items recommended to users T}|}{|\text{set of all recommendable items}|} \tag{2.12}$$

2.2 Artificial Neural Networks and Deep Learning

The Artificial Neural Networks (ANN) started with a perceptron. An algorithm for learning a linear binary classifier invented by Frank Rosenblatt in 1958 [29]. However as other linear classifiers it is able to divide the input space only by carving out simple regions using hyperplanes. This approach is not very robust against variations of the input that are irrelevant to the classification. If we try to differentiate between the images of dogs and wolves, their background is going to be completely irrelevant however it is still going to be present in the input and changing from image to image. A shallow linear model would not be able to correctly classify the images based on raw pixels due to these variations. Which is why complex feature extraction methods were invented to provide the shallow model with representations that are stable and selective to the important aspects of the images. Wanting to get rid of the engineered feature extractors led to the creation of Deep Learning (DL) [30].

Deep learning architecture is a connected stack of simple trainable modules with input at one end and output at the other. Each of the modules takes its input, processes it and outputs a new representation that is more selective and invariant than the ones outputted by the previous modules. The idea is to learn the feature extractors specifically for the current task which makes them more effective than the generic hand-engineered ones.

In terms of ANN, deep learning started with feed-forward multi-layer perceptrons trained by Group Method of Data Handling (GMDH) invented by Ivakhnenko in 1968 [31]. The GMDH learned even the structure of the networks with a variable number of layers and neurons. A much more efficient way to train deep ANNs with fixed differentiable structure by gradient descent has been enabled by the invention of the Backpropagation algorithm in the 1970s. Backpropagation provides an efficient way to calculate the derivative of the loss function with regard to all the parameters [32].

The increasing computational power and invention of various optimization tweaks led to a rejuvenation of the neural network field at the start of the 21st century [33], [34]. Since then deep learning has been successfully applied to many domains and achieved great results. Recommendation systems are not an exception with many different deep learning models being proposed at recent time. We are going to look into them in the next chapter however first we are going to delve deeper into the current training algorithms of the artificial neural networks.

2.2.1 Optimization algorithms

Using the backpropagation algorithm to calculate the gradient of the loss function with regard to every parameter, we can then apply the gradient to update the parameters. They are updated by taking a small step in the opposite direction of the gradient since that is the direction in which the loss function decreases the most in its value. The size of the step is controlled by a parameter η called *learning rate*. This is a general description of a *Gradient Descent* algorithm, however there are many different ways how to calculate the actual step and we will present the ones most relevant to this work.

2.2.1.1 Gradient Descent variants

There are three variants of the basic gradient descent algorithm and all of them share the following inputs:

$$\begin{aligned} \theta_0 &\quad \text{Initial parameters} \\ N &\quad \text{Number of training examples} \\ x_i,\ \forall i < N &\quad \text{Training examples} \\ \hat{y}_i,\ \forall i < N &\quad \text{Labels for the training examples} \\ T &\quad \text{Number of epochs} \end{aligned} \tag{2.13}$$

- **Batch Gradient Descent (BGD)** shown as Algorithm 1 calculates the step as an average of gradients over all the training examples. This approach unfortunately does not scale well because from a certain point you cannot load the whole dataset into memory making each single update extremely slow. Another problem is that it does not allow *online learning* meaning we cannot simply continue training with newly arrived data points.

Algorithm 1: Batch Gradient Descent

for $t \leftarrow 1$ **to** T **do**
  $g \leftarrow 0$
  for $i \leftarrow 1$ **to** N **do**
    $g \leftarrow g + \nabla L(\hat{y}_i, f(\theta_{t-1}, x_i))$
  $g \leftarrow \frac{1}{N} g$
  $\theta_t \leftarrow \theta_{t-1} - \eta g$

- **Stochastic Gradient Descent (original SGD)** shown as Algorithm 2 updates the parameters with the gradient of every training example sequentially. The training data is shuffled between epochs to add an element of randomness resulting in increased chances of finding a better local minimum. This approach alleviates both mentioned problems of BGD, unfortunately it introduces its own one. Due to the frequent updates, the global loss tends to have a very high variance which may help it escape local minima but it also complicates convergence.

Algorithm 2: Stochastic Gradient Descent

$\theta \leftarrow \theta_0$
for $t \leftarrow 1$ **to** T **do**
  shuffle(*dataset*)
  for $i \leftarrow 1$ **to** N **do**
    $g \leftarrow \nabla L(\hat{y}_i, f(\theta, x_i))$
    $\theta \leftarrow \theta - \eta g$

- **Mini-batch Stochastic Gradient Descent (SGD)** described in Algorithm 3 is the logical combination of the two mentioned methods. By updating the parameters after each mini batch of size B the algorithm achieves significantly less variance of the loss while keeping the advantage of frequent updates. It also leverages the fast matrix operations available on current graphical processors. The new hyperparameter B can be selected based on the size of the dataset and available memory to strike a balance between speed and variance. This algorithm is generally referred to as SGD because due to the strong disadvantages of both BGD and original SGD they are very rarely used.

Even though the SGD fixes the mentioned imperfections, it still has two significant problems:

- Finding a good learning rate can prove to be difficult, but we can offload this task to a *learning rate scheduler* that changes it during training based both on elapsed time steps and past performance [35]. Although the adaptable learning rate performs much better than a constant one, the fact that it is identical for all parameters causes problems in situation where each example while having a high dimension has only few non-zero features. Therefore some features occur

Algorithm 3: Mini-batch Stochastic Gradient Descent

$\theta \leftarrow \theta_0$
for $t \leftarrow 1$ **to** T **do**
  shuffle(*dataset*)
  foreach *mini_batch* $\in$ *dataset* **do**
    $g \leftarrow 0$
    for $i \leftarrow 1$ **to** B **do**
      $g \leftarrow g + \nabla L(\hat{y}_i, f(\theta, x_i))$
    $g \leftarrow \frac{1}{B} g$
    $\theta \leftarrow \theta - \eta g$

more frequently than others which is not reflected in the applied learning rate. That results in slower updates to the less frequent features.

- The loss function of a deep neural network is undoubtedly very complex which brings many challenges to optimization. The most profound difficulty in optimizing such a high dimensional non-convex function is however believed to stem from an extensive number of saddle or saddle-like points surrounded by large plateaus with high error [36]. This is where the loss function increases in some dimensions while it is constant or decreasing in other dimensions. A simple three dimensional example can be a slowly descending valley with steep slopes on both sides. The problem stems from the small decrease in value in the one dimension that actually leads to a minimum. An intuitive explanation of what will happen is that the SGD will keep jumping across the valley while moving very slowly in the desired direction.

The following methods alleviate one or both of the described issues [37].

2.2.1.2 AdaGrad

AdaGrad is an adaptive gradient method attempting to solve the issue of some features appearing less frequently than others [38]. The general idea is for the learner to give larger weight to infrequent features when they appear.

This is implemented by taking note of the past gradient updates and using the sum of the squared past gradients to divide the actual learning rate.

We start with a gradient of the loss function L with respect to a parameter i at time t:

$$g_{t,i} = \nabla_\theta L(\theta_{t,i}) \tag{2.14}$$

The classic mini-batch SGD update would look like this:

$$\theta_{t+1,i}^{SGD} = \theta_{t,i} - \eta g_{t,i} \tag{2.15}$$

However the AdaGrad update leverages the past gradient update information to adaptively change the learning rate for each of the parameters separately:

$$\theta_{t+1,i}^{AdaGrad} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2} + \epsilon} g_{t,i} \tag{2.16}$$

The epsilon is used to prevent division by an extremely small number. Even though this new update rule nicely adapts to each parameter, their learning rates keep getting smaller with increasing time steps. This is caused by the sum of squared gradients that can only increase with time resulting in a smaller and smaller effective learning rate, possibly reaching zero and stopping the training entirely. Another issue is the sensitivity to the initial setting of the learning rate. If the gradients are too large at the beginning, the parameter updates will be small for the rest of the training [39].

2.2.1.3 RMSProp and AdaDelta

Both RMSProp and AdaDelta have been invented around the same time to solve the mentioned issue of AdaGrad's diminishing learning rate. The RMSProp has been introduced by G. Hinton in his course at University of Toronto [40]. It is slightly simpler than AdaDelta [39] while using the same idea which is why we will discuss it here.

The central idea is to use a decaying running average of past squared gradients representing gradients from a certain time window instead of using all of them. This ensures that the training will not slow down after a large number of updates.

The running average of past gradients can be effectively calculated as:

$$\mathbb{E}[g^2]_t = \gamma \mathbb{E}[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{2.17}$$

Then we just replace the summation term in the AdaGrad update rule with this running average estimate:

$$\theta_{t+1}^{RMSProp} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} g_t \tag{2.18}$$

The decay rate γ is recommended to be set to 0.9 while a good default learning rate is 0.001. These methods are generally much less sensitive to a different initial learning rates making hyper-parameter tuning easier [39].

2.2.1.4 Adam

The Adam (adaptive moment estimator) method is an improvement of the previously mentioned RMSProp with an addition of an estimate of the first order momentum [41].

Just as the RMSProp the Adam calculates the decaying running average of the squared past gradients, here called v_t. It also calculates the running average of the gradients themselves, called m_t the same way. The m_t and v_t estimate the first and second order moments of the gradient corresponding to the mean and the uncentered variance.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{2.19}$$

The m_t and v_t are unfortunately biased toward zero at the start of training due to their initialization to zero vectors. To correct that bias the Adam algorithm divides the moment estimates with a time sensitive term approaching 1 with increasing number of time steps:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{(1 - \beta_1^t)} \\ \hat{v}_t &= \frac{v_t}{(1 - \beta_2^t)} \end{aligned} \tag{2.20}$$

These bias corrected moment estimates are then used in the actual update rule similarly to the RMSProp and AdaDelta algorithms:

$$\theta_{t+1}^{Adam} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t \tag{2.21}$$

The recommended default values for the hyper-parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. The positive aspects of the second order moment dividing the learning rate have been explained in the previous sections. The addition of the first order moment can be understood as a momentum term. It aims to solve the problem SGD has with getting out of saddle points.

The effects of momentum can be presented on the example with the slowly descending valley given at the end of Section 2.2.1.1.

If we imagine a ball without momentum in such valley, it will keep going up the opposing sides while slowly moving through the valley. If we add momentum to the ball, it will dampen its oscillation while increasing the speed of movement in the direction of consistent decrease in function value. We can compare that to adding weight to the ball.

Transferring the example to the effects on optimization, the updates in the dimensions where the gradient directions keep changing will be smaller while the updates in the dimensions that are consistently going a certain direction will become incrementally larger.

A follow-up research into Adam has uncovered several convergence issues and possible areas of improvement which led to the introduction of new versions such as:

- **NAdam:** Adapt the momentum term with the Nesterov accelerated gradient method [42].
- **AdamW:** Claims to fix the weight decay calculation[43].
- **AmsGrad:** Finds errors in the proof of Adam's convergence and claims to improve it by introducing AmsGrad algorithm [44].

To the contrary of the numerous papers claiming to improve the original Adam algorithm, experimental results show that it in many cases works better or at least as well as its newer variants. While some of them look promising, there is no single variant dominating others at all tested optimization tasks which is why we have chosen to use the default Adam for all of our experiments [45].

2.3 Autoencoders

It is not a rare situation to have access to a large amount of unlabeled data that we would like to use for a certain task. Any model attempting to solve this task would benefit from a compact representation of the available data. In case of recommendation systems, our task is to rank the items for each user and the data is typically the user/item rating or attribute vectors.

The beneficial representation should ideally be compact, retain as much information as possible and contain higher level features obtained by locating patterns in the data. Autoencoders are one of the possible ways to calculate such representations.

2.3.1 Traditional Autoencoder

The traditional Autoencoder, sometimes also called AutoAssociator, is a feed-forward neural network trained to reconstruct its input at the output. It consists of two parts: encoder and decoder. The Table 2.2 specifies the notation used in this section.

x	input vector
d	dimension of the input vector
k	hidden (latent) dimension = number of units in the last encoder layer

Table 2.2: Notation used in the Section 2.3 explaining Autoencoders.

The encoder function $f_{encoder}$ maps the input x into a hidden representation y. It is an affine transformation followed by a nonlinearity:

$$f_{encoder}(x, W, b) = g(Wx + b) = y \tag{2.22}$$

The W is a $d \times k$ weight matrix and b is a $k \times 1$ bias vector. The g is a nonlinear activation function such as Sigmoid.

The decoder function $f_{decoder}$ attempts to reconstruct the input x from the hidden representation y. Its structure (weights $W^{'}$ and biases $b^{'}$) mirrors the encoder one:

$$f_{decoder}(y, W^{'}, b^{'}) = g(W^{'}y + b^{'}) = \hat{x} \tag{2.23}$$

The training of the weights and biases is done by backpropagation following the loss function visible in Equation 2.24. It consists of a reconstruction loss represented by a squared error. It is also possible to use a cross-entropy loss in case of binary inputs and add regularization [46].

$$\mathcal{L} = \|x - \hat{x}\|^2 \tag{2.24}$$

If $k \geq d$ the autoencoder could reach zero reconstruction error by simply learning an identity transformations. Which would not be a helpful representation at all. To prevent that, the hidden dimension is set to be smaller than the input one. This forces the model to compress the available information and therefore create a compact representation retaining as much input data as possible.

2.3.2 Denoising Autoencoder

Even though the traditional autoencoder is capable of compressing the input data into a smaller representation, we would also like it to extract more complex patterns other than whatever clues are beneficial to reconstructing the input.

That can be achieved by corrupting the input vector while still expecting the autoencoder to reconstruct the original clean input. This forces the encoder to extract patterns that are robust to random perturbations introduced by the noise applied to its input. Such autoencoder is called a Denoising Autoencoder (DAE).

Frequently used types of noise are:

- **Additive Gaussian noise** adds a real valued vector sampled from a normal distribution to the original input x. The resulting corrupted vector $\tilde{x}$ is therefore $\tilde{x}|x \sim N(x, \sigma^2 I)$. This method is typically used on real valued inputs and the level of corruption is controlled by σ.
- **Salt and Pepper noise** sets certain elements of x to a minimum or maximum value. Which one it ends up as is decided by a coin flip. This noise is usually applied to integer data that tend to have small number of possible values.
- **Binary Masking noise** sets a portion of elements of the input to 0. This results in effectively disabling these features from the point of the model.

2.3.3 Stacked Denoising Autoencoder

As has been explained in the beginning of the Section 2.2, stacking multiple layers allows the network to extract patterns with multiple levels of abstraction. Which is exactly what is needed to find more efficient ways of compressing the input data [30].

Increasing the number of hidden layers in the Denoising Autoencoder gives rise to a Stacked Denoising Autoencoder (SDAE). Both the encoder and the decoder become feed-forward networks with multiple layers. The number of units in the hidden layers of the decoder typically mirrors the encoder ones however it is not necessary.

The name uses the word "stacked" due to the fact that SDAE was originally presented as a stack of multiple DAEs that were trained locally. This means that if we wanted a two level SDAE, we would first learn a

DAE on the corrupted version $\tilde{x}$ of the input x. Then we would take its trained encoder $f_e^1(\tilde{x}, W_e^1, b_e^1)$ and use its output as an input to a second DAE. The inputs are no longer corrupted after the first DAE. The output of the second trained encoder $f_e^2(f_e^1(\tilde{x}, W_e^1, b_e^1), W_e^2, b_e^2)$ is then considered our compressed representation. Or we can continue to stack further DAEs [46].

Even though the name stayed the same, the training of SDAEs has changed significantly since the rise of deep learning. They are now typically constructed similarly to feed-forward networks with a bottleneck layer in the middle and trained end to end with backpropagation through all the hidden layers.

Chapter 3

Analysis and design

This chapter introduces several latent factor model architectures including our proposed Hybrid cSDAE and exactly describes our implementation of the ones we chose to evaluate.

3.1 Common characteristics

Since all the discussed architectures are Latent Factor Models, they share a lot of similarities. That allows us to use common notation specified in the Table 3.1.

n	number of users
m	number of items
k	embedding dimension - same for user and item
R	pre-processed $n \times m$ rating matrix
$\tilde{R}$	R after application of noise
$\hat{R}$	approximation of R generated by the model
λ	regularization multiplier
g	dimension of the user attribute vector
h	dimension of the item attribute vector
G	$n \times g$ matrix of user attribute vectors
H	$m \times h$ matrix of item attribute vectors
U	$n \times k$ matrix of user embeddings
V	$m \times k$ matrix of item embeddings
X_i	i-th layer of a network
X_i'	i-th layer of a network symmetric to the layer X_i

Table 3.1: Notation common to the architectures described in Chapter 3.

The following sections go over the different model architectures. All of them are based on neural networks and use the Sigmoid activation function as is standard in similar publications [12], [47], [22].

The evaluated models are optimized by the Adam algorithm described in the Section 2.2.1. It requires a calculated gradient of a loss function with respect to all the trainable parameters. Fortunately the TensorFlow library provides an automated way to compute them from any supported function. All we therefore have to do to train a model is to specify its loss and run the Adam algorithm on it with proper training data. The loss functions of the models are described in their respective sections while the common creation of the training batches is described in the Section 4.3.

3.2 Matrix Factorization

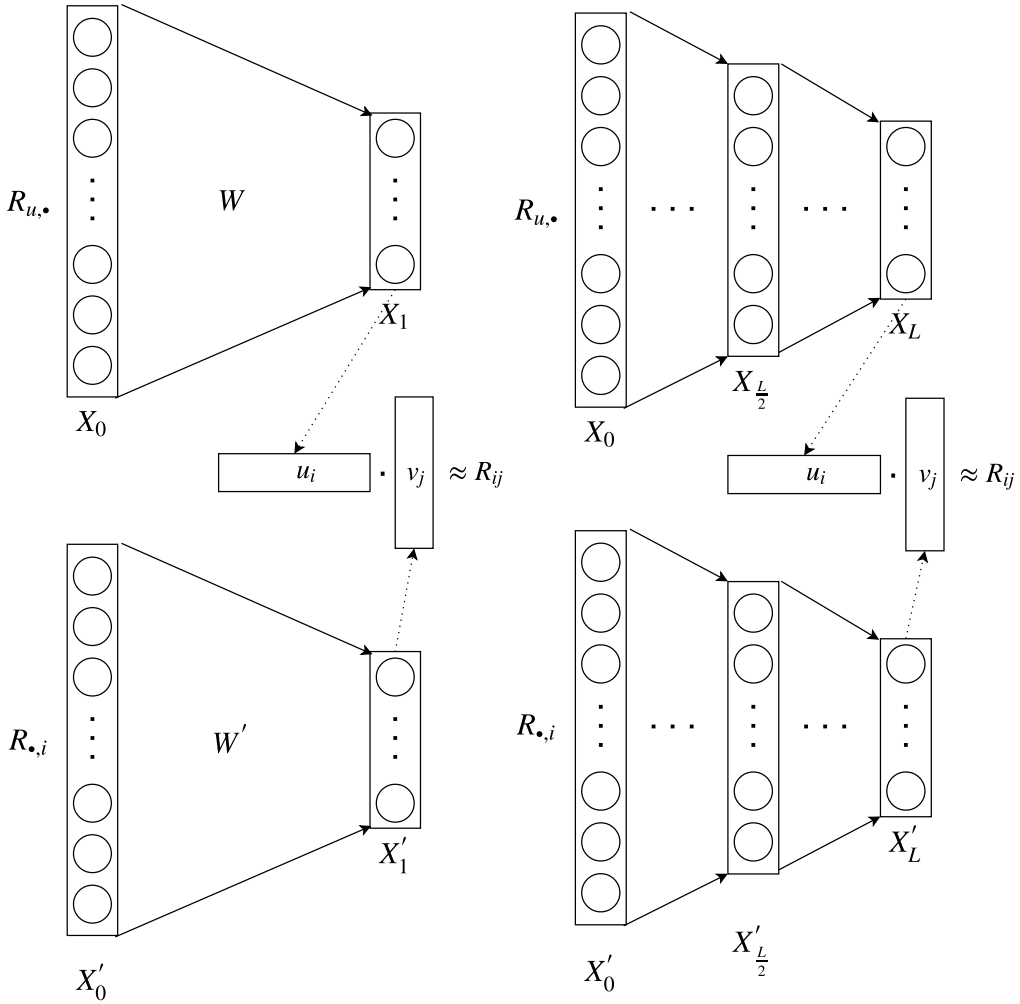
One of the most basic latent factor models is the Matrix Factorization (MF) [24]. The goal of MF is to decompose a large matrix into two smaller ones in such a way that their dot product approximates the original matrix. In the world of recommendation, the large matrix would be the rating matrix R which is expected to be very sparse. MF is intuitively a good fit for this case because it should be possible to efficiently compress the sparsely distributed information. However we do not know how complex are the relationships between the observed ratings and we therefore have to make some assumptions before selecting the right model to try to infer them.

Formally the MF is trying to minimize the loss function $\mathcal{L}$ in the following equation:

$$\mathcal{L} = \left\| R - UV^T \right\|^2 + \lambda \left\| U \right\|^2 + \lambda \left\| V \right\|^2 \tag{3.1}$$

The way we calculate the U and V basically specifies our model. If we assume a linear relationship between the input rating vectors and their latent representation we arrive at the simplest architecture depicted in Figure 3.1a.

As can be seen in the aforementioned Figure 3.1a, the architecture consists of two feed-forward neural networks. One of them takes the user rating vector as input and is therefore called the *user part* while the other one is called the *item part* of the model and processes the item rating vectors. Both of them output k dimensional latent representations of their respective inputs. The dot product of those embeddings then approximates the true ratings from R.



(a) MF architecture.

(b) Deep MF architecture.

Figure 3.1: Matrix factorization architectures.

We have chosen to design this model as a neural network because we wanted it to be easily expandable. This allows us to compare it with the more advanced models that are all based on neural networks similar to the one used for MF. Since we use the Sigmoid activation function (σ) in the hidden layer we allow the model to use some non-linearity in its calculation of embeddings. Using an identity function would result in purely linear operations but it would also complicate comparing the performance of this simple model with the Deep MF that has more hidden layers which of course use the Sigmoid activations. We are especially interested in the possible improvements gained by the increased number of layers, not by the

introduction of non-linearity by activation functions. Additionally we have observed that using identity functions leads to a very large decrease in all measured metrics which means that using the non-linear ones would solely overshadow any effects caused by the changes in the architecture.

The full loss function of the model is expressed exactly in the following equation:

$$\begin{aligned}\mathcal{L} = &\left\| R - \sigma(RW + b) \cdot \sigma(RW^{'} + b^{'})^{T} \right\|^2 \\ &+ \lambda(\|W\|^2 + \left\|W^{'}\right\|^2)\end{aligned} \tag{3.2}$$

We do not regularize the biases because we did not observe any benefits of it. It is also recommended by Jia *et al.* as a way to prevent overfitting [48].

Another advantage of the neural network based model is the ease of evaluation on new users. If we were for example using Alternating Least Squares (ALS) to optimize the MF, we would have to simulate the training step minimizing the loss with regard to the calculated item embeddings similarly to the Collaborative Deep Learning evaluation described in Section 3.7. In the case of this symmetric architecture we simply pass the user rating vector through the user part of the network and retrieve the embedding. The network inference is potentially less expensive than the aforementioned method because it consists only of a sequence of matrix multiplications which can be efficiently computed by the modern graphical processors.

3.3 Deep Matrix Factorization

The Deep Matrix Factorization (Deep MF) is a straightforward extension of the previously introduced Matrix Factorization architecture discussed in Section 3.2.

Since the MF already uses Sigmoid activations the only difference is the addition of the hidden layers. We have decided to add two of them with exponentially decreasing number of units. The resulting layers and their unit counts are specified in Table 3.2. The item part of the architecture not included in the Table 3.2 is symmetric to the user part.

We base the number of neurons on the desired dimension of the latent representation k. As can be seen from the table, the number of neurons N_l in hidden layer $l \in \{1, \ldots, L\}$ can be derived as:

$$N_l = k2^{L-l} \tag{3.3}$$

layer name	# units	description
X_0	m	input layer processing the user rating vector has a dimension equal to the number of items
X_1	256	first hidden layer
X_2	128	second hidden layer
X_3	64	third and final layer that outputs the latent representation

Table 3.2: Deep Matrix Factorization layers and the number of their units.

Using powers of two is standard in the area of neural networks as it is in other computationally intensive fields. It leads to an efficient way of utilizing the resources that all work in the base-2 numeral systems.

The decreasing layer size allows the model to gradually compress the input information while extracting more complex representations of it with each subsequent layer. This should ideally lead to a higher density of information stored in the embeddings compared to the simpler MF model. Which in turn should potentially lead to a better performance during evaluation.

Our implementation of this model minimizes the following loss function:

$$\mathcal{L} = \left\| R - UV^T \right\|^2 + \lambda(\|W_L\|^2 + \left\|W_L^{'}\right\|^2) \tag{3.4}$$

Not only do we not regularize the biases as in the MF model but we also do not regularize any weight matrices except the one calculating the last layer X_L. Again the reason is that we observed this configuration to perform better. However it is possible that it applies only to networks of this size and the situation changes with increased number of layers or their number of units.

3.4 Hybrid Stacked Denoising Autoencoder

Hybrid Stacked Denoising Autoencoder (Hybrid SDAE)is a simplified version of the Hybrid Additional SDAE presented in [13] but without the added side information.

Since it does not make use of the attribute information it is therefore not a hybrid collaborative filtering model in the strict sense however we choose to include the ”hybrid” word in the name to emphasize its connection to the more complex variant.

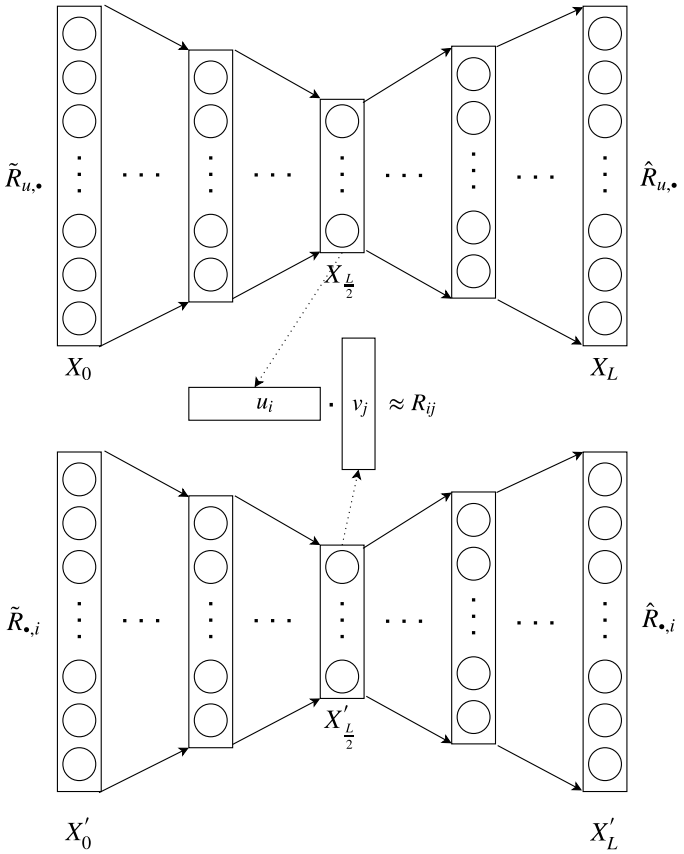


Figure 3.2: Hybrid SDAE network architecture.

The Hybrid SDAE architecture visualized in Figure 3.2 consists of two stacked denoising autoencoders (SDAEs). One of them processes the user rating information and is therefore called the user autoencoder. The other one processes the item rating vectors and is called the item autoencoder. The outputs of their bottleneck layers approximate the known ratings from R by a dot product.

As can be seen, the whole architecture can be also understood as an enhancement of the previous Deep Matrix Factorization with the user and item parts replaced by the SDAEs.

Using the autoencoders instead of simple feed-forward networks has the following advantages:

- The embeddings are forced to contain as much of the original information as possible because the decoder needs to be able to reconstruct it. This is important in the case when we do not only care about the approximated ratings but also about the embeddings themselves and

want them to be a good compressed representation of all the original information.

- The generalization capabilities of the model can be improved by feeding it a noisy version of input that needs to be denoised and subsequently reconstructed. This forces the model to extract more general rules from the input data and not to get fixated at specific features that would allow the model to cheat the loss function and achieve low error while learning sub-optimal representations.

The layer structure of the autoencoders simply expands the Deep Matrix Factorization architecture. This results in the first $\lceil \frac{L}{2} \rceil$ encoder layers X_0, X_1, X_2 and X_3 staying the same as they are described in Table 3.2 while the new decoder layers X_4, X_5, X_6 mirror the encoder layer counts, specifically the X_2, X_1 and X_0.

The model's loss function:

$$\begin{aligned}\mathcal{L} = &\left\|R - UV^T\right\|^2 \\ &+ \sum_i \left\|R_{i,\bullet} - f_d^{user}(\tilde{R}_{i,\bullet}, W, b)\right\|^2 \\ &+ \sum_j \left\|R_{\bullet,j} - f_d^{item}(\tilde{R}_{\bullet,j}, W^{'}, b^{'})\right\|^2 \\ &+ \lambda(\left\|W_{\frac{L}{2}}\right\|^2 + \left\|W^{'}_{\frac{L}{2}}\right\|^2)\end{aligned} \tag{3.5}$$

consists of:

- The rating loss. The approximated ratings are calculated as a dot product of the user and item embeddings taken from the bottleneck layer of the respective autoencoders. This layer is effectively the output of the encoder part named f_e^{user} for the user autoencoder and f_e^{item} for the item one. The embeddings U and V are therefore calculated as:

$$\begin{aligned}U_{i,\bullet} &= f_e^{user}(\tilde{R}_{i,\bullet}, W, b) \\ V_{j,\bullet} &= f_e^{item}(\tilde{R}_{\bullet,j}, W^{'}, b^{'})\end{aligned} \tag{3.6}$$

- The reconstruction loss. Both autoencoders attempt to reconstruct the original clean ratings R while being given their noisy versions $\tilde{R}$. The reconstructions are the outputs of the decoder parts named f_d^{user} and f_d^{item} for the user and item autoencoders respectively.

- The regularization loss. We regularize only the weight matrix preceding the bottleneck layer in both autoencoders since we observed worse performance while regularizing all of them.

3.5 Hybrid Additional Stacked Denoising Autoencoder

Hybrid Additional Stacked Denoising Autoencoder (Hybrid aSDAE) is a recent architecture proposed by Xin Dong *et al.* in 2017 [13]. It supports incorporating both user and item attributes into the final latent representation U and V. It is therefore a truly hybrid recommendation model combining both collaborative and content based filtering through the latent factor based matrix factorization.

The model architecture visualized in Figure 3.3 is similar to the previously explained Hybrid SDAE with the both SDAEs replaced by a novel autoencoder architecture called Additional SDAE.

3.5.1 Additional Stacked Denoising Autoencoder

Additional SDAE (aSDAE) expands the traditional SDAE architecture described in Section 2.3 with the support for another source of information. In the case of recommendation systems, this model is able to process both the ratings and attributes of items and users.

Since it is a denoising model, the input rating information R and attribute information A is corrupted before feeding it into the network as $\tilde{R}$ and $\tilde{A}$ respectively. The used corruption is either a binary masking noise typically used on the sparse ratings or additive Gaussian noise which is a good fit for real valued attributes.

The architecture of the autoencoder visible in the top and bottom part of Figure 3.3 consists of newly added layers combining the processed attribute information with the well known SDAE component which is at its heart.

The output h_l of each hidden layer $l \in \{1, \cdots, L-1\}$ is computed as:

$$h_l = f(W_l^r h_{l-1} + W_l^a \tilde{A} + b_l) \tag{3.7}$$

while h_0 is a batch of corrupted rating vectors from $\tilde{R}$ and f is the chosen activation function.

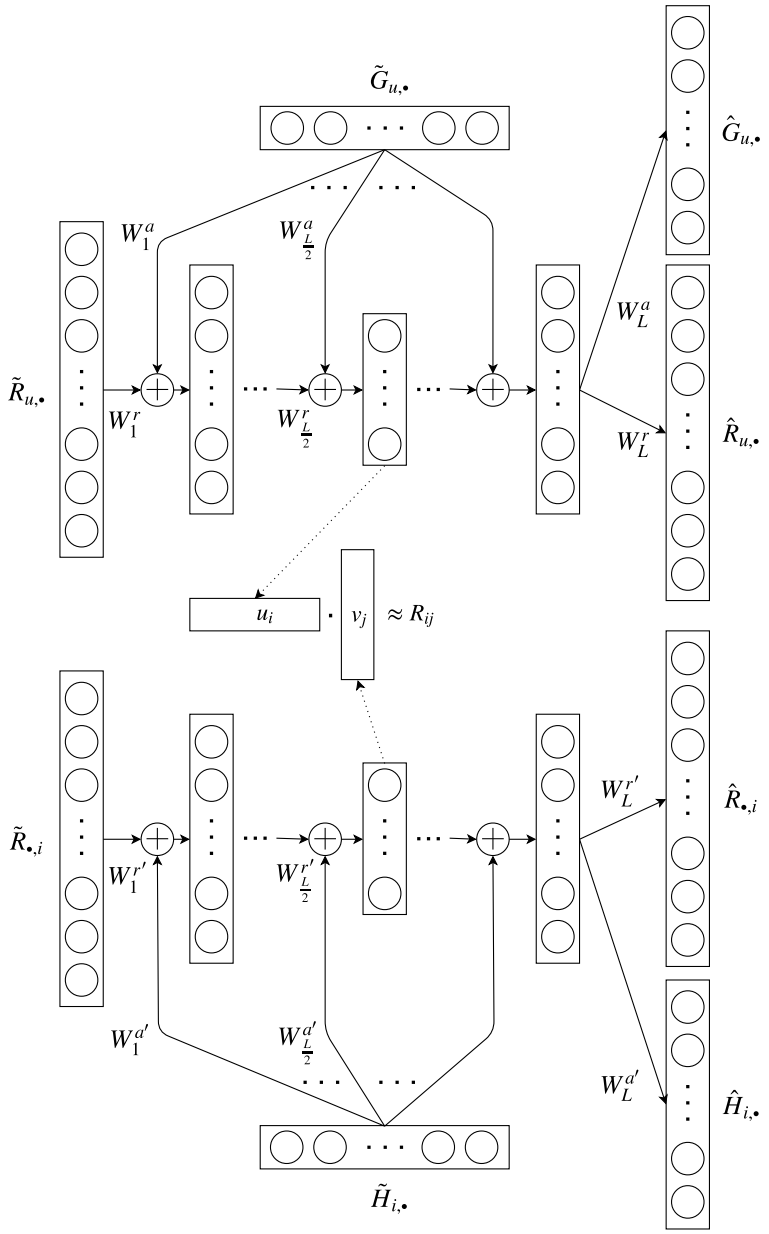


Figure 3.3: Hybrid Additional SDAE network architecture.

The outputs $\hat{R}$, $\hat{A}$ of the last layers attempt to reconstruct the original clean inputs, specifically the ratings R and attributes A as:

$$\begin{aligned} \hat{R} &= f(W_L^r h_L + b_L^r) \\ \hat{A} &= f(W_L^a h_L + b_L^a) \end{aligned} \tag{3.8}$$

As can be seen the name *additive* comes from the fact that the attribute vectors are added to the output of each hidden layer. The attribute vectors need to be multiplied by their own weight matrices W_l^a before being added to the output of the previous layer h_{l-1} multiplied by its special weights W_l^r. This need arises for two reasons:

- It allows the network to extract useful information from the noise corrupted attributes to be able to both reconstruct them from the last layer and find more complex patterns.
- To be able to actually add the attribute vectors to the layer outputs, their dimensions must match. The weight matrices W_l^a therefore have dimensions of $a_{dim} \times n_l$ where a_{dim} is the dimension of the attribute vector and n_l is the number of neurons in the l-th layer.

The first $\frac{L}{2}$ layers act as an encoder and we mark them as f_e. The output of this part is considered as an embedding of the input. The rest of the layers act as a decoder.

Training of the autoencoder follows the loss function specified in Equation 3.9. It consists of the distances between the clean input and the reconstructed one and weight regularization. The reconstruction losses are balanced by the α parameter allowing us to control on which part should the model focus more during training.

$$\begin{aligned}\mathcal{L} = \alpha \left\| R - \hat{R} \right\|^2 + (1-\alpha) \left\| A - \hat{A} \right\|^2 \\ + \lambda(\|W^r\|^2 + \|W^a\|^2)\end{aligned} \tag{3.9}$$

Being able to incorporate additional sources of information is undoubtedly beneficial however the fact that the aSDAE has two sets of weights (W^r and W^a) unfortunately leads to a significant increases in the number of trainable parameters and therefore the training time. This problem is especially noticeable in case of high dimensional attribute vectors.

3.5.2 Hybrid aSDAE

To create the final hybrid recommendation model, two aSDAEs are combined through the goal of factorizing the rating matrix R similarly to all the previously mentioned models.

Figure 3.3 clearly shows the user aSDAE at the top and the item one at the bottom connected by a dot product of their bottleneck layers. The outputs of the bottleneck layers of the user and item parts are called f_e^{user}

and f_e^{item} respectively because they represent the encoder parts of the autoencoders. These outputs form the latent representations stored in the U and V matrices as seen in the following equation:

$$\begin{aligned} U_{i,\bullet} = u_i &= f_e^{user}(\tilde{R}_{i,\bullet}, \tilde{G}_{i,\bullet}, W^r, W^a, b) \\ V_{j,\bullet} = v_j &= f_e^{item}(\tilde{R}_{\bullet,j}, \tilde{H}_{\bullet,j}, W^{r'}, W^{a'}, b') \end{aligned} \tag{3.10}$$

The dot product of the latent representations forms the approximated ratings $\hat{R}$. The loss function used for training the model is visible in Equation 3.12and consists of several terms:

- The sum of errors of the approximated ratings. However only selected ratings having 1 in the binary matrix I_{ij} are counted towards it. The authors use only the known ratings.
- The reconstruction losses of the user and item autoencoders. Parameter α_1 is used to balance the errors of ratings and attributes in case of the user AE while α_2 is used for the item AE. The reconstructions are calculated by the last layers as:

$$\begin{aligned} \hat{R}_{user} &= f(W_L^r h_L + b_L^r) \\ \hat{G} &= f(W_L^a h_L + b_L^a) \\ \hat{R}_{item} &= f(W_L^{r'} h_L^{'} + b_L^{r'}) \\ \hat{H} &= f(W_L^{a'} h_L^{'} + b_L^{a'}) \end{aligned} \tag{3.11}$$

- Last term is a regularization of all the weights and biases used in the network multiplied by the λ hyper parameter.

$$\begin{aligned} \mathcal{L} = &\sum_{i,j} I_{ij}(R_{ij} - u_i v_j^T)^2 \\ &+ \alpha_1 \left\| R - \hat{R}_{user} \right\|^2 + (1 - \alpha_1) \left\| G - \hat{G} \right\|^2 \\ &+ \alpha_2 \left\| R - \hat{R}_{item} \right\|^2 + (1 - \alpha_2) \left\| H - \hat{H} \right\|^2 \\ &+ \lambda(\|W^r\|^2 + \|W^a\|^2 + \|b\|^2 + \left\|W^{r'}\right\|^2 + \left\|W^{a'}\right\|^2 + \left\|b'\right\|^2) \end{aligned} \tag{3.12}$$

We did not implement this architecture because we believe it would reach similar performance to our proposed Hybrid cSDAE discussed in the next section while requiring more time to train.

3.6 Hybrid Concatenated Stacked Denoising Autoencoder

Hybrid Concatenated Stacked Denoising Autoencoder (Hybrid cSDAE) is a novel architecture inspired by the Hybrid aSDAE discussed in the previous section. The primary goal is to reduce the number of trainable parameters as much as possible while still supporting the attribute information and therefore keeping all the benefits of a hybrid recommendation model.

The architecture visible in Figure 3.4 is similar to the Hybrid aSDAE one with the Additional SDAEs replaced with our newly proposed Concatenated SDAE (cSDAE).

3.6.1 Concatenated Stacked Denoising Autoencoder

Concatenated Stacked Denoising Autoencoder (cSDAE) modifies the traditional SDAE by allowing multiple sources of information at the input and being able to reconstruct them at the end of the decoder.

We specifically want to avoid the duplication of each SDAE layer which aSDAE needs to be able to add the compressed attribute vector at every step to the output of the previous layer. To achieve that, we compress the attribute vector only one time at the beginning and concatenate the result with the compressed rating vector. This concatenated vector is then passed as an input to a traditional SDAE architecture. Its output is then used to reconstruct both of the inputs by a layer structure mirroring the one compressing the inputs.

The ratings are corrupted by a binary masking noise before being passed into the network as is expected, however we found that corrupting the attribute vectors did not improve the results and we therefore input them unchanged.

Both the network f^r_{comp} compressing the ratings R and the network f^a_{comp} compressing the attributes A are simple feed-forward architectures with decreasing number of neurons in each subsequent layer. They can be imagined as an extensions of the encoder part of the SDAE at the core. We compress the inputs to the same dimension before concatenating them. We did not test other ratios of the compressed dimensions that may be beneficial in certain cases.

After compression and concatenation, the resulting vector h_c is passed to the first layer of the SDAE encoder called X_{a_in}:

$$h_c = concat(f^r_{comp}(\tilde{R}_{i,\bullet}, W^r_{comp}, b^r_{comp}), f^a_{comp}(A_{i,\bullet}, W^a_{comp}, b^a_{comp})) \tag{3.13}$$

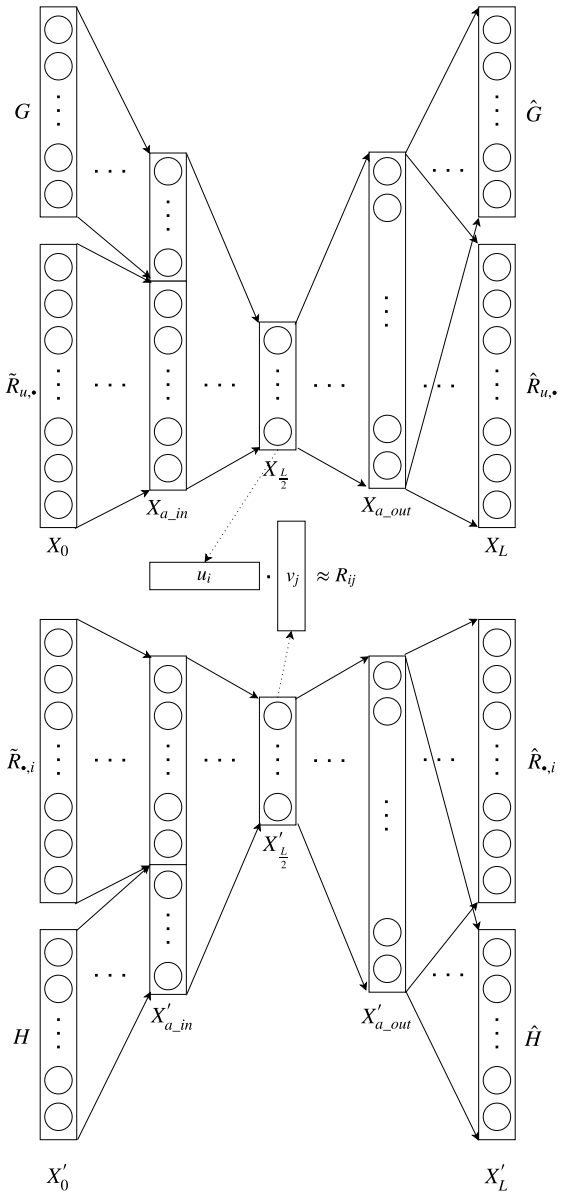


Figure 3.4: Hybrid Concatenated SDAE network architecture.

The encoder part f_{encode} of the SDAE further compresses the input and generates a latent representation $h_{\frac{L}{2}}$ as a result of the bottleneck layer $X_{\frac{L}{2}}$:

$$h_{\frac{L}{2}} = f_{encode}(h_c, W_{encode}, b_{encode}) \tag{3.14}$$

Then the decoder part f_{decode} produces a vector h_{a_out} with the same

dimensions as h_c as an output of the layer X_{a_out}:

$$h_{a\_out} = f_{decode}(h_{\frac{L}{2}}, W_{decode}, b_{decode}) \tag{3.15}$$

Finally, the reproductions $\hat{R}_{i,\bullet}$, $\hat{A}_{i,\bullet}$ of the original $R_{i,\bullet}$, $A_{i,\bullet}$ are produced by the decompressing feed-forward networks extending the decoder and mirroring the structure of the compression networks $f^{\bullet}_{comp}$:

$$\begin{aligned} \hat{R}_{i,\bullet} &= f^{r}_{decomp}(h_{a\_out}, W^{r}_{decomp}, b^{r}_{decomp}) \\ \hat{A}_{i,\bullet} &= f^{a}_{decomp}(h_{a\_out}, W^{a}_{decomp}, b^{a}_{decomp}) \end{aligned} \tag{3.16}$$

If we use one layer f^{a}_{comp} and set the output dimension of it equal to the dimension of the first hidden layer of aSDAE, we can see that:

$$dim(W^{a}_{1}) = dim(W^{a}_{L}) = dim(W^{a}_{comp}) = dim(W^{a}_{decomp}) \tag{3.17}$$

meaning the cSDAE architecture reduces the number of trainable parameters by not needing the aSDAE layers processing the attributes between the first and last layer of the SDAE. Specifically, using the aSDAE notation, the following number of parameters is removed:

$$\sum_{l=2}^{L-1} (size(W^{a}_{l}) + size(b^{a}_{l})) \tag{3.18}$$

Training follows the loss function in Equation 3.19 which is almost identical to the aSDAE loss. The only difference is that cSDAE regularizes only the weights $W_{\frac{L}{2}}$ that precede the last layer of the encoder.

$$\begin{aligned} \mathcal{L} = \alpha \left\| R - \hat{R} \right\|^2 + (1-\alpha) \left\| A - \hat{A} \right\|^2 \\ + \lambda(\left\| W_{\frac{L}{2}} \right\|^2) \end{aligned} \tag{3.19}$$

3.6.2 Hybrid cSDAE

Combining two cSDAE to create the hybrid recommendation model is done in exactly the same way as with Hybrid aSDAE.

The final loss function visible in Equation 3.20 is again similar to the one of Hybrid aSDAE. The latent representation matrices U, V are user, item variants of an encoder output $h_{\frac{L}{2}}$ seen in Equation 3.14. The reconstructions

$\hat{R}_{user}$, $\hat{G}$ and their item equivalents are calculated according to the Equation 3.16.

$$\begin{aligned}\mathcal{L} = &\left\|R - UV^T\right\|^2 \\ &+ \alpha_1 \left\|R - \hat{R}_{user}\right\|^2 + (1-\alpha_1)\left\|G - \hat{G}\right\|^2 \\ &+ \alpha_2 \left\|R - \hat{R}_{item}\right\|^2 + (1-\alpha_2)\left\|H - \hat{H}\right\|^2 \\ &+ \lambda(\left\|W^{user}_{\frac{L}{2}}\right\|^2 + \left\|W^{item}_{\frac{L}{2}}\right\|^2)\end{aligned} \tag{3.20}$$

The only major differences to the Hybrid aSDAE loss function is the first term. Calculating the loss only on the known ratings may lead the model to predict 1 for every rating. To prevent that, we inject some of the unknown ratings into the training batches and use them as known negative ratings. This whole process is described in detail in Section 4.3.

3.6.3 Architecture details

The specific Hybrid cSDAE architecture used in all our experiments has the following layer structure:

- The network f^r_{comp} compressing the ratings is a three layer feed-forward net. The second layer has 256 neurons and the third one 128.
- The network f^a_{comp} compressing the attributes is a two layer feed-forward net with the second layer having 128 neurons. This is an example of the flexibility of this architecture, since all the datasets we evaluate on have fairly simple attribute vectors, we can choose to use a smaller network to compress them. Smaller in comparison to the one compressing the ratings that generally have much larger dimensionality.
- The SDAE following the compression of inputs has three layers. The input one has 256 neurons due to the concatenation of the two 128 inputs. Following is the bottleneck layer with $k = 64$ neurons and then the decoder mirroring the layers of the encoder.
- The decoder output is passed onto the feed-forward decompressing networks f^r_{decomp} and f^a_{decomp} that reconstruct the inputs. Their structure mirrors the structure of the compressing networks.
- The hyperparameters α_1 and α_2 controlling the importance given to the reconstruction of the ratings compared to the reconstruction of

attributes are both set to 0.8 in our experiments unless stated otherwise.

As can be seen we have chosen the layer unit counts carefully to be as close as possible to the other evaluated architectures. We could easily expand both the compression nets and the core SDAE with additional hidden layers to allow the model to extract more complex patterns with little added computation cost. However we seek to compare the described models on fair grounds and therefore try to minimize the differences between them.

3.7 Collaborative Deep Learning

The Collaborative Deep Learning (CDL) as it is defined in the original publication [47] is a general framework that can incorporate several models. We will focus on a special case using classic SDAE described in Section 2.3 for the item attribute feature extraction. This case is discussed by the CDL authors as λ_s going to infinity. We are going to use the notation described in Table 3.3. Only notable changes to the paper are renaming a to simply 1 and b to α.

H	clean $m \times h$ matrix of all item attribute vectors
$\tilde{H}$	noisy $m \times h$ matrix of all item attribute vectors
W^+	weight matrices of all layers
W_l	weight matrix of layer l
b_l	bias vector of layer l
$f_e(\text{input}, \text{weights})$	encoder function representing the first half of the autoencoder
$f_d(\text{input}, \text{weights})$	decoder function representing the whole autoencoder
C	$n \times m$ matrix with the weights of the ratings
1	weight of known ratings
$\alpha < 1$	weight of unknown ratings

Table 3.3: CDL specific notation.

As can be seen in the Figure 3.5 CDL is quite different from the previously described models. It can be understood as a combination of an autoencoder and matrix factorization. Only item attributes are supported, not user ones which is one of the disadvantages of CDL. Although the architecture could be extended to be symmetric, the resulting model would have even more complicated evaluation of new users. Nevertheless it is definitely possible and an interesting direction of future work.

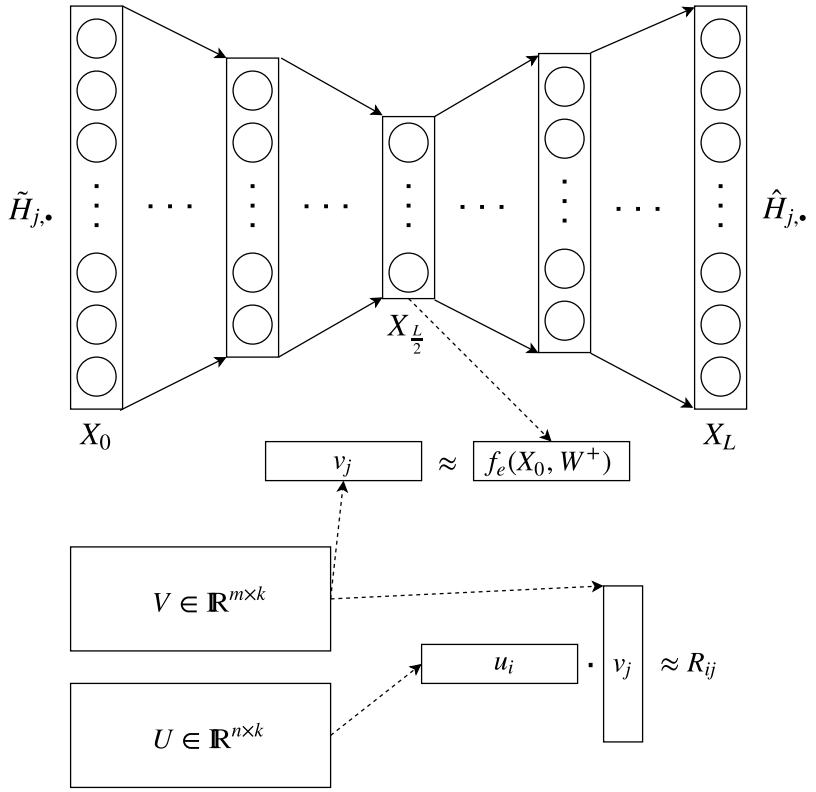


Figure 3.5: CDL network architecture.

The single autoencoder is trained to create compressed vector representations of the item attributes. These representations are continuously used to optimize the actual item embeddings that are also connected to the user embeddings through approximation of the train ratings. This approximation is done by a standard dot product of the embeddings and is compared to the known ratings. The ratings are used only to calculate the error of the approximation, they are not passed as an input.

The loss function depicted in Equation 3.21 connects all the described parts of CDL and consists of the following terms:

- $L2$ regularization of the user embeddings controlled by the λ_u hyperparameter. Note that there is no explicit regularization of the item embeddings. They are indirectly regularized by the other terms.

- $L2$ regularization of all the weights and biases of the autoencoder controlled by the λ_w hyperparameter.

- $L2$ distance of the item embedding to the latent representation of item attributes outputted by the encoder f_e. The distance is multiplied by the λ_v hyperparameter.

- Reconstruction loss of the autoencoder multiplied by the λ_n hyperparameter.

- Distance of the approximated ratings to the original ones weighted by the C matrix. The C matrix is 1 for the known ratings and α for the unknown ones.

$$\begin{aligned}\mathcal{L} = &\frac{\lambda_u}{2}\sum_i \|u_i\|^2 \\ &+ \frac{\lambda_w}{2}\sum_l (\|W_l\|^2 + \|b_l\|^2) \\ &+ \frac{\lambda_v}{2}\sum_j \left\|v_j - f_e(\tilde{H}_{j,\bullet}, W^+)^T\right\|^2 \\ &+ \frac{\lambda_n}{2}\sum_j \left\|f_d(\tilde{H}_{j,\bullet}, W^+) - H_{j,\bullet}\right\|^2 \\ &+ \sum_{i,j}\frac{C_{ij}}{2}(R_{ij} - u_i^T v_j)^2\end{aligned} \tag{3.21}$$

All the new lambda hyperparameters including the α need to be carefully tuned to ensure that one of the loss terms does not overshadow the remaining ones. Since there are five of them, we have decided to find the best combination of values for each of our dataset manually to avoid a very expensive grid search.

3.7.1 Training

The training consists of optimization of the autoencoder that tries to reconstruct the noisy item attributes and two floating point matrices representing all the item and user embeddings respectively.

We feed just a batch of item indexes, their ratings for the loss function and noise ratio to corrupt the item attribute vectors. The noise masks only hide some ones in the binary attribute vectors that we use. The fact that we do not input the rating vectors into any neural network is a significant advantage compared to the other discussed models that might have problems with scaling in case of large number of users and items.

The gradients of the loss function in Equation 3.21 are calculated by backpropagation. We take turns in optimizing the autoencoder part and the embeddings part of the network. One training step updates embeddings of items from the batch and all the user embeddings. The optimization of all the user embeddings with each batch of items necessarily comes out of the choice to approximate the whole item rating vectors that are compared to the known ones in the loss function. This is unsurprisingly quite costly and in our experience completely negates the benefit of not using a neural net for processing the rating vectors.

Since we approximate the whole item rating vector, we need to weight the error of the unknown ratings that we represent as zeroes. If we did not weight them we would end up with a model predicting zero for every rating and having a very small loss. We therefore use the alpha parameter to multiply the squared error corresponding to the unknown ratings.

Both the user and the item embedding matrices are randomly initialized at the start of training. The item embedding vectors are optimized to minimize two constraints:

- Their $L2$ distance from the autoencoder's latent representation of the item attribute information.
- Square of the weighted distance of its dot product with the user embedding from the known rating.

The user embeddings are optimized only to minimize the distance of the dot product from the known rating. Which makes them much more free to change.

All the other models presented in this chapter use a different training regime mainly due to the fact that calculating all the user embeddings during each update would be too expensive since it requires a neural network inference for each user. The same applies to the item embeddings of course because all the other architectures are symmetric. As a result of this constraint we had to choose a different approach to the loss and to preventing the model from approximating every rating as unknown. It is described in Section 4.3.

3.7.2 Evaluation

Since CDL in its original form presented in [47] does not support evaluation on new users, we had to devise a method to use the trained model to

recommend items for validation and test users that the model never saw during training. This section uses notation specified in Table 3.4.

V	$m \times k$ matrix of all item embeddings
λ_u	user embedding regularization strength
u	index of user whose embedding we want to calculate
w	desired embedding of user u
Y	$m \times 1$ column rating vector of user u
s	items implicitly rated by the user u
m_s	number of items in s
V_s	$m_s \times k$ matrix of embeddings of items rated by u
$\mathbb{1}$	$m_s \times 1$ column vector of ones
$\mathbb{I}_k$	$k \times k$ identity matrix

Table 3.4: Notation specific to the CDL evaluation equations.

The problem is that in case of CDL, the user embeddings are not received from a certain layer in a neural network but are all optimized separately. As a result we cannot simply get an embedding for a new user u by just inputting his rating information. We however have all the item embeddings calculated and we can therefore use them to simulate one step of the CDL training algorithm to get the embedding of user u. The goal is to minimize the distance of the dot product of the user embedding and the item embeddings to the corresponding ratings of the user u.

We derive the exact solution for our rating matrix consisting only of ones for positive ratings and zeroes for either unknown or negative ratings. We need to minimize the following loss function with regards to w which is the embedding of some validation user u:

$$\mathcal{L}(w) = \alpha \left\|Y - Vw\right\|^2 + (1-\alpha) \left\|\mathbb{1} - V_s w\right\|^2 + \lambda_u \left\|w\right\|^2 \tag{3.22}$$

Solution for w starts by calculating the gradient of $L(w)$ and setting it to 0:

$$\begin{aligned} \nabla\mathcal{L}(w) &= -2\alpha V^T(Y - Vw) - 2(1-\alpha)V_s^T(\mathbb{1} - V_s w) + 2\lambda_u w \\ &= \alpha V^T Y - \alpha V^T V w + (1-\alpha)V_s^T \mathbb{1} - (1-\alpha)V_s^T V_s w - \lambda_u w \\ &= 0 \end{aligned} \tag{3.23}$$

Then we just simplify the linear system of equations to a typical form

of $Aw = b$ that can be solved by standard library functions:

$$\begin{aligned}
&\alpha V^T V w + (1-\alpha) V_s^T V_s w + \lambda_u w = \alpha V^T Y + (1-\alpha) V_s^T \mathbb{1} \\
&(\alpha V^T V + (1-\alpha) V_s^T V_s + \lambda_u \mathbb{I}_k) w = \alpha V^T Y + (1-\alpha) V_s^T \mathbb{1} \\
&(\alpha V^T V + (1-\alpha) V_s^T V_s + \lambda_u \mathbb{I}_k) w = \alpha V_s^T \mathbb{1} + (1-\alpha) V_s^T \mathbb{1} \\
&(\alpha V^T V + (1-\alpha) V_s^T V_s + \lambda_u \mathbb{I}_k) w = V_s^T \mathbb{1}
\end{aligned} \tag{3.24}$$

After getting the validation user embedding we approximate his unknown ratings by a dot product of his embedding with the corresponding item embeddings. Recommendations are then standardly selected as the top K items from the sorted list of approximated ratings.

Chapter 4

Experiments

4.1 Implementation

We implemented all the models and experiments in Python 3.5. The following libraries were used:

- **TensorFlow 1.11** to implement all the models [49].
- **Numpy** for fast matrix operations during evaluation [50].
- **Pandas** for efficient loading and preprocessing of the training datasets [51].
- **scikit-learn** for simple transformation of numerical values to one-hot encoding [52].

4.2 Datasets

We use the following datasets for evaluation: MovieLens 100k, MovieLens 1m and BookCrossing. The number of publicly available datasets that satisfy our requirements of having both user and item attribute information is quite low, which is the reason why we did not evaluate our methods for example on the larger variants of the MovieLens.

We only consider users and items with the attribute information filled out. In case of MovieLens we keep users and items with at least two ratings. The same applies to the items at BookCrossing but the users have to have at least 20 ratings.

Preprocessing specific to the MovieLens datasets:

- **User attributes:** Gender, age, occupation and zip code. We one-hot encode all of them while using only the first 3 symbols of the zip code. The resulting user attribute vector is a concatenation of these one-hot encodings.
- **Item attributes:** The item attribute vector for both MovieLens datasets is just the one-hot encoding of the respective item genres. This means that dimension of the item attributes is much smaller than the user ones.

Preprocessing specific to the BookCrossing dataset:

- **User attributes:** Concatenation of one-hot encoded age, country and a region. We extract the country and region from the provided Location column.
- **Item attributes:** We again one-hot encode and concatenate the publisher and the publication year, all the other provided features have too high variance.

We decided to use implicit binary ratings. That means we represent all the unknown and negative ratings as 0 while assigning 1 to the positive ones. Since all chosen datasets have explicit ratings we transform them by considering all ratings smaller than a certain cut-off point as negative and the rest as positive. The cut-off point is 4 for MovieLens and 5 for BookCrossing.

	MovieLens 100k	MovieLens 1m	BookCrossing
# users	942	6037	3153
# items	1447	3532	104749
# implicit ratings	55375	573582	207669
portion of all ratings	4.0625%	2.6900%	0.0629%
avg ratings per user	58.8	95.0	65.8
avg ratings per item	38.2	162.4	1.9
user attributes dim	415	707	439
item attributes dim	19	18	8709

Table 4.1: Dataset statistics after preprocessing.

The basic statistics of the datasets after preprocessing are shown in the Table 4.1. It can be clearly seen that the two MovieLens datasets have roughly equal numbers of users and items whereas the BookCrossing one has more than thirty times more items than users. It is also much sparser with only 1.9 ratings per item on average.

4.3 Model training

The training follows similar rules for all the mentioned architectures except for the Collaborative Deep Learning which uses a special procedure allowed by the fact that it does not have a neural network calculating the user embeddings. The training of CDL is described in detail in Section 3.7 while the training procedure applicable to the rest of the architectures is described here.

We initialize all layers in all the neural networks by *Glorot uniform initializer* also called *Xavier uniform initializer* introduced by Xavier Glorot and Joshua Bengio in [53] and implemented in TensorFlow.

The actual training is done by the Adam algorithm described in Section 2.2.1. We tune the learning rate for each model and dataset combination while keeping the other parameters at default values of: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

We use 80% of all the users for training, the rest is split evenly among validation and test sets. The whole evaluation process is described in detail in the Section 4.4.

We have already covered what do we pass into the models during training and how we use a binary masking noise to hide some of the ratings from the input rating vectors while describing the architectures in Chapter 3. All the experiments were conducted with noise ratio 0.2 meaning that 20% of the known ratings were hidden in the input rating vectors. What remains is the description of how exactly are the batches created which can have a significant impact on the performance of the trained model.

Since we use implicit binary ratings we cannot simply shuffle them and pass them to the model, it would only learn to predict 1 for all the inputs. On the other hand, if we considered all the unknown ratings as known negative ones and used them with the known positive ones during training, the model would probably learn to constantly predict a negative rating due to their overwhelming majority out of all the ratings. There are two possible ways to alleviate this problem:

- Feed all the ratings including the unknown ones that are 0 and give more importance to the prediction of the known ratings. This approach makes sense if we predict multiple ratings at once, typically whole rating vectors. A prime example is the CDL model which predicts whole item rating vectors at each training step and multiplies the errors by a weight matrix.

- Sample some unknown ratings and present them as known negative ones to the model. That means the approximation of both 1s and 0s is given the same importance. The different weights of the errors are replaced by a specific ratio of the sampled unknown and known ratings in each training batch.

All the models except the CDL predict the ratings one by one and we therefore choose the second option with balanced batches. We use a parameter γ called **negative sampling rate** to describe the composition of the batches. Each batch of size N consists of $\frac{N\gamma}{\gamma+1}$ sampled unknown ratings and $\frac{N}{\gamma+1}$ known ratings. This translates to γ unknown ratings per one known rating. We use $\gamma = 4$ as default value which has been found to be a reasonable choice according to the experiments of He *et al.* in [54].

To create these batches for one epoch, we first shuffle the known ratings of the training users and cycle through them in $\frac{N}{\gamma+1}$ chunks. The rest of each batch is always filled by random unknown ratings that are sampled by cycling through the training users and through all the items. These are both shuffled each time the iterator reaches the end.

The same issue with learning to predict a constant value arises at the reconstruction end of the autoencoders. We therefore calculate the loss only on selected reconstructed ratings. The selection includes all known ratings and then γ times more randomly sampled unknown ratings. This masking approach is more computationally efficient than multiplying the loss by a weight matrix and produces good results.

4.4 Model evaluation

We first split the preprocessed dataset per users into train, validation and test sets using 0.8, 0.1, 0.1 ratios respectively. We then split the validation and test user ratings to train ratings used as an input to the trained model and test ratings used for evaluation of model's performance with ratios of 0.8 and 0.2 respectively. The whole process is in simplified form shown as a diagram in Figure 4.1.

We measure the following metrics: RMSE, Recall, Test Recall and NDCG. Because it is infeasible to calculate the predicted value for all possible ratings we uniformly sample 1000 items for each evaluated user that he did not rated. We then calculate the model predictions on these sampled items together with the rated ones. Detail descriptions of our implementation of the used metrics follows:

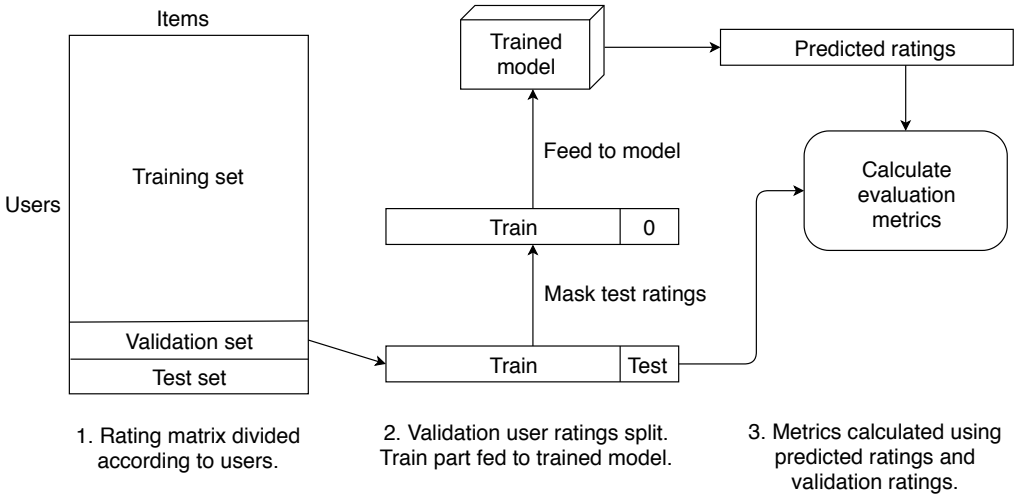


Figure 4.1: Calculation of evaluation metrics on the validation set. Exactly the same process is applied to the test set. The metrics are calculated each 1000 steps during training to be able to find the best score for each metric.

- **RMSE:** The error is calculated only on the withheld known ratings which the model never encountered. They are all 1 due to the fact that we use implicit ratings. As a result, a model predicting only ones would have a zero RMSE. It also means that the RMSE could be changed by simply scaling the outputs. For example if all the predictions were around 0.5 and we added 0.1 to all of them, the RMSE would get lower but the ordering of the items based on their predicted ratings would stay exactly the same. This metric is therefore not very useful for comparing the performances of different models since they can just be inherently biased towards returning lower or higher predictions. It can however be used to visualize the impact of changing a specific hyperparameter in case of a single model.

- **Classic Recall@K:** Adaptation of the previously described Recall metric specified in Algorithm 4. To calculate it for a single user we take all of his ratings including the ones used as an input to the model, combine them with the 1000 sampled items and calculate the predicted rating values for them. We then select the top K of these items sorted according to the predictions and count the number of positively rated ones among them. After normalizing this number we get the desired Recall@K for a single user. To get the overall Recall@K we simply average them over all evaluated users. This metric is used by the authors of the Hybrid aSDAE architecture and we include it

to show how it can produce misleading results compared to the Test Recall[13].

The problem comes from the fact that we count the positively rated items provided to the network at input as successful "hits" if they are in the top K recommended items. This means that the network can reach high recall by remembering and consequently reproducing the input ratings. Which is not trivial since it did not see the validation user ratings during training nevertheless it does not properly represent the model's capabilities to predict unknown ratings.

Algorithm 4: Classic Recall@K for user u

input : *all_pos_itemids*: all positive ratings of u,
sampled: 1000 randomly sampled items not rated by u
output: $recall_u$: Classic Recall@K for user u

$itemids \leftarrow all\_pos\_itemids \cup sampled$
$sorted\_itemids \leftarrow itemids$ descend. sorted acc. to predicted ratings
$top\_k\_itemids \leftarrow sorted\_itemids$[:K]
$num\_pos\_top\_k \leftarrow |all\_pos\_itemids \cap top\_k\_itemids|$
$max\_num\_pos\_top\_k \leftarrow minimum(K, |all\_pos\_itemids|)$
$recall_u \leftarrow \frac{num\_pos\_top\_k}{max\_num\_pos\_top\_k}$

- **Test Recall@K:** Is very similar to the Classic Recall@K metric. The key difference is that we do not calculate the predicted rating values for the ratings that were passed into the model as an input. We do not include them in the top k items at all. We basically only look at the randomly sampled items that were not rated and the withheld ones never shown to the model. The exact description is in Algorithm 5. Compared to the Classic Recall, this metric better focuses at the desired task of predicting the unknown ratings of new incoming users.

Algorithm 5: Test Recall@K for user u

input : *test_pos_itemids*: withheld positive ratings of u,
sampled: 1000 randomly sampled items not rated by u
output: $recall_u$: Test Recall@K for user u

$itemids \leftarrow test\_pos\_itemids \cup sampled$
$sorted\_itemids \leftarrow itemids$ descend. sorted acc. to predicted ratings
$top\_k\_itemids \leftarrow sorted\_itemids$[:K]
$num\_pos\_top\_k \leftarrow |test\_pos\_itemids \cap top\_k\_itemids|$
$max\_num\_pos\_top\_k \leftarrow minimum(K, |test\_pos\_itemids|)$
$recall_u \leftarrow \frac{num\_pos\_top\_k}{max\_num\_pos\_top\_k}$

- **NDCG@K:** The creation of the top k list of items and all the associated variables is identical to the way Test Recall does it. We then calculate IDCG_K according to the simplified equation 2.11 due to using the implicit ratings. The DCG_K is a sum of the simplified term over the top k items that were positively rated by the user. The rest of the items have relevance equal to zero. The Algorithm 6 describes the calculations using the variables from Algorithm 5.

Algorithm 6: NDCG@K for user u

input : *top_k_itemids*: calculated as in Algorithm 5,
test_pos_itemids: calculated as in Algorithm 5,
max_num_pos_top_k: calculated as in Algorithm 5

output: $NDCG_K$: NDCG@K for user u

$IDCG_K \leftarrow \sum_{i=1}^{max\_num\_pos\_top\_k} \frac{1}{\log_2(i+1)}$
$DCG_K \leftarrow 0$
for $i \leftarrow 1$ **to** K **do**
  if $top\_k\_itemids[i] \in test\_pos\_itemids$ **then**
    $DCG_K \leftarrow DCG_K + \frac{1}{\log_2(i+1)}$
$NDCG_K \leftarrow \frac{DCG_K}{IDCG_K}$

- **Coverage@K:** The coverage represents the ratio of the recommended items to the number of all the items. The recommended items are counted as *top_k_itemids* in Algorithm 5. That means we do not count the items with known ratings passed as an input to the model.

We calculate all the metrics based on K for $K \in (10, 50, 100, 150, 200, 250, 300)$ and then select the best result for each K. In other words we select the training step with the best validation score for each combination of model, dataset, metric, K and report the test score of the model at this training step.

In order to eliminate as much randomness as possible in the evaluation, we use 5-fold cross-validation during every run of the hyper-parameter search and all the other experiments. We implement it by cyclically choosing the 20% of users used for evaluation. The first half of them is marked as validation while the second one as test set. The rest of the users is used for the training.

The cross-validation allows us to report the mean and standard deviation of each of the hyperparametrization runs.

4.5 Evaluation results

First we comprehensively evaluate all the implemented models described in Chapter 3 by conducting a grid search over their hyperparameters and evaluating each of the parametrizations by the method explained in the Section 4.4. This generates enough data to compare the different architectures among each other.

The following experiments focus on the effects of a single hyperparameter on the performance of the proposed Hybrid cSDAE architecture. We investigate the roles of the negative sampling rate and the noise ratio.

4.5.1 Grid search results

In order to compare the different models fairly we tried to find the best hyperparameters for them on each dataset. The search was done by training the models on all the combinations of the parameters in Table 4.2. Even though the grid search does not appear very extensive it still took over 300 hours of computation time on a server with a single GTX 1080. All the models were trained for 6 epochs and evaluated every 1000 steps. One step equals to processing one batch.

parameter	**tested values**
λ	$\{1 \times 10^{-3}, 1 \times 10^{-4}, 1 \times 10^{-5}\}$ for all models except CDL
batch size	$\{32, 64, 128\}$ on MovieLens 100k $\{128, 256, 512\}$ for the other two datasets
learning rate	$\{4 \times 10^{-3}, 4 \times 10^{-4}\}$

Table 4.2: Tested combinations of the hyperparameters.

Due to the large amount of regularization parameters of the CDL, we fine tuned them by hand. This resulted in: $\alpha = 0.1$, $\lambda_w = 0.001$, $\lambda_u = 0.01$, $\lambda_v = 100$, $\lambda_n = 20$ for the MovieLens 100k dataset and $\alpha = 0.1$, $\lambda_w = 0.001$, $\lambda_u = 0.01$, $\lambda_v = 10$, $\lambda_n = 2$ for the MovieLens 1m and the BookCrossing datasets.

The results are visualized in Figures 4.2, 4.3 and 4.4 that compare the implemented models on the MovieLens 100k, MovieLens 1m and BookCrossing datasets respectively.

Each figure consists of two rows. The first row contains graphs comparing the models using the Classic Recall, Test Recall and NDCG metrics that were explained in Section 4.4. Each of the points in these graphs belongs

to a certain dataset, metric and model. They are calculated by finding a training step with the best metric value on the validation set. This is done for each of the cross-validation iterations. The parametrization with the best average validation score (over the cross-validation iterations) is chosen and the test values from the previously selected training steps are put together. Each point on the graph then represents the mean of these test values with the errorbars signifying the standard deviation obtained thanks to the cross-validation.

The second row of graphs visualizes the coverage of the model on the test set. The points are again the means of the test values, however this time we take them from the parametrizations and training steps selected by a different metric. Concretely the metric having the graph at the same position but in the first row. This means that the points on graphs in the same column and belonging to the same K are a result of the exact same model. We can therefore see the coverage of a model selected for its high Recall or NDCG.

It would not make sense to show a coverage graph created the same way as the ones in the first row. The reason is due to selecting the best training step combined with the fact that a random model trivially achieves a maximum coverage. This would lead to showing coverage values obtained right after the random initialization of the network that are not interesting at all. The shown graphs are useful because we are especially interested in models that achieve both high recall and high coverage. Such models are able to recommend a broad set items that are relevant to different groups of users which is more desirable than a model focusing on a small set of popular items.

Firstly we examine the performance of the models on the smallest dataset: MovieLens 100k. The graphs with results are in the Figure 4.2. The low number of users leads to a high variance of the measured metrics. Both of these factors result in all of the models achieving similar performance in both types of recall and NDCG. The differences are however clearly seen in the coverage scores and they all follow a similar pattern. CDL has the worst coverage, Hybrid SDAE with Hybrid cSDAE outperform the rest and the MF models are in between.

It is important to remember that we are looking at the coverage scores of models with the highest recall or NDCG. In the case of this dataset, since all the models have reached similar scores of both recall and NDCG, we can see that the autoencoder based models would produce more diverse recommendations while keeping the reached recall which is typically desirable.

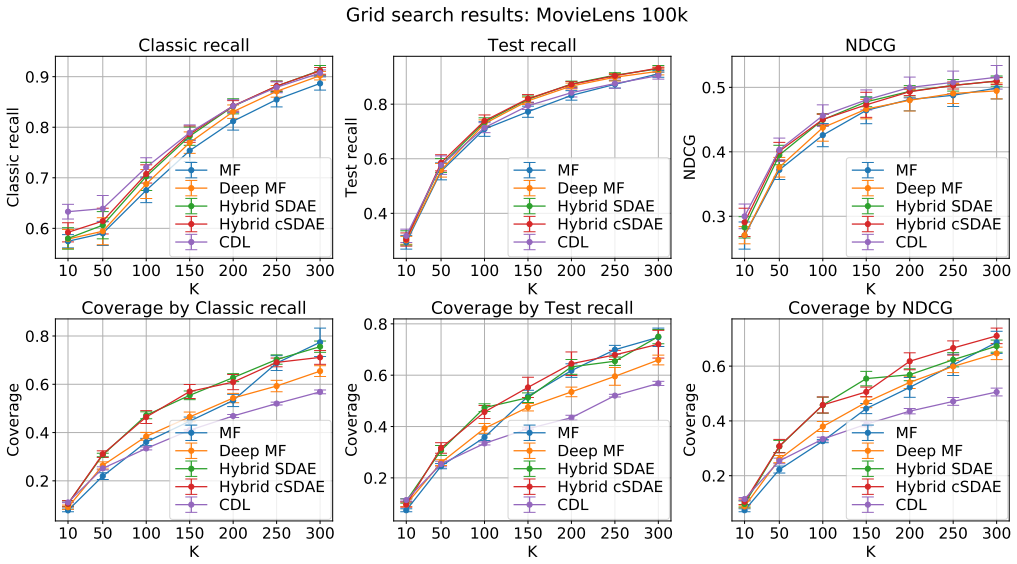


Figure 4.2: Model results on the MovieLens 100k dataset. Autoencoder based models achieve higher coverage while having similar recall and NDCG to the other models.

The model results on the bigger MovieLens 1m dataset are visualized in the Figure 4.3. The first thing that catches attention is the drop in Classic recall from $K = 10$ to $K = 50$. To understand why it is happening we have to delve into the exact computation of the metric for a certain user. It firstly counts the number of positively rated items in the top K and then divides this number by a minimum of K and the number of items positively rated by the user.

The drop comes from a combination of two factors: the fact that the metric counts the items passed as an input among the top K recommended items and the fact that if majority of users have significantly more positive ratings than K, it is then possible to reach recall of 1 by putting only some input items into the top K while ranking the other positively rated items very badly. This is exactly what happens here because the vast majority of users have more than 20 implicit ratings. We use 80% of them as input which means the model just has to rank 10 out of 16 items in the top K to achieve recall of 1. At the $K = 50$ the recall drops because the denominator increases and reveals that the other positively rated items are not ranked high enough to be counted in the top K.

However we are of course interested in how well the model generalizes to the unseen items not how well can it reproduce the inputted ratings. That is why we are going to focus on the Test recall instead of the Classic

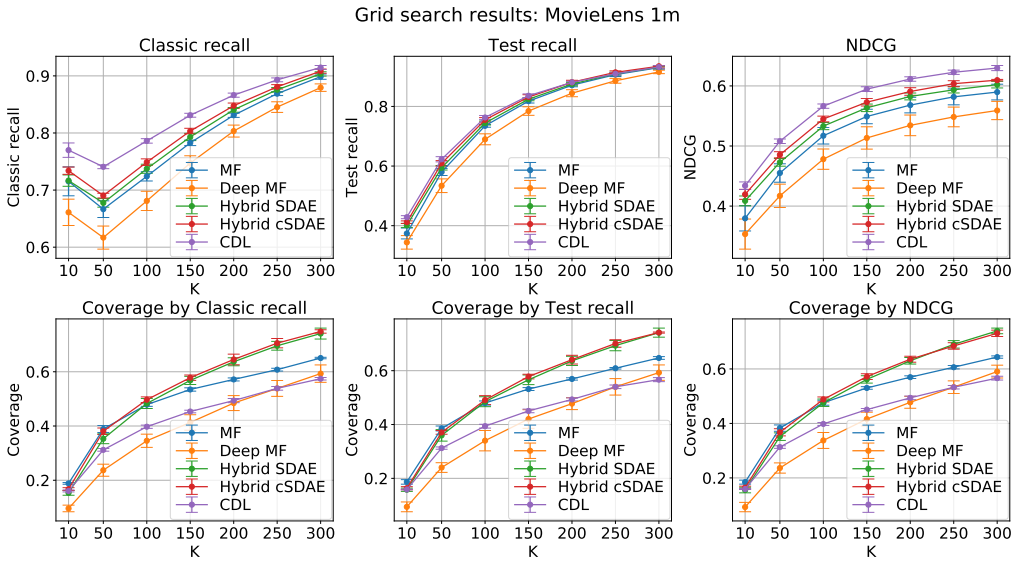


Figure 4.3: Model results on the MovieLens 1m dataset. CDL achieves the highest NDCG of all the models but simultaneously has one of the lowest coverage scores. The proposed Hybrid cSDAE has both high NDCG and coverage. The variant without attributes is close with a slightly worse NDCG. The Deep MF surprisingly has the worst results across all the measured metrics.

one. According to the Classic recall, the CDL clearly outperforms the other models, if we look at the Test recall however, we can see that they all reach similar results pointing to a similar level of generalization. This is where the NDCG comes in to clearly show the separation between the models. Just as a reminder, the NDCG calculates the top K items the same way as the Test recall but the resulting score takes into account the position of the relevant items in the top K meaning that the score gets higher if the relevant items are higher. On the other hand the recall only counts the number of items in the top K no matter their position.

Looking at the NDCG results, we can see that the CDL is at the top, followed by the our proposed Hybrid cSDAE which is close to the Hybrid SDAE. If we look at the coverage scores for these models we can see a similar story as with the MovieLens 100k dataset. The autoencoder based models have significantly better coverage than the CDL. This is the case where no single model dominates the others in all the relevant metrics and choosing the best one would depend on the domain specific needs.

There is one more surprising thing evident in the results. The Deep MF model has consistently the worst performance of all the models which

is strange because it should theoretically be able to emulate the simpler MF model. The reason why it is not able to do so probably lies in the hyperparameter search that did not contain a hyperparametrization allowing the Deep MF model to properly train. This just shows how non-trivial is the neural network training process. Even a relatively simple change in the architecture consisting of adding 2 feed-forward layers can demand a completely different set of hyperparameters to train.

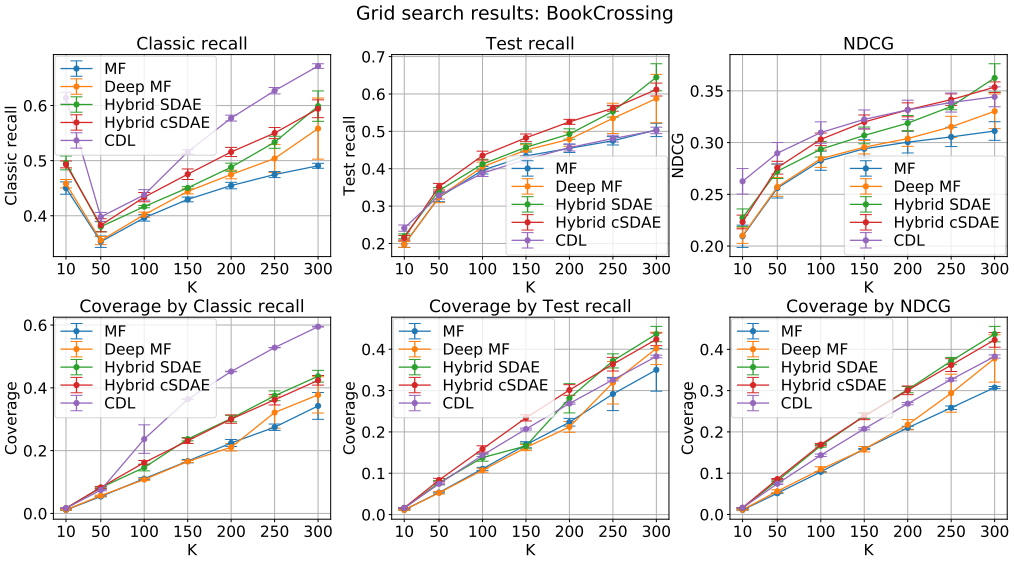


Figure 4.4: Model results on the BookCrossing dataset. Hybrid cSDAE and Hybrid SDAE achieve very good results in Test recall, NDCG and coverage. CDL is able to reach comparable NDCG scores but at the cost of slightly worse coverage.

The BookCrossing dataset is extremely sparse with a low number of users and large number of items. It averages only 1.9 implicit ratings per item which makes it very different from the previous MovieLens datasets. The model results visualized in the Figure 4.4 also show a different story than before.

The Classic recall performance puts the CDL at the top again however it clearly has the worst Test recall at par with the simple MF model. This even more solidifies the case of Classic recall producing misleading results in terms of model's generalization capabilities. Our proposed Hybrid cSDAE reaches the highest Test recall and simultaneously the highest coverage. In both cases it is closely followed by the Hybrid SDAE and this time also by the Deep MF which has therefore achieved much better results on the

sparser BookCrossing dataset than on the MovieLens 1m. This shows that the architectures and hyperparametrizations successful at one dataset are not directly transferable to another one which is logical since both of them have a direct impact on what is the model capable of learning.

On the other hand if we look at the model instances reaching the highest NDCG, we can see the Hybrid cSDAE, Hybrid SDAE and CDL trading places at the top depending on the chosen K. This nicely shows that there is rarely a single model dominating all the chosen metrics.

All in all the results have shown that the proposed Hybrid cSDAE architecture is consistently among the best evaluated models when looking at the Test recall, NDCG and coverage metrics. In all the results it has been closely followed or tied by the Hybrid SDAE that does not use the attribute information. This shows that the addition of the attributes either directly improved the achieved results or it led to an increased model capacity allowing the model to store more information. No matter which version is true in the case of the evaluation on these datasets, Hybrid cSDAE is definitely able to produce user and item embeddings containing the added information and approximating the ratings when multiplied. We can see the rating approximation capabilities in the high values of both recall and NDCG. We also know that the produced embeddings contain the rating and the attribute information because the model had no problem achieving low reconstruction errors during training. The analysis of the properties of the embeddings is unfortunately out of scope of this work and is therefore left for future research.

4.5.2 Negative sampling rate experiment

All the previous runs were made with a negative sampling rate (γ) equal to 4. That means each training batch consisted of 4 sampled unknown ratings per one known positive rating. To explore the effect this parameter has on the performance of the proposed Hybrid cSDAE architecture, we evaluate it on the BookCrossing and the MovieLens 1m datasets with $\gamma = \{1, 2, 4, 8, 16\}$. Each of the parameter setting is evaluated using cross-validation as in the previous experiments. The values of the other hyperparameters can be seen in Table 4.3.

The hyperparameters were selected based on the Test recall and NDCG results obtained through the grid search detailed in the previous section. We have decided to evaluate the model on the MovieLens 1m and the BookCrossing dataset due to their higher number of ratings and differ-

parameter	value used in the experiment
λ	1×10^{-5}
batch size	128
learning rate	4×10^{-3}
number of epochs	6

Table 4.3: Hyperparameter values of Hybrid cSDAE during the negative sampling rate experiment.

ent structures. The results for the MovieLens 1m dataset are visible in the Figure 4.5 and the ones for the BookCrossing are in the Figure 4.6.

The Figure's top and bottom rows are produced the same way as the ones in the Section 4.5.1 examining the results of the hyperparameter grid search. The only difference is that instead of comparing different models we are comparing different hyperparametrizations of the Hybrid cSDAE. It therefore makes sense to show the best obtained RMSE scores since we are looking at the results of a single model. We exclude the Classic recall from the Figures due to its misleading nature discussed in the previous sections.

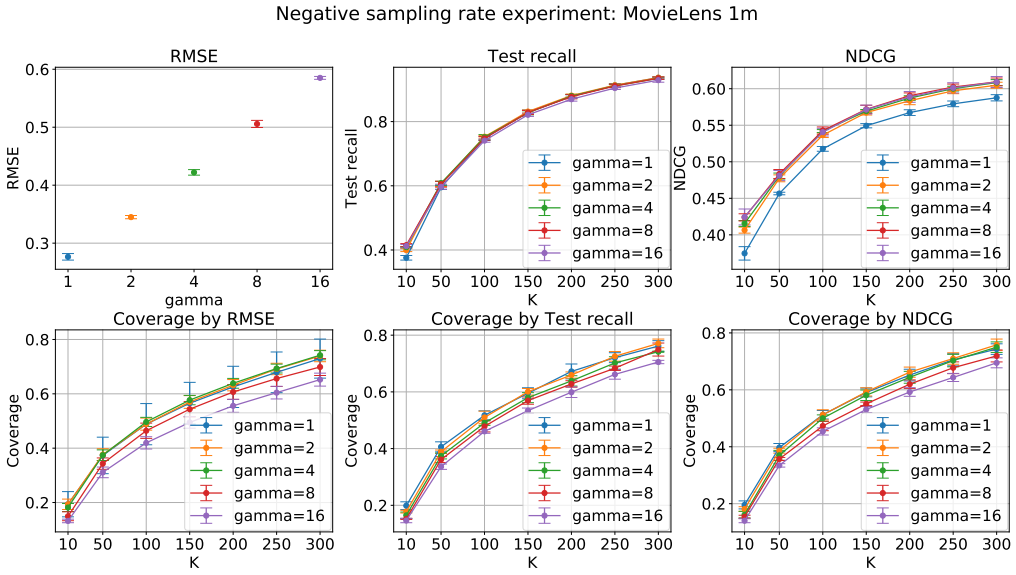


Figure 4.5: Effect of the negative sampling rate on the performance of the Hybrid cSDAE trained using MovieLens 1m.

Looking at the results on both datasets we can clearly see the lowest reached RMSE rising with the increasing negative sampling rate. This is

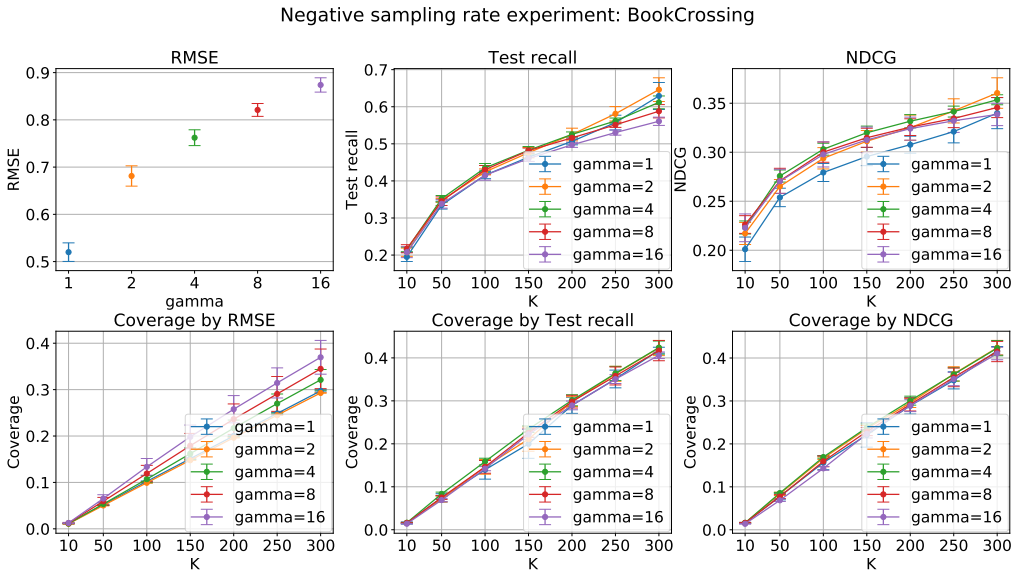


Figure 4.6: Effect of the negative sampling rate on the performance of the Hybrid cSDAE trained using BookCrossing.

expected since increasing the γ means increasing the number of negative ratings in each batch which indirectly raises the weight of the unknown ratings and pushes the model to either predict more zeroes or output ratings closer to zero. Which is exactly what the RMSE that is calculated only on the known implicit ratings measures.

The Hybrid cSDAE is surprisingly robust to the different values of γ in terms of the best reached Test recall. In case of MovieLens the reached recall values are practically identical for all Ks whereas the lower γ values gain a slight upper hand at $K \in \{250, 300\}$ on the BookCrossing dataset.

The model's NDCG performance is similarly robust to the changes in γ with the exception of $\gamma = 1$ which leads to consistently worse results over all values of K and both of the datasets.

The coverage scores reached by the models with the best RMSE reveal an interesting trend. Increasing the γ leads to worse coverage in case of MovieLens however the complete opposite applies to the BookCrossing dataset. This can be caused by the fact that the BookCrossing dataset is much more sparse and contains vastly more items. This probably leads to the increased coverage when the model predicts lower ratings on average. More items then have similar low predicted ratings and get mixed up into the top K which would also explain why the margins between the coverage scores increase with the increasing K, something that is not happening for

the MovieLens dataset.

All in all, the Hybrid cSDAE has proven to be quite robust against the different negative sampling rates if we avoid using values smaller than two.

4.5.3 Noise ratio experiment

Corrupting the autoencoder's input with noise is an important part of enabling it to produce meaningful compact representations of the data. We evaluate the behavior of the proposed Hybrid cSDAE architecture with noise ratios in $\{0.0, 0.2, 0.4, 0.6, 0.8\}$ while the default ratio used in the previous experiments was always 0.2. The model is again trained on the MovieLens 1m and the BookCrossing dataset. The other hyperparameters are set to the same values as in the negative sampling rate experiment specified in the Table 4.2. The experiment results are visualized in the Figures 4.7 and 4.8.

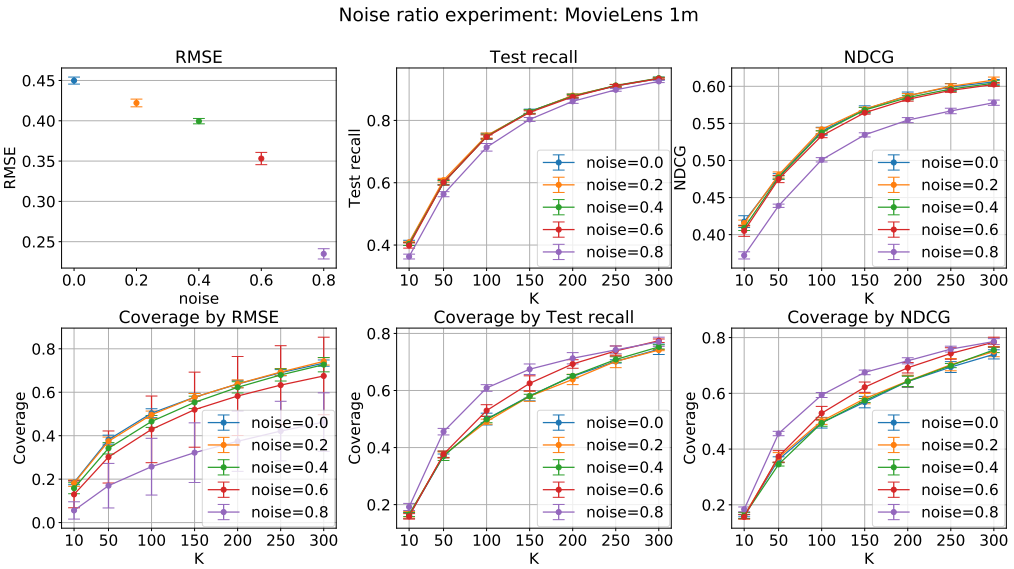


Figure 4.7: Effect of noise ratio on Hybrid cSDAE trained on the MovieLens 1m dataset. High noise leads to worse recall and NDCG but also to better coverage.

We can see that for both datasets the best reached RMSE is clearly getting lower with the higher noise ratio. That is expected since more noise leads to more 1s turned into 0s in the input. The model is then trained to reconstruct these 1s from the 0s which leads to the model predicting more

1s as it struggles to distinguish between the noisy 0s and the actual 0s. The relation between noise and number of non-zero values comes from the choice of binary masking noise that can only replace ones with zeroes.

The results on MovieLens 1m visible in the Figure 4.7 show practically identical scores of Test recall and NDCG for all ratios except 0.8. The worse performance with high noise is expected however we would also expect a decline in the test results with the absence of noise which would suggest overfitting. That is obviously not a problem in this case which may be caused by the structure of the dataset or the regularization properties of the Hybrid cSDAE model with the chosen hyperparametrization.

Even though the models with the highest noise ratio achieved the lowest Test recall and NDCG, they had significantly higher coverage at the same time. That is not surprising considering the high noise introduces a lot of uncertainty into the approximation of ratings which leads to different items getting to the top K.

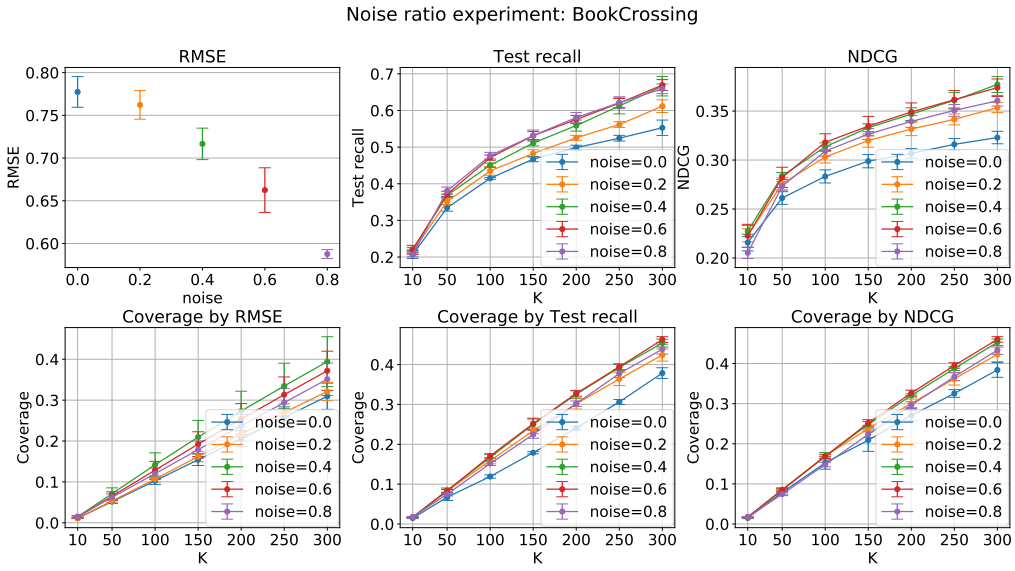


Figure 4.8: Effect of noise ratio on Hybrid cSDAE trained on the BookCrossing dataset. Absence of noise leads to worse recall, NDCG and also coverage.

Comparing the results on BookCrossing visible in the Figure 4.8, we can see that the Test recall and NDCG scores suffer with the absence of noise while staying relatively high with a large amount of noise. This is a complete opposite of the evaluation on MovieLens which is probably a

result of the serious differences in the structure of the datasets. The large amount of items and sparsity of their ratings apparently makes the model robust to higher noise while making it more susceptible to overfitting at the same time.

The coverage scores of the models getting the plotted Test recall and NDCG results are following a similar story. Instead of a high noise leading to a high coverage as with the MovieLens dataset, it results in similar coverage scores as most of the other ratios. The zero noise ratio is the value standing out this time, leading to a significantly lower coverage than the rest of them. This somewhat supports the overfitting hypothesis since an overfitted model would learn to predict only a specific group of items and therefore possibly reach a lower coverage than a well trained model.

Since we used the same model with the same hyperparametrizations, the only difference between the two evaluation results is the choice of the dataset. That means their structure is the most likely culprit for the different model results based on the various noise ratios. The number of users and items also influences the number of parameters the model has which could impact its regularization ability. It would be interesting to further investigate what exactly makes the datasets more susceptible to being overfitted on or otherwise causes the observed behavior. We leave this area for the future work.

Chapter 5

Conclusion

We have started the Chapter 2 by introducing the basics of the recommendation systems and by categorizing the algorithms they use according to how they generate the recommendations. We primarily looked at the model based algorithms to explain the latent factor models that are the focus of this work.

The LFM recommend the top K items sorted by their ratings that were approximated by a dot product of a user and item embedding. These embeddings can be produced in many ways and as the title of the thesis suggests, we focused on the models based on the artificial neural networks. We explained the different optimization algorithms and went over the advanced autoencoder architectures that are the basis of our proposed model.

In the Chapter 3 we presented various models with increasing complexity that lead to our novel Hybrid cSDAE architecture. We started by a neural network adaptation of the traditional matrix factorization, then we added more hidden layers to see what effect would it have on the performance. Replacing the feed-forward networks of the Deep MF with stacked denoising autoencoders lead to the Hybrid SDAE model which has been inspired by the Hybrid aSDAE that we presented but did not implement. Our proposed architecture Hybrid cSDAE is able to process both rating and attribute information in similar way as the aSDAE variant however it uses much less parameters to do so. We also added a Collaborative Deep Learning model that combines matrix factorization with an autoencoder processing item attributes in a quite specific way. We described both the theory and the details of our implementation of the models.

After introduction of the models including our novel architecture we

moved on to the final Chapter 4 explaining the experiments. We firstly discussed the chosen datasets and their preprocessing. Two MovieLens ones with a similar structure but different sizes and the BookCrossing dataset with a large number of items and a small number of users.

We also specified the training of the implemented models in detail. Especially how we solved the issue of training on the implicit ratings by constructing batches with a certain ratio of sampled unknown ratings presented as negative ones.

Then we introduced our design of the evaluation process. The most important point being that we split the rating matrix into the training and test sets by users instead of by ratings which is more usual in the literature. Splitting by users creates a more realistic scenario where the models never saw the test users during training. To eliminate the noisiness coming from the random selection of test users we ran a 5-fold cross-validation on each parametrization of the models in all the experiments.

In order to compare the five implemented models we conducted a grid search over selected hyperparameters. The results have shown how important it is to use several different metrics such as the Test recall, NDCG and coverage in our case. Taking all of them into consideration, our proposed Hybrid cSDAE architecture has achieved better or comparable results to the other models.

The last two experiments have investigated the effect of the negative sampling rate and the noise ratio on the performance of the Hybrid cSDAE model. The main takeaway point is that the Hybrid cSDAE is quite robust against the different values of both of these hyperparameters.

All in all the comparison of the models on several different metrics and datasets shows that there is rarely a single model dominating on all fronts. Even though some tasks can be solved more efficiently by simpler models like CDL, in case we need to incorporate both item and user attributes into the model and generate corresponding embeddings, our proposed Hybrid cSDAE is to our knowledge the most efficient autoencoder based model to do so.

5.1 Future work

There are a lot of possible future research directions to explore. We definitely plan to evaluate the quality of the embeddings produced by the various models. Especially how do they improve the performance of the standard recommendation algorithms such as K-nearest neighbors. It would

also be interesting to look at the effects of the increasing size of the bottleneck dimension on the various metrics. Since increasing the layer size requires a new hyperparameter search we unfortunately could not fit it into this work.

In order to improve the proposed model we plan to explore new ways of creating the training batches, test different activation functions and optimization algorithms. Using a pairwise loss instead of the pointwise mean square error could also bring some benefits to the ranking capabilities.

Bibliography

[1] Bartyzal, R. *Optimization of Recommender Systems*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technoľogy, 2016.

[2] All reading titles offered on Amazon [online]. `https://www.amazon.com/gp/feature.html?docId=1002872331#faq/ref=insider_ar_reading_primereading`, [Online; accessed 2018-September-06].

[3] All Movie DVDs offered on Amazon [online]. `https://www.amazon.com/s/ref=Movies_H1_DVD?rh=i%3Amovies-tv%2Cn%3A2625373011%2Cn%3A%212625374011%2Cn%3A2649512011%2Cp_n_format_browse-bin%3A2650304011&bbn=2649512011&rw_html_to_wsrp=1`, [Online; accessed 2018-September-04].

[4] Burke, R. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, volume 12, no. 4, 2002: pp. 331–370.

[5] Cremonesi, P.; Koren, Y.; et al. Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on Recommender systems*, ACM, 2010, pp. 39–46.

[6] Trewin, S. Knowledge-based recommender systems. *Encyclopedia of library and information science*, volume 69, no. Supplement 32, 2000: p. 180.

[7] Breese, J. S.; Heckerman, D.; et al. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth*

conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc., 1998, pp. 43–52.

[8] Lops, P.; De Gemmis, M.; et al. Content-based recommender systems: State of the art and trends. In *Recommender systems handbook*, Springer, 2011, pp. 73–105.

[9] Schein, A. I.; Popescul, A.; et al. Methods and metrics for cold-start recommendations. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2002, pp. 253–260.

[10] Brynjolfsson, E.; Hu, Y. J.; et al. From niches to riches: Anatomy of the long tail. 2006.

[11] Al Shalabi, L.; Shaaban, Z.; et al. Data mining: A preprocessing engine. *Journal of Computer Science*, volume 2, no. 9, 2006: pp. 735–739.

[12] Sedhain, S.; Menon, A. K.; et al. Autorec: Autoencoders meet collaborative filtering. In *Proceedings of the 24th International Conference on World Wide Web*, ACM, 2015, pp. 111–112.

[13] Dong, X.; Yu, L.; et al. A Hybrid Collaborative Filtering Model with Deep Structure for Recommender Systems. In *AAAI*, 2017, pp. 1309–1315.

[14] Wang, X.; Wang, Y. Improving content-based and hybrid music recommendation using deep learning. In *Proceedings of the 22nd ACM international conference on Multimedia*, ACM, 2014, pp. 627–636.

[15] Christakopoulou, E.; Karypis, G. Local item-item models for top-n recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems*, ACM, 2016, pp. 67–74.

[16] Zhang, S.; Yao, L.; et al. Autosvd++: An efficient hybrid collaborative filtering model via contractive auto-encoders. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval*, ACM, 2017, pp. 957–960.

[17] Liang, D.; Krishnan, R. G.; et al. Variational Autoencoders for Collaborative Filtering. *arXiv preprint arXiv:1802.05814*, 2018.

[18] Agrawal, R.; Imieliński, T.; et al. Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, ACM, 1993, pp. 207–216.

[19] Sarwar, B.; Karypis, G.; et al. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM conference on Electronic commerce*, ACM, 2000, pp. 158–167.

[20] Resnick, P.; Iacovou, N.; et al. GroupLens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, ACM, 1994, pp. 175–186.

[21] Sarwar, B.; Karypis, G.; et al. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, ACM, 2001, pp. 285–295.

[22] Wu, Y.; DuBois, C.; et al. Collaborative Denoising Auto-Encoders for Top-N Recommender Systems. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining - WSDM '16*, ACM Press, 2016, ISBN 9781450337168, pp. 153–162, doi:10.1145/2835776.2835837. Available from: `http://dl.acm.org/citation.cfm?doid=2835776.2835837`

[23] Ning, X.; Karypis, G. Slim: Sparse linear methods for top-n recommender systems. In *2011 11th IEEE International Conference on Data Mining*, IEEE, 2011, pp. 497–506.

[24] Koren, Y. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2008, pp. 426–434.

[25] Koren, Y.; Bell, R.; et al. Matrix factorization techniques for recommender systems. *Computer*, , no. 8, 2009: pp. 30–37.

[26] Gunawardana, A.; Shani, G. Evaluating Recommender Systems. In *Recommender Systems Handbook*, Springer, 2015, pp. 265–308.

[27] Said, A. *Evaluating the accuracy and utility of recommender systems*. Dissertation thesis, Universitätsbibliothek der Technischen Universität Berlin, 2013.

[28] Wang, Y.; Wang, L.; et al. A Theoretical Analysis of Normalized Discounted Cumulative Gain (NDCG) Ranking Measures. In *Proceedings of the 26th Annual Conference on Learning Theory*, COLT, 2013.

[29] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, volume 65, no. 6, 1958: p. 386.

[30] LeCun, Y.; Bengio, Y.; et al. Deep learning. *nature*, volume 521, no. 7553, 2015: p. 436.

[31] Ivakhnenko, A. G. The Group Method of Data of Handling; A rival of the method of stochastic approximation. *Soviet Automatic Control*, volume 13, 1968: pp. 43–55.

[32] Rumelhart, D. E.; Hinton, G. E.; et al. Learning representations by back-propagating errors. *nature*, volume 323, no. 6088, 1986: p. 533.

[33] Hinton, G. E.; Salakhutdinov, R. R. Reducing the dimensionality of data with neural networks. *science*, volume 313, no. 5786, 2006: pp. 504–507.

[34] Bengio, Y.; et al. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, volume 2, no. 1, 2009: pp. 1–127.

[35] Darken, C.; Chang, J.; et al. Learning rate schedules for faster stochastic gradient search. In *Neural Networks for Signal Processing [1992] II., Proceedings of the 1992 IEEE-SP Workshop*, IEEE, 1992, pp. 3–12.

[36] Dauphin, Y. N.; Pascanu, R.; et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, 2014, pp. 2933–2941.

[37] Ruder, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[38] Duchi, J.; Hazan, E.; et al. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, volume 12, no. Jul, 2011: pp. 2121–2159.

[39] Zeiler, M. D. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[40] G. Hinton's lecture introducing RMSProp [online]. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, [Online; accessed 2018-October-17].

[41] Kingma, D. P.; Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[42] Dozat, T. Incorporating nesterov momentum into adam. 2016.

[43] Loshchilov, I.; Hutter, F. Fixing weight decay regularization in adam. *arXiv preprint arXiv:1711.05101*, 2017.

[44] Reddi, S. J.; Kale, S.; et al. On the convergence of adam and beyond. 2018.

[45] FastAI's experimental results of Adam variants [online]. `http://www.fast.ai/2018/07/02/adam-weight-decay/`, [Online; accessed 2018-October-17].

[46] Vincent, P.; Larochelle, H.; et al. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, volume 11, no. Dec, 2010: pp. 3371–3408.

[47] Wang, H.; Wang, N.; et al. Collaborative Deep Learning for Recommender Systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*, ACM Press, 2014, pp. 1235–1244, doi:10.1145/2783258.2783273. Available from: `http://dl.acm.org/citation.cfm?doid=2783258.2783273http://arxiv.org/abs/1409.2944`

[48] Jia, X.; Song, S.; et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[49] Abadi, M.; Agarwal, A.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015, software available from tensorflow.org. Available from: `https://www.tensorflow.org/`

[50] Numpy. `http://www.numpy.org/`, [Online; accessed 27-September-2018].

[51] Pandas. `https://pandas.pydata.org/`, [Online; accessed 27-September-2018].

[52] scikit-learn. `http://scikit-learn.org/stable/`, [Online; accessed 27-September-2018].

[53] Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[54] He, X.; Liao, L.; et al. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2017, pp. 173–182.

[55] Adomavicius, G.; Tuzhilin, A. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, volume 17, no. 6, 2005: pp. 734–749.

Appendix A

Acronyms

AE Autoencoder

ANN Artificial Neural Network

CDL Collaborative Deep Learning

DAE Denoising Autoencoder

DL Deep Learning

LFM Latent Factor Model

MF Matrix Factorization

NDCG Normalized Discounted Cumulative Gain

RMSE Root Mean Square Error

SDAE Stacked Denoising Autoencoder

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with data disk contents description
- `src` ..... the directory of source codes
  - `implementation` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `DP_Bartyzal_Radek_2019.pdf` ..... the thesis text in PDF format