FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Neural Autoencoders in Recommender Systems
Student:	Bc. Michal Bajer
Supervisor:	doc. Ing. Pavel Kordík, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2019/20

Instructions

Survey variants of neural autoencoders and their applications in recommender systems. Implement a variational autoencoder, explore the effect of hyper-parameters such as the size of the bottleneck layer or the learning coefficient. Explore different regularization strategies (dropout, sparse encoding, l2 norm, etc.). Compare the performance of variational autoencoder with standard autoencoders of the same capacity on at least two recommendation problems. Evaluate experimental results and recommend the best architecture to maximize recall, catalogue coverage and other criteria.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Head of Department	Dean

Prague December 10, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Neural Autoencoders in Recommender Systems

Bc. Michal Bajer

Department of Applied Mathematics
Supervisor: doc. Ing. Pavel Kordík, Ph.D.

May 3, 2019

Acknowledgements

I would like to thank my supervisor doc. Ing. Pavel Kordík, Ph.D. for his support and advice during the work on this thesis. I would also like to thank my family and friends for their support.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme ”Projects of Large Research, Development, and Innovations Infrastructures” (CESNET LM2015042), is greatly appreciated.

Declaration

In Prague on May 3, 2019 …………………

Citation of this thesis

Bajer, Michal. *Neural Autoencoders in Recommender Systems*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce se zabývá možnostmi využití autoencoderů v rámci doporučovacích systémů, jejich potenciálem pro předpovídání chování uživatelů a rozdíly mezi různými variantami těchto modelů.

Cílem práce je zmapovat možné přístupy, stanovit vhodné metriky pro posouzení kvality doporučení, implementovat slibné varianty a porovnat jejich úspěšnost na dostupných datech.

Výsledkem práce je analýza a diskuze možných řešení, zmapování vlivu hyperparametrů na kvalitu doporučení a výběr nejvhodnějšího modelu na základě provedených pokusů.

Klíčová slova Neuronové sítě, autoencodery, doporučovací systémy.

Abstract

This thesis is concerned with the potential usage of neural autoencoders in recommender systems, their ability to predict user behaviour and the differences between variants of the models.

The goal of the thesis is to explore the possible solutions, determine suitable metrics for measuring the quality of recommendations, implement the promising solutions and compare their performance on available datasets.

The result of the thesis is an analysis and a discussion of possible solutions, experimental study of the effects of hyperparameters on the quality of recommendations and the choice of the most suitable model based on performed experiments.

Keywords Neural networks, autoencoders, recommender systems.

Contents

List of Figures

List of Tables

Introduction

As the Internet spread throughout the world more and more large businesses were built around it. Many of them have been capable of using the advantages offered by the technological development to outcompete their offline competitors.

One of the advantages is the unprecedented ability to collect data and analyse them. The results can then be used to adjust the content for each individual user.

Another advantage is the fact that online business can often reach many people in a large geographic area with small additional cost. That allows them to operate at scale. As a result, Amazon.com almost certainly offers more books than any book shop and Netflix has a larger collection of film than any video rental shop.

This gives online businesses potential competitive advantage, but it also presents some new challenges. For example, Amazon.com sells hundreds of millions of products [4] in the United States. How are you ever supposed to find what you're looking for among so many products? And how can the website help you?

One way of helping you is to use the collected data to point you towards products, which might interest you and hide those, which you won't buy anyway. This is the basic idea behind the topic of this thesis – the recommender system.

The goal of the recommender system is to predict what items a user might be interested in. Based on this prediction a website can point the user towards these items. If the recommendations are good, users will easily find what they are looking for and the merchant will sell more goods. Or at least that is the hope.

As the online stores became larger, more attention has been given to developing these systems. In this thesis I will explore one of the many ways in which the predictions of user preferences can be generated. I will be using neural networks or more specifically neural autoencoders. This algorithm has been popular in the field of Artificial Intelligence in recent years, but so far it hasn't been used in recommendations very often.

The goal of the thesis is to research the relevant literature and find appropriate ways of using autoencoders for the problem of recommendation.

First part will contain a discussion of the research area and relevant algorithms. The results of this discussion will be used to design a way of recommending items.

Once the design is complete, the resulting approach will be implemented together with a way of measuring the quality of the generated recommendations. The implementation will be tested on available datasets.

Finally, the experiments and the measured results will be analysed and the best model (or models) will be chosen.

Chapter 1

Research background

1.1 Recommender systems

The dream of the recommender system is to predict which products will be interesting to you and show them to you directly. For example, representatives from Netflix sometimes talk about potential future, where Netflix would only need to show the user several options. The user would almost certainly like at least one of them and started watching almost immediately.

The problem of recommendation has two main components. One is the user for whom the recommendations are generated. The other is in the literature usually called an item. An item can be a product, an article, a video, an advertisement or many other things, which are being offered to the user.

Data about both components and the interactions between them can be helpful. The basic assumption of the recommender system is that there are dependencies between users and items and the data can be used to find them.

For example, a user, who is interested in a horror film, is more likely to be interested in a similar film. Therefore, the knowledge that a given user has watched one horror film and the knowledge about which films are horrors could be used to generate recommendations for the user.

Obtaining data about items is a domain specific process. Recommendations are typically made with a database of items. If the items are sorted into categories or labelled, that information can be used to analyse similarities between them.

Data about users and their preferences are more difficult. In some cases explicit user feedback is available. For example, users may rate items using a five-star system.

Very often the user feedback is implicit. For example, a user clicking on a product is considered a positive feedback. In this case only user's interest in certain items is known. There is no negative feedback. The advantage of this approach is that data can be collected in the background. There is no need to ask the user for collaboration. As a result, there is a lot more data to analyse.

The systems are usually separated into three categories. The first category is collaborative filtering. These systems are based on predicting missing values in the matrix of user-item interactions. They use similarities between rows (users) or items (columns) to do so.

Content-based systems follow the premise, that user ratings can be explained based on the properties of items (e.g. film genres). Unlike the collaborative systems the description of items plays a key role.

The last category is knowledge-based systems. These systems are used in domains, where it is unlikely to collect enough data about a user to generate good predictions. One example could be real estate. There are many factors that influence choice of a house and people rarely look through houses, unless they intend to buy one. That makes prediction of user preferences almost impossible. Therefore, these systems rely on the user to share their own preferences. The result is an iterative search through the database.

There are two ways to define the recommendation problem with respect to its output [5].

1. Prediction problem: The system is expected to predict user ratings for items, which haven't been rated by the user yet. This can also be viewed as matrix completion problem, because it is based on an incomplete matrix of users and items.

2. Ranking problem: The system is expected to select *k* items, which will be shown to the user. This is a more real-world definition of the problem, because this is what is typically done with the selected items. This problem is also referred to as the top-k recommendation problem.

In practical terms the goal of the recommender system is to increase the merchant's sales. However, that is not an operational goal that an algorithm inventor can follow. There are several common operational goals [5].

1. Relevance: Almost by definition recommended items should be relevant to the user. Otherwise, they will be meaningless. However, there are other important goals.

2. Novelty: It is important that the recommended items haven't been seen by the user before. For example, there is very little value in recommending the most popular film of a given genre, because the user has very likely already seen it.

3. Serendipity: It seems to be beneficial, if sometimes the recommendation is surprising to the user. Meaning it is not only a new item similar to the ones, that the user has already seen. This helps the user to expand their horizons slightly and for the merchant it increases sales diversity.

4. Diversity: Typically, the final recommendation is a list of top-k items displayed to the user. If all the items are very similar, the user might not like any of them. More diverse list improves the chances, that at least one of the items will be interesting to the user.

1.1.1 History

One of the first recommendation systems was developed in 1994 by Paul Resnick from MIT and others. It was called GroupLens. It focused on bulletin boards, which were used to share articles online. As the number of the articles grew, many of them became noise.

In the words of the authors: ”GroupLens provides a new mechanism to help focus attention on interesting articles. It draws on a deceptively simple idea: people who agreed in their subjective evaluation of past articles are likely to agree again in the future.” [6]. In other words, this system used collaborative filtering to generate recommendations.

In more recent years the development of these systems was encouraged by Netflix with the Netflix Prize. The contest started in 2006 and the goal was to improve upon the results of algorithm called Cinematch, which was used by Netflix to recommend films.

A large dataset was published for the purposes of this contest. The training dataset contained approximately 100 000 000 ratings of 18 000 films from 480 000 users. The goal was to reach 10% improvement.

The winning team was called ”BellKor’s Pragmatic Chaos”. They reached improvement 10,06% in 2009 and won one million dollars [7]. Their algorithm was a combination of around hundred different elements. The main part was collaborative filtering using matrix factorization. Other parts included processing the time of viewing and other available information. The authors published the algorithm in their own article [8].

1.1.2 Collaborative filtering

Systems in this category use only the user ratings matrix to generate their recommendations. This makes the algorithms generally applicable, because they don’t have to be adjusted as much for a specific domain.

One of the biggest challenges is the fact, that the user ratings matrix is very sparse. Let’s take online streaming service as example. The service may easily offer hundreds of thousands of films or episodes. But most users have seen only tens or hundreds. Many have seen almost none.

The idea behind collaborative filtering is that the missing ratings can be predicted, because there are similarities between users and items. For example, let’s image that Steve and Bill have liked the same five films and Steve has also liked one more. The basic assumption is that there is now an increased chance that Bill will also like the sixth film.

There are two basic types of collaborative filtering algorithms.

1. Memory-based algorithms: They are also called neighbourhood-based. The recommendation is based on the most similar users or items (the neighbourhood). Typically, either users (rows) or items (columns) are used. The algorithm is then called user-based or item-based.

   Recommendation for Bill is required. For the user-based version rows will be processed. First the system will find n most similar users (rows) to Bill using a similarity function. Afterwards it can find ratings for items which Bill hasn't rated yet. Finally, it computes weighted average (by similarity) and generates the predicted ratings. The items with the highest value should be recommended.

   This approach is relatively easy to implement and can work well. The issue often arises with too sparse matrices. If an item has no rating yet, it can never be recommended.

2. Model-based algorithms: They process the matrix using variety of optimization algorithms and machine learning models. The goal is to find patterns in the matrix and use them to predict the missing ratings.

   One example of this type are latent factor models popularized by the Netflix prize. They are based on a simple observation. The rating matrix is sparse and the values are not independent. That suggests that it could be well approximated by a much smaller matrix.

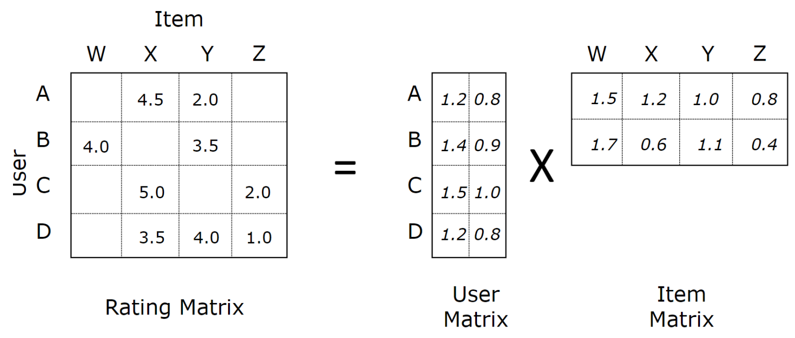


Figure 1.1: Example of Matrix factorization [1]

In state-of-the-art systems this is usually achieved using Matrix factorization methods like Singular Value Decomposition. The goal of Matrix factorization is to find two smaller matrices, which can be multiplied to create a new matrix as similar to the original one as possible. Missing values in the original matrix make it easier. The generated values in the new matrix also represent our predicted ratings. Example of these matrices can be seen in figure 1.1.

1.1.2.1 Singular Value Decomposition

SVD can be formally defined as factorization of a matrix into tree matrices:

$$R \approx Q_k \Sigma_k P_k^T \tag{1.1}$$

The rank k represents the size of matrix Σ_k and must be smaller than the smaller side of the original matrix. It is the only parameter which needs to be tuned in order to use SVD for recommendation purposes. The final matrices shown in picture 1.1 can be obtained in the following way:

$$U = Q_k \Sigma_k \tag{1.2}$$
$$V = P_k \tag{1.3}$$

The process of finding the relevant matrices in 1.1 is an optimization problem. Different ways of solving it can be found (among others) in the book [5].

1.1.3 Content-based systems

Unlike collaborative filtering methods, content-based systems consider only ratings given by the target user. They relate the rating of an item to its properties.

For the sake of simplicity, this section will consider a database of films, where each film has tags assigned to it. These tags may include the genre, the director, etc.

When a user gives a rating to a film, it is assumed that the rating can be explained by corresponding tags. When the films, their ratings and their tags are combined, it is possible to derive a set of attributes, which the user seems to like. The system can then look for other similar films and recommend those.

In reality the process of deriving more abstract description of user preferences can be based on many attributes and use a complex algorithm. This approach has one interesting advantage.

When a new film is added to the database, there are no user ratings for it. This would prevent collaborative filtering from ever recommending it. However, a content-based algorithm can recommend it as well as any other film as long as there are tags for it.

On the other hand, when the system is presented with a new user, there is no way to generate recommendation for them. That is because there is no history of ratings and therefore no information about the user's preferences.

Another potential disadvantage comes from the fact that with more ratings available the system refines the user preferences and stays within them. This can lead to obvious items being recommended and very little novelty.

1.1.4 Knowledge-based systems

The previous section introduced systems, which use the properties of items. They relied on ratings provided by users or collected as implicit feedback to generate recommendations. However, sometimes there is no reasonable expectation of collecting ratings or any form of feedback.

An example of such items could be cars. People usually buy a new car once in several years at most. The whole market can change during that period and so can the user preferences. At the same time a car is a very complex product with many attributes and many options. There is simply no way to collect sufficient amounts of data to generate good recommendations.

This is where knowledge-based systems can be used. They are based on a simple idea. If the system doesn't know anything about the user, it can ask them. As a result, these systems are typically a form of iterative search, where the user step by step clarifies their preferences.

The name knowledge-based comes from the fact, that these systems require a so-called knowledge base to function. The base contains domain-specific rules or similarity functions, which allow the systems to find the connection between the user preferences and items. There are two main types of knowledge-based systems:

1. Constraint-based recommender systems: The user interface allows the user to specify values for certain attributes, for example, a price range. The system then displays a set of items and the user can change the constraints. This iterative approach allows the user to find items they like.

2. Case-based recommender systems: Alternatively, the interface presents the user with examples. The user provides feedback and the system uses similarity functions to find items more alike the ones the user preferred.

1.1.5 Ensembles

Each type of recommender systems discussed in previous sections has advantages and disadvantages. For example, the knowledge-based systems are the best when there are no data about the user. Collaborative filtering can work very well when there are a lot of data.

Therefore, it can be interesting to use a combination of different methods. A simple way of doing this would be to use a different method for new users. Once enough data could be collected, second method would be used.

In realistic situations there are many ways to combine different systems. This situation is similar to ensemble machine learning systems used to generate more robust results.

Different algorithms of the same category can be used, or multiple types can be combined. Algorithms can process the same data, or they can be adapted to handle different data. This allows different types of data to be incorporated.

Once all the algorithms generate recommendations, rules or meta-model can be used to create the final set of recommended items.

1.1.6 Evaluation

The problem of recommendation can be viewed as a generalisation of regression in the following way. In regression the data are in the form of a matrix with columns representing features and a column, which represents target variable. The algorithm then predicts the values of the target variable based on the values of features.

In recommendation (especially in collaborative filtering) the data are a matrix with columns representing items. It is just as valid to invert the matrix and have columns represent users. The problem stays the same. There are no distinctions between the columns. The goal is to fill in any missing values in any column.

Because of this connection, many of the basic evaluation metrics from regression could be applied. For example, a mean squared error could be computed. However, because practically speaking the typical goal is the top-k list of recommendations, these metrics don't correspond very well to the expectations of the recommender system.

1.1.6.1 Online evaluation

From this perspective, the best way to evaluate a recommender system is A/B testing. The merchant starts with already functioning service, connects part of the users to the new system and compares the results to the old recommender system. The evaluation metric can be anything that has a business value, for example the conversion rate of users clicking on films, which were recommended.

However, this is not always an option and it can only be done with a system, which is already relatively mature. Therefore, offline evaluation metrics are necessary as well. Creating an offline metric with correlation to the results of an online tests is a difficult task [9].

To be able to perform offline evaluation, historical dataset of user interactions is required. Example of this can be the Netflix prize dataset discussed in section 1.1.1. As was discussed before, there are different goals for the recommendation system. Because of that, multiple metrics should be computed to try to fulfil different goals. Some of the most common ones are the following.

1.1.6.2 Accuracy

Because of the similarity to regression, accuracy can be applied very easily. Since part of the goal of recommender systems is to predict user preferences, accuracy can be used to measure this aspect. An example of a popular function in this category is the root mean squared error (RMSE).

$$RMSE = \sqrt{\frac{\Sigma_{i=1}^{n}(\hat{P}_i - P_i)^2}{n}} \tag{1.4}$$

The formula in 1.4 shows how to compute RMSE for n predicted values. $\hat{P}_i$ represents predicted value and P_i represents original value.

1.1.6.3 Recall

Recall is a metric which focuses on the top-k list of recommended items for given user u. The calculation is performed using a held-out set of items, that the user has clicked on. The recommended top-k list generated based on the remaining ratings is compared to the held-out set. The more items from the top-k list can be found in the set, the closer the resulting value of recall is to one.

$$Recall@K(u, \omega) = \frac{\Sigma_{k=1}^{K}\mathbb{I}|\omega(k) \in I_u|}{\min(K, |I_u|)} \tag{1.5}$$

The formula in 1.5 shows how to compute recall for the list of K items. The function $\omega(k)$ represents an item at position k in the top-k list and function $\mathbb{I}$ in an indicator function, which determines whether the item is present in the held-out set. Finally, the denominator is the minimum of K and the number of items clicked on by user u. This normalizes the value to range between zero and one [10].

1.1.6.4 Normalized discounted cumulative gain

NDCG is a metric similar to recall. It focuses on top-k list of recommended items and compares it to the set of held-out items. Unlike recall, it also considers the order of the items in the top-k list. When an item at better position is found in the held-out set, it contributes higher weight. The weight is given by monotonically decreasing function (e.g. logarithm).

$$DCG@K(u, \omega) = \sum_{k=1}^{K} \frac{2^{\mathbb{I}|\omega(k) \in I_u|} - 1}{\log(k+1)} \tag{1.6}$$

NDCG@K is the DCG@K linearly normalized to [0, 1] after dividing by the best possible DCG@K, where all held-out items are ranked at the top [10].

1.1.6.5 Coverage

Catalog coverage (CC) is a measure of the fraction of the database of items, which is recommended by the system to at least one user. As discussed in the previous sections, some systems can have issues with recommending new items. Because one the goals of recommender systems is diversity, the fraction of items ever recommended is relevant.

$$CC = \frac{|\cup_{u=1}^{m} T_u|}{n} \tag{1.7}$$

The formula in 1.7 shows how to compute catalog coverage for database of n items and top-k list generated for m users. T_u represents top-k list for user u.

The structure of the recommendation systems section was based on the book Recommender systems by Ch. Aggarwal [5].

1.2 Neural networks

This section serves as an introduction into models, which will be ultimately used in this thesis for recommendation. The origin of these models is in the field of machine learning. The introduction will use examples from this field.

The idea of a neural network in based loosely on our understanding of neuroscience and the function of our brain. The elementary unit is called a neuron. Roughly speaking, neurons are connected into a larger network and communicate using electrical impulses. The structure of a single neuron shown in figure 1.2 consists of several key parts.

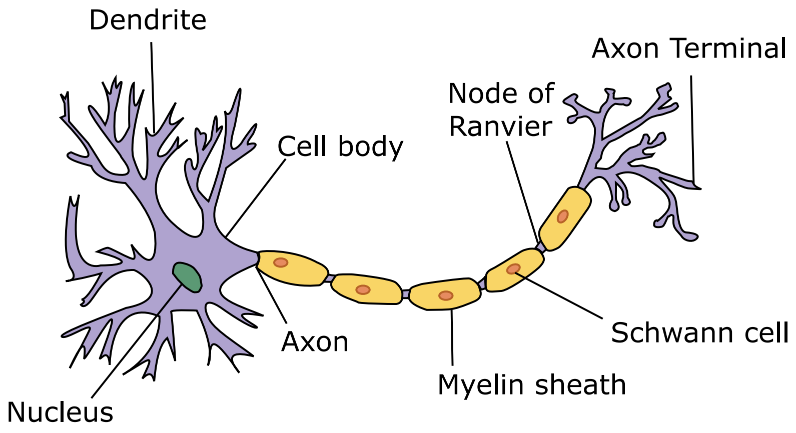


Figure 1.2: Structure of neuron in a human brain [2]

The dendrite is where the neuron receives impulses from other neurons. A single neuron has many of them and receives impulses from many other neurons. The axon is a longer nerve fibre, which transmits impulses away from the neuron and connects to dendrites of other neurons. The cell body, through its internal reactions, decides when to send an impulse over the axon.

This basic idea has inspired an abstract model called the neural network. One of the first attempts to use this model as a classifier was made by Frank Rosenblatt in 1958 and became known as the perceptron [11].

The perceptron is an algorithm for binary classification which determines its output using the formula 1.8. The function acts as a threshold. Vector $\mathbf{w}$ represents the weight assigned to each input. Vector $\mathbf{x}$ represents the input data. Parameter b is called bias. By adjusting (learning) $\mathbf{w}$ and b the output can be changed to conform to our expectations.

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases} \tag{1.8}$$

In modern terminology of neural networks, this is a neuron with Heaviside step function used as the activation function. It was soon discovered that a single neuron (or a single layer of neurons), where the input is a vector of features and the output is a class, can only be used to classify linearly separable problems. This led to a period of low interest in this algorithm, until the idea of the multi-layer perceptron was introduced.

1.2.1 Multi-layer perceptron

As the name suggests, the idea of the multi-layer perceptron (MLP) is a generalization of the original perceptron. The behaviour of a single unit is similar. First step is a weighted sum of its inputs.

$$in_j = \Sigma_{i=0}^{n} w_{i,j} a_i + b = \mathbf{w} \cdot \mathbf{x} + b \tag{1.9}$$

The formula 1.9 is very similar to 1.8. The input for neuron j is a weighted sum of n values plus bias. The main difference between a neuron in MLP and in the original perceptron is the activation function. The final output of the neuron is an activation function $\mathbb{A}$ applied to its input.

$$out_j = \mathbb{A}(in_j) \tag{1.10}$$

Any MLP consists of, at least, three layers of nodes: an input layer, a hidden layer and an output layer. The input layer is in fact only an abstraction for the input vector $\mathbf{x}$, which contains the feature values from a given dataset. The hidden and output layers contain neurons. Except for the first hidden layer, the input vectors $\mathbf{x}$ of these neurons are always a vector of outputs of all neurons in the previous layer.

The activation function should be a nonlinear function. That allows the MLP to represent a nonlinear function applied to feature vectors. The output of the function represents the class. As a result, this model can be used to classify problems, which are not linearly separable [12].

The activation function can be a threshold like in the case of the perceptron. More commonly it is differentiable function. For example, the logistic sigmoid 1.11.

$$S(z) = \frac{1}{1 + e^{-z}} \tag{1.11}$$

The resulting MLP is a complex function with many parameters to be learned. This is done using a method called backpropagation, which was derived by multiple researchers in the early 60's. It is a shorthand for "the backward propagation of errors".

1.2.2 Backpropagation

To explain how neural networks can be trained, first a loss function must be defined. The loss function represents an error between the output of the neural network and the expected output in the training dataset.

$$Loss = |\mathbf{y} - h_{\mathbf{w}}(\mathbf{x})|^2 = \Sigma_{k=1}^{n}(y_k - a_k)^2 \tag{1.12}$$

Formula 1.12 is an example of mean squared error used as the loss function. Vector $\mathbf{y}$ represents expected output vector and $h_{\mathbf{w}}(\mathbf{x})$ represent the neural network with a_k being an element of the output. Therefore, the error is between the expected and actual output of the network.

The training process is an optimization using gradient descent. In order to be able to use this method, the entire loss function needs to be differentiable with respect to weights $\mathbf{w}$ [12].

$$\frac{\partial}{\partial w} Loss = \frac{\partial}{\partial w}\Sigma_{k=1}^{n}(y_k - a_k)^2 = \Sigma_{k=1}^{n}\frac{\partial}{\partial w}(y_k - a_k)^2 \tag{1.13}$$

For the output weights in the output layer, gradients can be derived relatively simply as shown in 1.13. However, the entire network can be a complex function. The output of every layer is the input of the next and as a result every layer is an extra nested function.

However, the entire function can be decomposed using the chain rule. The weights are updated layer by layer starting with the output layer and going back through the network. That is why the process is called the backpropagation of error.

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j \tag{1.14}$$

Each weight can be updated by rule 1.14. The value of the weight $w_{i,j}$ of the connection between neurons i and j changes between iterations. The change is given by the learning rate α, the output a_i of neuron i and the propagated delta Δ_j.

The value of Δ_j is determined by the sum of deltas Δ_k of neurons in the next layer weighted be the weights $w_{j,k}$. The last important element of the calculation is the derivative $g'(in_j)$ of the activation function.

$$\Delta_j = g'(in_j)\Sigma_{k=1}^{n} w_{j,k}\Delta_k \tag{1.15}$$

1.2.3 Convolutional neural networks

The structure of MLP is often called feed-forward neural network with fully connected layers in modern literature. The standard approach is to take outputs of all neurons in the previous layer and use all of them as the input of every neuron in the next layer. However, this approach turned out to be impractical for visual data.

For visual inputs the convolutional layers are commonly used. The neurons are organized into 2D grid and every neuron uses as its inputs only the outputs of a corresponding neuron in the previous layer and its neighbours. The vector of weights is replaced by a matrix of weights, which is shared by all neurons in a given 2D grid. As a result, the matrix acts as a mask applied to every element of the input matrix. This operation is called convolution and gives name to the layer.

Because applying single mask to the input may cause too much information to be lost, every layer typically consists of several grids. This effectively gives the layer third dimension. Each grid takes input from corresponding grid in the previous layer. This way multiple independent representations of the original image can be passed through the network.

It is not strictly necessary to apply convolution in two dimensions. 1D version can be used, for example, on time series data, while 3D version can be used on a video.

1.3 Autoencoders

Until now, the discussion focused only on neural networks, which are used for classification. That means they belong in the category of supervised learning. However, they cannot be directly used in a recommender system. There are no labels in recommendation. That is why this chapter will end with a version of neural networks used for unsupervised learning.

The autoencoder is a neural network with two parts: encoder and decoder. During the learning process the same vector is used as the input into the encoder and as the target of the decoder (in place of the class variable in MLP).

Most importantly, the output vector of the encoder (the latent vector) is the input of the decoder. And the latent vector is smaller than the original input vector. During the training, the encoder is forced to generate smaller representation of the input vector, which can then be used by the decoder to recreate the original vector. In other words, the encoder is forced to create more compact representation of the input.

With randomly generated input vectors, this task would not be possible. However, for any meaningful learning the datasets contain input vectors, which represent something and are related to each other. The autoencoder then

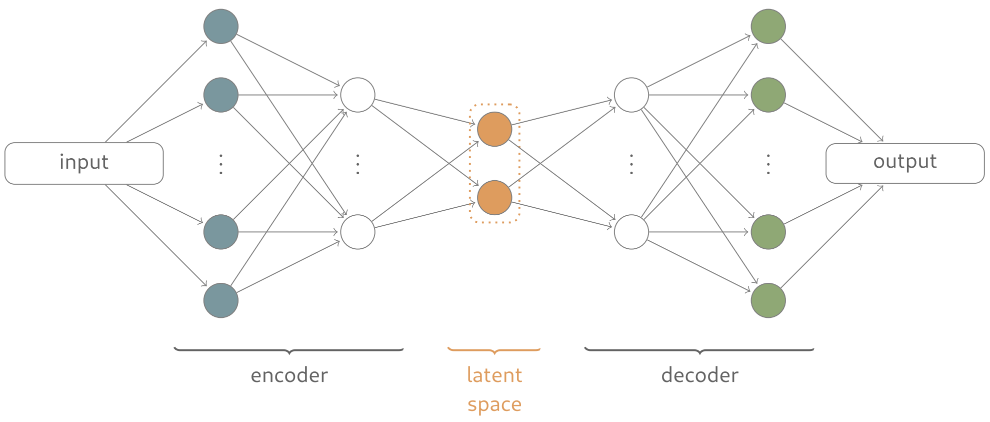


Figure 1.3: Structure of an autoencoder [3]

attempts to learn the hidden structures in the dataset in order to find low dimensional representation of the vectors [13].

Loss function is needed to be able to backpropagate through the autoencoder. The main component of this function is the similarity between the input and output vectors. The term reconstruction error is typically used for this function. However, more terms can be added to the equation and include other aspects into the learning process.

1.3.1 Sparse autoencoder

Questions arise about how to choose the structure of the autoencoder. For example, how large the latent vector should be. The hyperparameters of neural networks often need to be found using experiments or iterative optimization strategy.

In the case of the optimal size of layers, it is possible to make that part of the learning process in the following way. If the loss function could be extended to penalize too many activations of neurons, the optimization process would also attempt to use as little neurons as possible in every layer during training. This extra term in the loss function is called sparsity constraint.

$$J_{sparse} = J(\mathbf{w}, b) + \beta \Sigma_{j=1}^{n} KL(\rho || \hat{\rho}_j) \tag{1.16}$$

The formula 1.16 is the loss function of the sparse autoencoder. The original reconstruction loss is represented by $J(\mathbf{w}, b)$. Parameter β controls the weight of the sparsity penalty term and n is the number of neurons in a layer. The term $\Sigma_{i=1}^{n} KL(\rho || \hat{\rho}_j)$ is the Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$. KL-divergence is a standard function for measuring how different two different distributions are [13]. The term is computed using the formula 1.17,

$$\Sigma_{j=1}^{n} KL(\rho||\hat{\rho}_j) = \Sigma_{j=1}^{n} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{(1-\rho)}{(1-\hat{\rho}_j)} \tag{1.17}$$

where ρ represents the sparsity parameter and $\hat{\rho}_j$ is the activation (the output) of a neuron j averaged over the dataset. Parameter ρ is typically chosen to be a small number (for example $\rho = 0.05$). The effect of KL-divergence is bringing $\hat{\rho}_j$ closer to our selected ρ.

In implementation, the sparsity of a layer can also be achieved by applying L1 normalization to it. This penalizes too many activate neurons at same time.

In conclusion, by adding extra term to the loss function, it is possible to make more compact latent representation one of the goals of the learning process.

1.3.2 Variational autoencoder

There is a potential downside to the standard autoencoder. The vectors in the latent space don't follow any specific distribution. This isn't an issue for applications, where autoencoders are used for compression or removing noise. However, it is a complication for use in generative models.

For generative models it would be beneficial to be able to artificially create a latent vector, which the decoder will transform into meaningful output. For this purpose, it is better to know the distribution from which the latent vector should be sampled. Variational autoencoders have been developed to allow this.

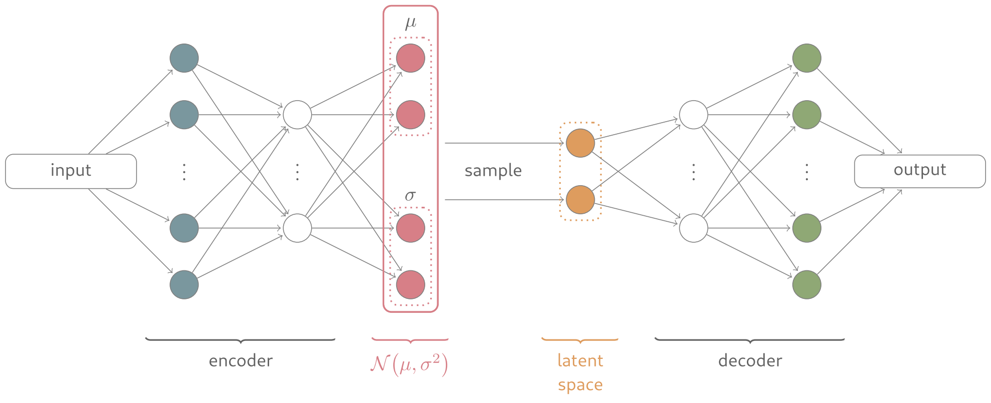


Figure 1.4: Structure of a variational autoencoder [3]

The way a variational autoencoder guarantees the distribution is quite simple. Instead of passing the output of the encoder directly into the decoder, the output is interpreted as two vectors. These vectors are then used as mean

μ and standard deviation σ of normal distribution $\mathcal{N}(\mu, \sigma^2)$. The latent vector is then sampled from this distribution.

As a result, the decoder learns to create something similar to the original input vectors from samples generated from normal distribution. This makes the usage in generative modelling easier.

In order to use the normal distribution meaningfully, one more adjustment is necessary. If someone simply used reconstruction loss during the learning process, the standard deviation σ would go to zero and mean μ would effectively became the latent vector from standard autoencoder.

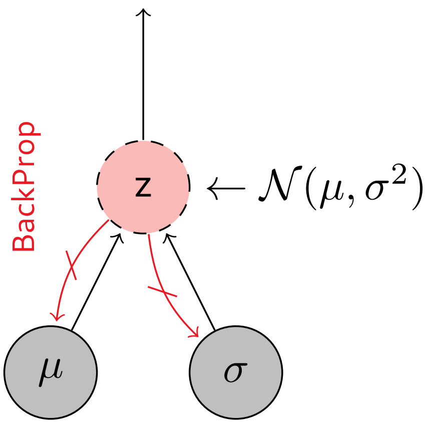
To prevent this, KL-divergence (one example of it is equation 1.17) is added to the loss function as a distance between the learned distribution and the standard normal distribution $\mathcal{N}(0, 1)$.

The loss function consists of reconstruction loss and KL-divergence. In order to be able to calculate gradient for every layer and use backpropagation, it is not possible to have the normal distribution in the latent layer directly. The reparameterization trick is typically used to solve this issue.

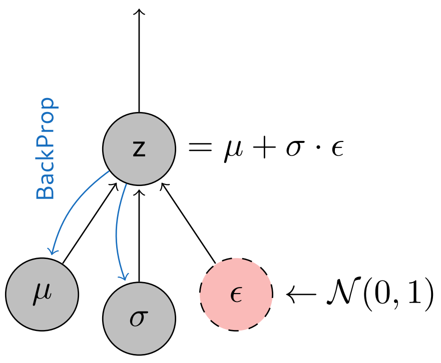
$$\mathcal{N}(\mu, \sigma^2) \sim \mu + \sigma^2 \mathcal{N}(0, 1) \tag{1.18}$$

$$z = \mu + \sigma^2 \epsilon, \quad \epsilon \leftarrow \mathcal{N}(0, 1) \tag{1.19}$$

Thanks to the equivalency 1.18 latent vector z can be generated using formula 1.19. As a result, the learned parameters μ and σ are no longer parameters of the normal distribution and therefore the training doesn't depend on the computation of its gradient. This is illustrated in figure 1.5.



(a) Before


BackProp
z
$= \mu + \sigma \cdot \epsilon$
μ
σ
ϵ
$\leftarrow \mathcal{N}(0, 1)$

(b) After

Figure 1.5: The principle of the reparameterization trick [3]

Chapter 2

Analysis and design

2.1 Datasets

2.1.1 MovieLens

The MovieLens 20M Dataset has been collected by researchers from the GroupLens Research Group at the University of Minnesota. It was compiled from user activity on movielens.org between January 09, 1995 and March 31, 2015. It contains 20,000,263 ratings across 27,278 films. The data were created by 138,493 users. Users were selected at random for inclusion. All selected users had rated at least 20 films. [14].

The ratings are on a 5-star scale between 0.5 and 5 stars. For the purposes of this work, the scale was transformed into implicit feedback matrix. Ratings of 4 stars and above became 1, everything else became 0. The final size of the ratings matrix can be found in table 2.1.

The dataset also contains tags for every film and so-called tag genome. It is a matrix, which encodes how much a certain film corresponds to given tag. For each film the title and genre are also known. This information could be used for recommendation in more context-aware systems.

2.1.2 Netflix

Netflix provided a dataset of 100,480,507 ratings, given by 480,189 users to 17,770 movies, for the purposes of the Netflix prize [7]. The average user in the dataset rated over 200 movies, and the average film was rated by over 5,000 users. Some films in the dataset have as few as 3 ratings, while one user rated over 17,000 movies.

The format is similar to the MovieLens dataset. The ratings are given in the form of 5-star scale. For the purposes of this work, it has been transformed into implicit feedback matrix. Ultimately, the preprocessed dataset becomes a sparse matrix in the Compressed Sparse Row format.

Dataset	MovieLens	Netflix
Number of users	136,472	461,381
Number of items	17,218	15,422
Size of CSR matrix	7.5MB	34.9MB

Table 2.1: Comparison of processed dataset sizes

2.2 Preprocessing

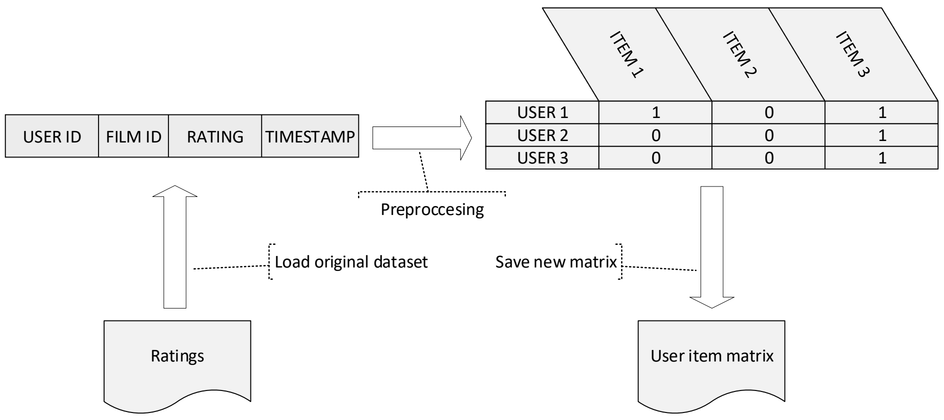


Figure 2.1: Visualization of data preparation

Prepared datasets are to necessary perform the experiments, which will form the core of this thesis. Datasets used in this thesis come in a similar format. They contain a list of ratings given by a user to an item. The rating is given in the form of 5-star scale. Additionally, they also contain other metadata about the items.

The implemented algorithms expect ratings in the form of the user-item matrix. The basic transformation into this form is illustrated in diagram 2.1.

The original data are a list of vectors in the following format:

$$[userId, movieId, rating, timestamp]$$

For the purposes of this work, explicit user feedback in the form of 5-star scale isn't ideal. That is why it has been transformed into implicit feedback. One represents an item that a user took interest in, everything else is zero. Ratings of four and above are considered positive feedback.

As the last step, the vectors have been transformed into a matrix, where rows represent users and columns represent items. Because the matrix is very large and sparse, the Compressed Sparse Row format is well suited to represent the matrix in memory.

2.3 Methodology of experiments

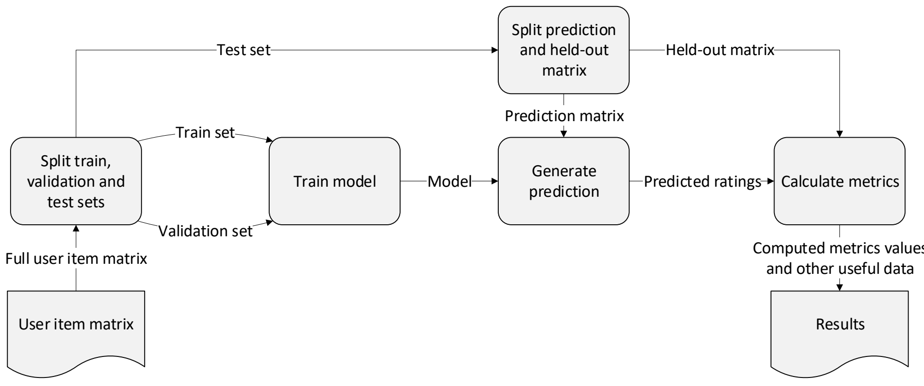


Figure 2.2: Processing steps and data flow between them

Once the dataset is prepared, models can be trained and evaluated. In order for the results to be meaningful, several steps are necessary. The basic steps and the data flow between them are visualized in diagram 2.2.

2.3.1 Splitting sets

First the dataset needs to be split into training, validation and test sets. This effectively simulates the situation, when model is trained on given data and later used for predictions on new data. Evaluating on different data allows for verification in order to find out whether the model generalizes well.

Training set will be used to train the model. Validation set will be used to evaluate model during training to limit over-fitting. Test set will be used, when the training is finished, to generate predictions and calculate metrics.

Training set will contain $\frac{3}{5}$ of the original dataset. Validation and test sets will each contain $\frac{1}{5}$. The split will be along the user dimension. The items will remain the same in every subset.

2.3.2 Training model

The training process of a neural network is an iterative process. The structure of the network and the training process itself are governed by many parameters. Finding good values for these hyperparameters is a major part of this thesis.

2.3.3 Evaluation

The test set will be used for the purposes of evaluation. However, before it can be used, it needs to be split into two parts. All users and all items will be present in both parts.

In order to calculate many of the metrics described in section 1.1.6 two sets of ratings need to be created. One matrix (the prediction matrix) will contain most of the ratings, and the second matrix (the held-out matrix) will contain the remainder.

This split allows to generate predictions for ratings using the prediction matrix and then compare them to the actual ratings in the held-out matrix. This simulates the situation, where a user receives a recommendation. Their actual preferences (known from the held-out set) are compared to the models predicted preferences.

The held-out matrix will contain 20% of the ratings. Because during the split some users in the prediction matrix can be left with no ratings (no ones), these users will be removed. As a result, the two matrices have the same dimensions, but some users have been removed compared to the original matrix.

2.4 Hyperparameter optimization

Hyperparameters are configurable parameters of a model. It could be the maximal depth of a decision tree, the number of neighbours in the k-nearest neighbours algorithm or the number of hidden layers in a neural network. In an abstract sense, it is simply a value with a given range of meaningful values, which can be adjusted to improve the performance of the model.

The purpose of the hyperparameter optimization is finding the hyperparameters of a model in a way, that maximizes its performance, while investing sensible amount of resources into it.

2.4.1 Uniform sampling

One way of a finding a good value for a parameter is to start by determining a range of values for it. The range can often be deduced from the context reasonably well. Then a process can sample the range uniformly and use the value, which allowed the model to reach the best performance.

When this approach is used for multiple parameters at the same time, it is typically called the grid search. For every parameter it chooses a list of values and trains the model with every combination of the values.

This approach works well for a few parameters, but the number of models to train grows quickly with the number of parameters. This becomes especially problematic for models, which have large number of parameters and take long time to train. That happens to be the case with neural networks.

2.4.2 Model-driven optimization

The number of models, which need to be trained, to find a good configuration of parameters can be lowered by assuming, that there are underlying dependencies, which can be modelled.

Using this assumption, a model of the space of parameters can be build and used to guide the sampling process, so that the process primarily trains models with parameter values, which seem to be promising.

There are multiple models which can be used for this purpose. One option is the Bayesian optimization using Gaussian Processes [15], which seems to have good empiric results. Various other options can be used. For example, the Scikit-optimize library [16] also offers the decision trees and the gradient boosted trees.

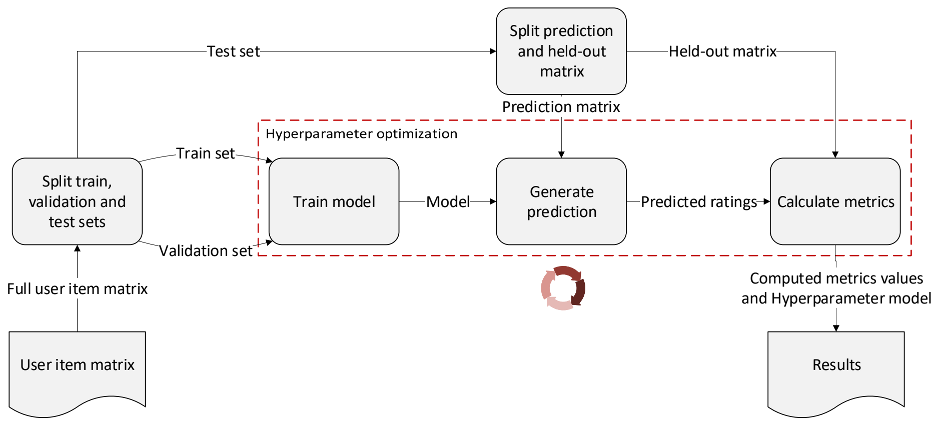


Figure 2.3: Hyperparameter optimization as part of the training process

Chapter 3

Implementation

All software used for the preprocessing of data, training of models and evaluation of the results was implemented in Python 3.6. The implementation and testing of the code were done locally using small parts of the used datasets.

More resources were necessary to process the full datasets. I used the services provided by metacentrum.cz (operated by CESNET z.s.p.o). This allowed me to process the entire matrices in memory, especially when they needed to be dense, which sometimes required tens of gigabytes of memory.

They also provide access to GPUs, which allowed me to train large models in reasonable time. This ability, for example, allows for more interesting parameter search experiments.

3.1 Libraries

The main reason for choosing Python as the language for the implementation was the availability of good libraries for machine learning and data processing tasks. The description of the main used libraries follows.

3.1.1 Tensorflow

Tensorflow is a library primarily used for training neural networks. The user describes the structure of the network and the library creates a computational graph.

High-level API called Keras can be used to describe the network. It can be combined with low-level API, where the user specifies the computational nodes directly. This allows combining standard functionality from the library with custom code easily.

The computational graph is then connected with the data on which the computation will be performed. The computation itself can be performed on a CPU, GPU or a specialized accelerator. It can even be performed in distributed environment.

Version 1.11 was used for this implementation.

3.1.1.1 Tensorboard

Tensorflow library has its own tool for monitoring progress of training. It is very easy and useful to collect metrics (primarily the loss function for training and validation sets) during the training process and display it using this tool.

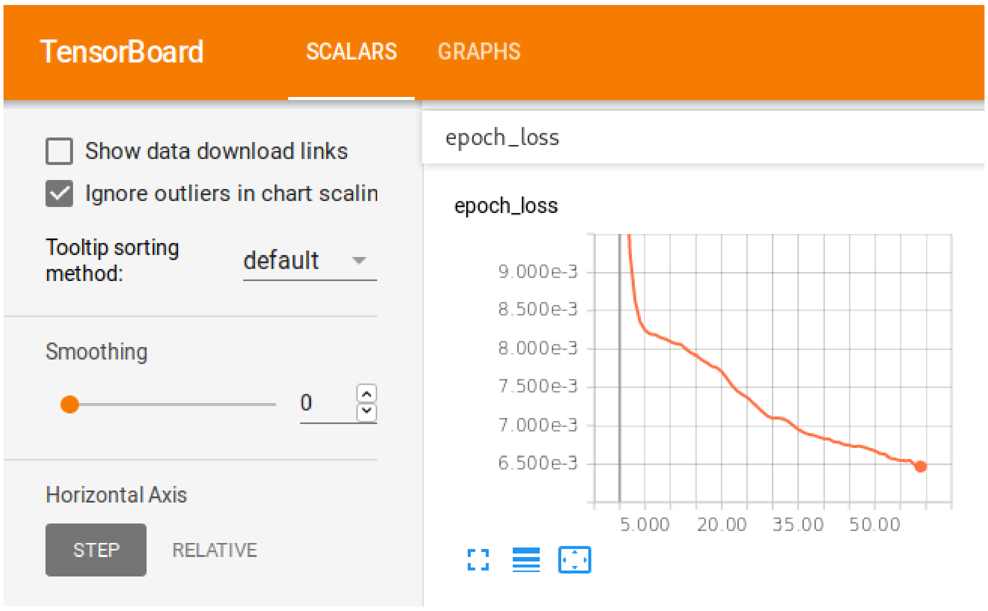


Figure 3.1: Example of epoch loss plotted in Tensorboard user interface

Because the experimental part of this thesis is primarily focused on hyper-parameter search, the visualizations usually represent a set of models. Therefore, the output from Tensorboard for single model isn't shown in the experiments chapter.

The tool was very useful during the initial phase of implementation to validate the training process and to monitor over-fitting during the experiments.

3.1.2 Numpy/Scipy

Numpy and Scipy are the de facto standard choice for processing vectors and matrices in Python. Numpy is focused primarily on dense matrices and linear algebra. Scipy focuses on wider range of mathematics.

For the purposes of this work, the main usage of Scipy was the sparse matrix representation, which allowed to store the datasets much more efficiently. The size of the resulting files is on the order of tens of megabytes. The dense matrix equivalent would require several gigabytes of storage.

3.1.3 Scikit-optimize

This is a relatively new library, which became part of the larger Scikit ecosystem. It focuses on hyperparameter optimization and provides several methods of achieving it. It also includes tools for visualizing the results.

In order to use these methods, one metric needs to be chosen. Its value will then be minimized. Most experiments have used the negative of NDCG as the metric. The other metrics of the models were also saved and evaluated later.

Chapter 4

Experiments

This chapter will cover the performed experiments and their results. The structure should follow a ”constructive” approach. First start with a simpler model. Evaluate the model and then attempt to add something that could help and evaluate again. That is the basic work flow, which I intend to follow.

The first section introduces metrics used for evaluation. Afterwards the first experimental section uses a different well established model to achieve the same results. That should provide a baseline comparison of the results. In the next section the first autoencoder will be presented.

4.1 Metrics

The models will be evaluated using the following metrics.

1. Recall@20
2. Recall@50
3. NDCG@100
4. Coverage@50

The theory behind them can be found in section 1.1.6. The first three could be summarized as a representation of the accuracy of the predicted user preferences. Coverage can be seen as representing diversity of recommendations. The choice was inspired by paper [10].

4.2 Matrix Factorization

4.2.1 MovieLens

The entire dataset was used to evaluate the SVD method of matrix factorization. Since SVD processes and predicts the entire matrix at the same time, there is no concept of train or test sets.

In order to perform the standard evaluation used in this work (results are in table 4.1) the entire matrix was split into prediction and held-out matrix. During this process some users were left with only zeros in the prediction matrix. They were removed, and the total number of users dropped to 120,720.

4.2.2 Netflix

The Netflix dataset was too large to process because the SVD method requires dense matrices and the memory requirements were too extreme (about 560 GB of RAM would be necessary).

Therefore, the method was applied only to first 50,000 users, which were further limited during the split into prediction and held-out matrix. At the end 42,363 users were used.

Dataset	Recall@20	Recall@50	NDCG@100	Coverage@50	RMSE
MovieLens	0.16036	0.30237	0.15459	0.04686	0.03494
Netflix	0.17015	0.31099	0.17333	0.05984	0.04369

Table 4.1: Results of matrix factorization applied to two datasets

4.3 Autoencoders

4.3.1 AE with no hidden layers

The first autoencoder (AE) to evaluate is the simplest one. This model contains only the input layer, the latent layer and the output layer. The activation in the latent layer is linear, which means only the weights of the connections between the input and the latent layer are relevant (and the bias of each neuron).

This means that there are only few hyperparameters left to choose. The size of the latent layer is the most important one. The size of the layer should be to a certain degree given by the dataset itself. Ideally the smallest possible size, which can represent the input, should be used.

The other parameters are more connected to the learning process. It is the learning rate and the number of epochs. The hyperparameter optimization was used to explore the parameter space.

4.3.1.1 Parameter search

Graph 4.1 shows the results of the search as represented by the Scikit-optimize library. The individual graphs show pairwise relationships between the three parameters. Lighter colour means better value of the optimized metric NDCG. The graph at the end of each row show the relation between each parameter value and the metric as modelled by the optimization process.

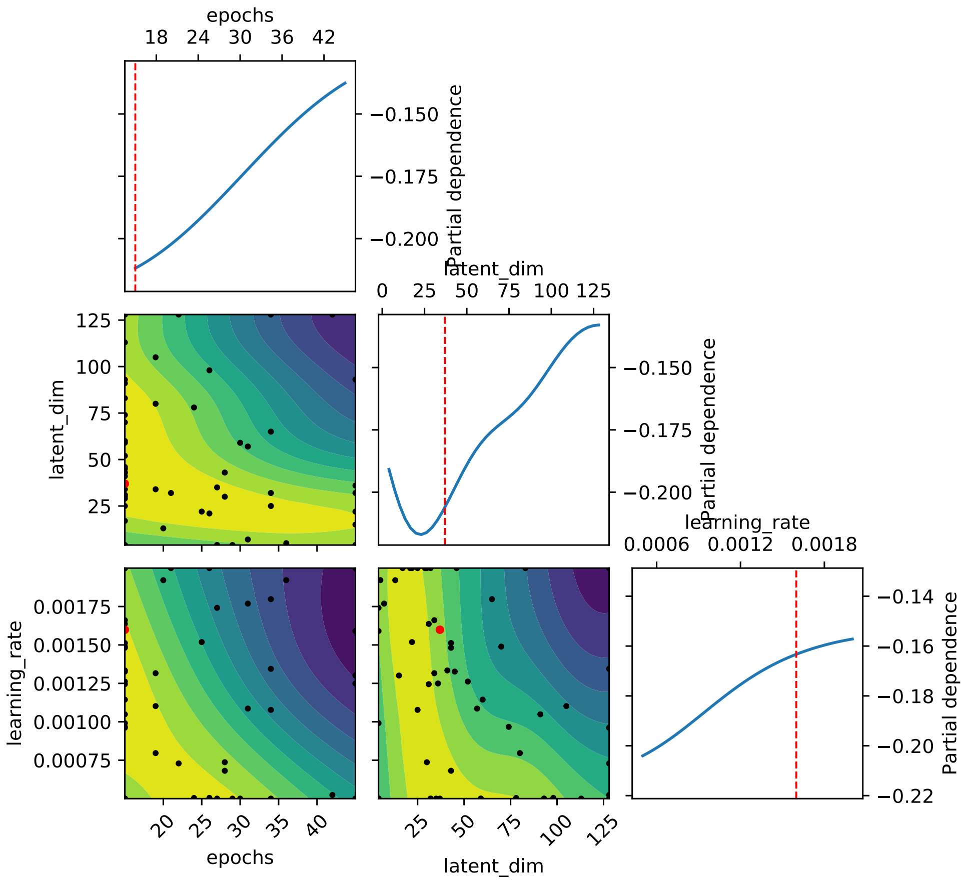


Figure 4.1: Visualization for the parameter space of AE with no hidden layers

Based on the results from MovieLens and similar results from Netflix dataset, the best values appear to be:

- Number of epochs: 20
- Size of the latent vector: 32
- Learning rate: 0.001

These values were used to train a model for each dataset. The results will be presented in the next section.

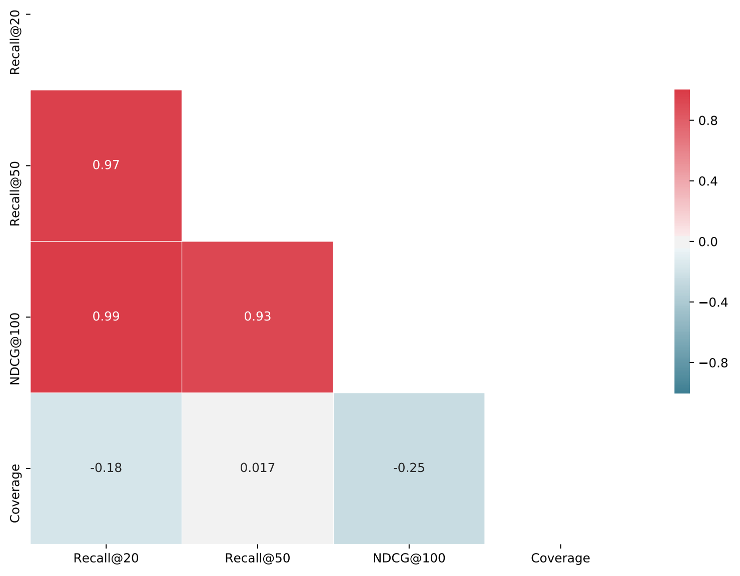


Figure 4.2: Correlation between used metrics

The results of the parameter search can be used to show one interesting property. The correlations between the metrics used to evaluate the model. The graph 4.2 shows the pairwise correlations.

The measurements generally agree with expectations. Recall@20 and Recall@50 are almost fully positively correlated. NDCG@100 also has very high positive correlation with recall. Coverage has significant negative correlation with both previous metrics.

The negative correlation was expected. As a model becomes more successful in predicting the user preferences, it recommends less arbitrarily and the fraction of recommended items goes down.

4.3.1.2 Results

Once the appropriate parameters were found, the final model was trained for both datasets. Table 4.2 shows the results. When compared to results of the matrix factorization (table 4.1), values of all metrics have improved. It is also interesting to point out, that coverage has increased alongside the other metrics.

That shows the differences between the models. For the same model, coverage tends to decrease as the other metrics increase. This result suggests, that autoencoder will overall give a chance to a large set of items. That is often a beneficial property for recommendation system.

The results also show, that the performance on Netflix dataset is worse than on MovieLens dataset. That is likely (at least partially) caused by the fact, that the rows of the Netflix dataset contain less ratings. As a result, the input vectors are sparser and the training process is more difficult.

Dataset	Recall@20	Recall@50	NDCG@100	Coverage@50
MovieLens	0.25348	0.40961	0.22961	0.11766
Netflix	0.19062	0.33683	0.19431	0.25885

Table 4.2: Results of AE with no hidden layers applied to two datasets

4.3.2 AE with one hidden layer

In this section the complexity of the model will increase. One hidden layer will be added to both encoder and decoder. This should in theory allow the model to better learn a complex function.

The new layers also add more hyperparameters, which need to be configured. The most obvious one is the size of the new layers. The same optimization technique as in the previous section was used to find the best value. The optimal value turned out to be 512 nodes in a layer. Adding extra nodes beyond this value hasn't improved the resulting metrics.

The following subsections will explore other new parameters. The loss function will also be discussed. It was of course present in the previous section as well, but its influence wasn't discussed.

4.3.2.1 Activation function

There is large number of potential activation functions, which can be used in the hidden layers. The rectified linear unit (ReLU) is one option, which is commonly used in modern networks.

There is a very active discussion about the properties of different functions, which can be used. One of the more recent comparisons can be found here [17]. Since no function is universally better and they are readily available in libraries, it is easy to run the experiment. The relevant results are demonstrated in graph 4.3.

The graph shows that most functions behave similarly on the data, except softmax. That is understandable, because softmax places specific extra constraints on the output of a layer. The goal of the softmax function is to ensure, that the whole output vector sums to one. This allows the output to be interpreted as probabilities. While that is an interesting property, which

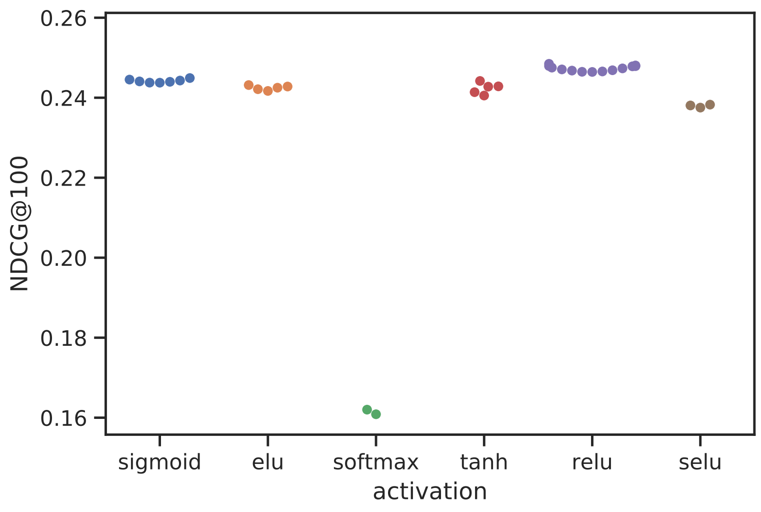


Figure 4.3: NDCG@100 for different activation functions in the hidden layer

can be interesting for usage in the output layer of many networks, it adds no value here.

ReLU achieved the best results (although by a small margin). Therefore, it will be used in future measurements.

4.3.2.2 Dropout

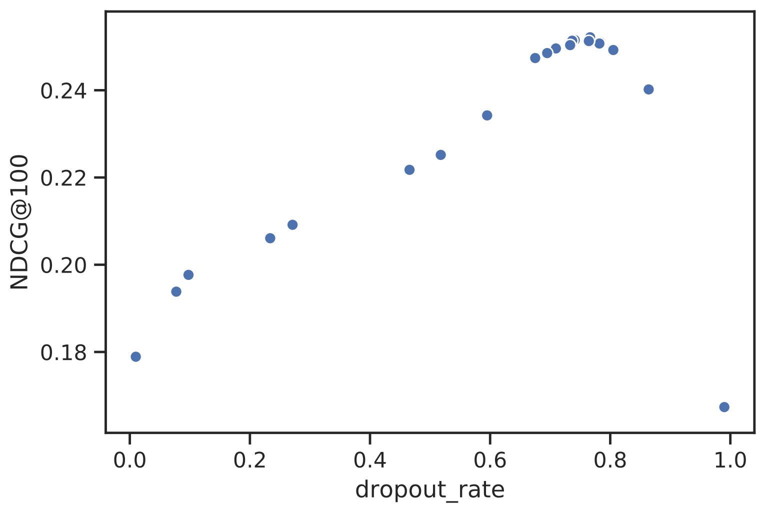


Figure 4.4: NDCG@100 for different values of dropout

Dropout can be applied to the output of each hidden layer. It causes a subset of the neurons in a layer to be left out from one forward and backward pass. The usage of dropout can help prevent over-fitting and improve the results.

Graph 4.4 shows different rates of dropout applied to models trained on the same dataset with other parameters fixed.

The expected pattern arises. Too small or two high values of dropout hurt the results. There is continuous improvement up to a certain value and then the performance drops again. The optimal value seems to be approximately 0.75.

4.3.2.3 Loss function

The loss function represents the error between predicted values of the network's output and the expected output. It should be chosen to fit the type of data. The graph 4.5 shows results on MovieLens dataset.

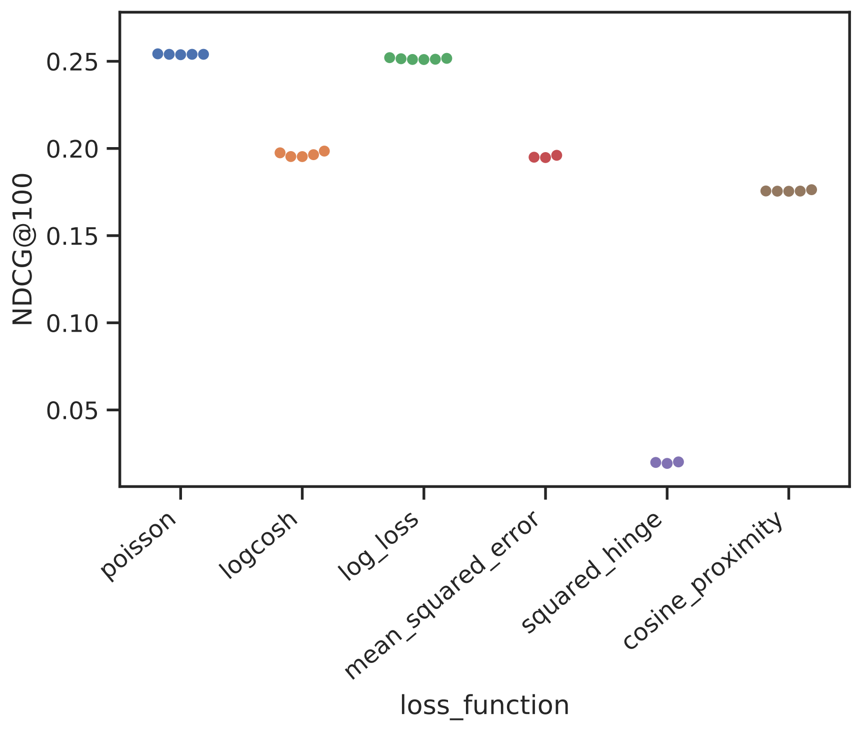


Figure 4.5: NDCG@100 for different loss functions

It is interesting to see that squared hinge loss function leads to very poor results. It is clearly not well suited for this problem. That is not surprising, since it is primarily used for binary classification in Support Vector Machines. It focuses on the largest difference between vectors, which is not well suited for long vectors of zeros and ones.

Other tested functions have performed much better. The best option seems to be either Poisson loss function or the log loss. Both are variants of measuring the difference of two statistical distributions. Since most experiments have already been using log loss, this will remain the default option.

4.3.2.4 Results

After all relevant parameters were optimized, new models were trained on both datasets. The performance has generally improved for all metrics by about 10%. The only exception is recall on Netflix dataset, which dropped slightly.

Dataset	Recall@20	Recall@50	NDCG@100	Coverage@50
MovieLens	0.28019	0.45730	0.25018	0.13520
Netflix	0.23253	0.38753	0.23734	0.20995

Table 4.3: Results of AE with one hidden layer applied to two datasets

In conclusion, it seems that adding one hidden layer has been beneficial to the ability of this type of model to generate predictions.

4.3.3 AE with multiple hidden layers

The last thing that should be attempted in order to improve to performance of the model is to add more hidden layers. This can be both beneficial and detrimental.

In many cases the extra layers allow the model to ultimately learn a better representation of the dataset. However, the complexity of the training process increases quickly and it might not even be realistic to train the model once it reaches certain level complexity.

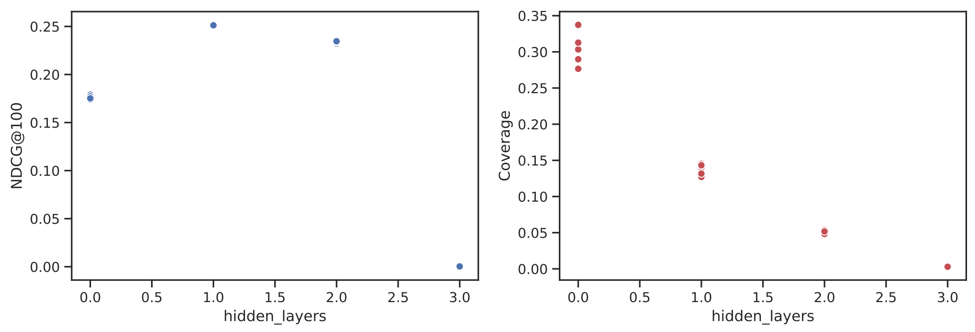


Figure 4.6: NDCG@100 and coverage for different number of hidden layers

Both effects are visible in graph 4.6. At first more hidden layers help the model. However soon the complexity of the training process wins over. In this experiment the model with three hidden layers (three in the encoder and three in the decoder) collapsed completely. It always recommended only a couple of items.

This could be improved, if special attention was paid to this model. For example, all layers in this experiment were of the same size. That is most likely not a good idea for a model of this complexity. Further tuning and long training could almost likely eventually make it competitive with the simpler models.

The graph shows one interesting property. Coverage only decreases. Since it tends to be negatively correlated to NDCG@100, that is understandable at the begging. Overall it seems that the models with more hidden layers were recommending fewer items even when NDCG@100 also dropped. That suggests, that using deeper model than necessary simply isn't good for the final recommendations.

4.3.4 L1 normalization in the latent layer

L1 normalization applied to the latent layer creates a sparse autoencoder. The loss function now contains an element, which leads the learning process to create latent vectors with as many zeros as possible.

It is possible to apply the normalization to every layer, but the results for the problem of recommendation appear to be the same.

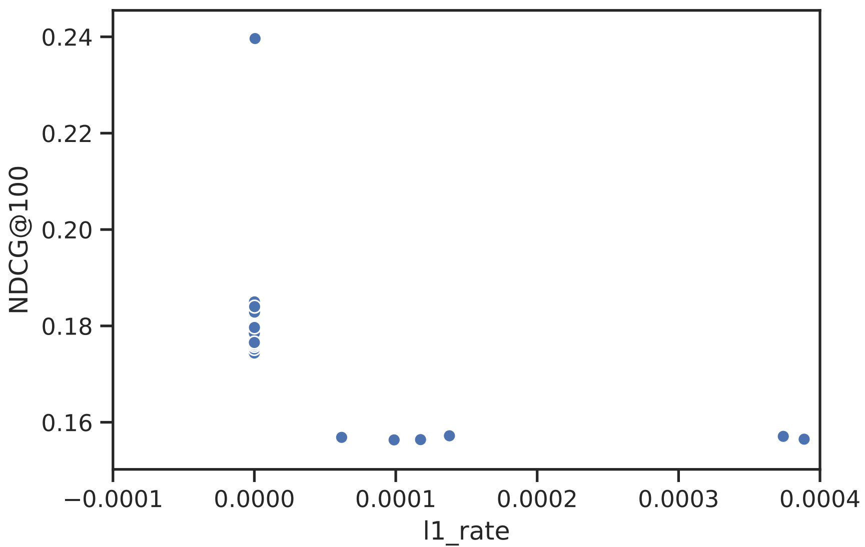


Figure 4.7: NDCG@100 for L1 normalization

This new element of the loss function has a weight associated with it called the L1 rate. Graph 4.7 shows the performance of models with different rates. The models used in this experiment had one hidden layer.

Very small values (around 10[−]7) lead to a performance comparable to models with no normalization. Higher values lead to performance degradation. L1 normalization doesn't seem to benefit models developed in this chapter.

The results are even worse, when dropout is applied to hidden layers of the network at the same time. This is likely because both methods are regularization methods and they ultimately interfere with each other. The same is true for the L2 normalization in the next section.

Even the best model in this section doesn't perform better than previous models with dropout only. One of the contributing factors may be the fact, that the size of the hidden layer has already been optimized. As a result, not many of the values in the latent vector can be zero without hurting performance.

4.3.5 L2 normalization in the latent layer

L2 normalization is an alternative regularization method. It doesn't produce sparse latent vectors, but it can be used to prevent over-fitting.

When added to model without dropout, the L2 normalization improves performance. The optimal values of the L2 rate appear to be around 0.0001. Graph 4.8 shows the performance of models with different values of the parameter.

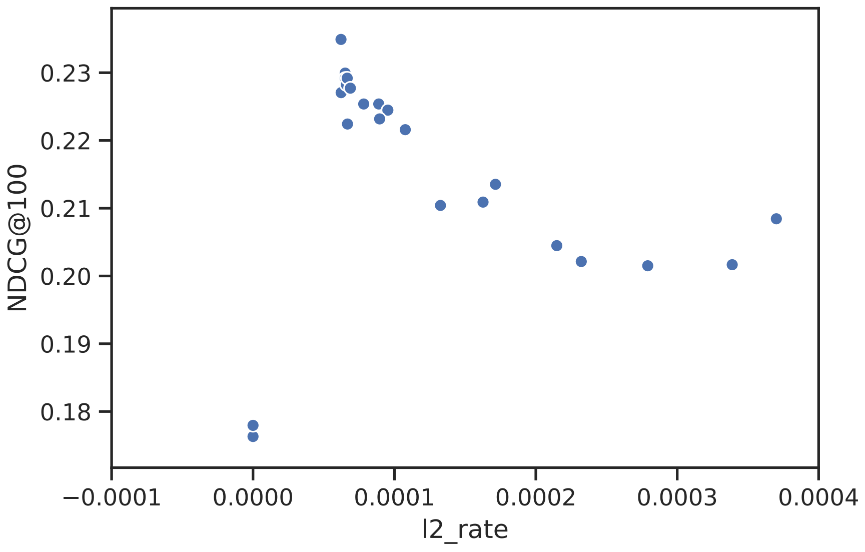


Figure 4.8: NDCG@100 for L2 normalization

The results of the best found models are summarized in table 4.4. The values of all the metrics are lower (by several percent) then the values of a similar model, which uses dropout instead of L2 normalization.

Dataset	Recall@20	Recall@50	NDCG@100	Coverage@50
MovieLens	0.26053	0.41577	0.23589	0.08142
Netflix	0.21857	0.36832	0.22358	0.24828

Table 4.4: Results of AE with l2 normalization rate 0.000062

This regularization method does improve performance, but overall doesn't outperform dropout when used for the same task.

4.4 Variational autoencoders

The basic idea of the variational autoencoder (VAE) was described in 1.3.2. Unlike the standard autoencoder, its learned latent representation represent parameters of a probability distribution. This means that despite a similar name, it is a very different model. This section will study its potential use for generating recommendations.

4.4.1 VAE with no hidden layer

The first evaluated version is the simplest one. Similarly to the experiments with the standard autoencoder, first a parameter search was performed to find the optimal values of the basic parameters. The optimal parameters seem to be:

- Number of epochs: 60
- Size of the latent vector: 32 (for each of the two vectors)
- Learning rate: 0.0015

When compared to the basic autoencoder, the biggest difference was the number of epochs required to train the best model. This is a result of the fact, that the basic variational autoencoder is already more complex model, that the simplest standard autoencoder. The loss function is more complex and the latent representation in larger.

In case of the standard autoencoder, the optimal size of the latent vector was also 32, but it was only one vector. In case of the variational autoencoder, it is one vector representing the mean of a multivariate normal distribution and second vector to represent the standard deviation.

Graph 4.9 shows the tested options for the size of latent vectors. Size around 30 seems to be ideal, but smaller vectors would likely work similarly. The size 32 was therefore chosen to correspond with previously performed experiments.

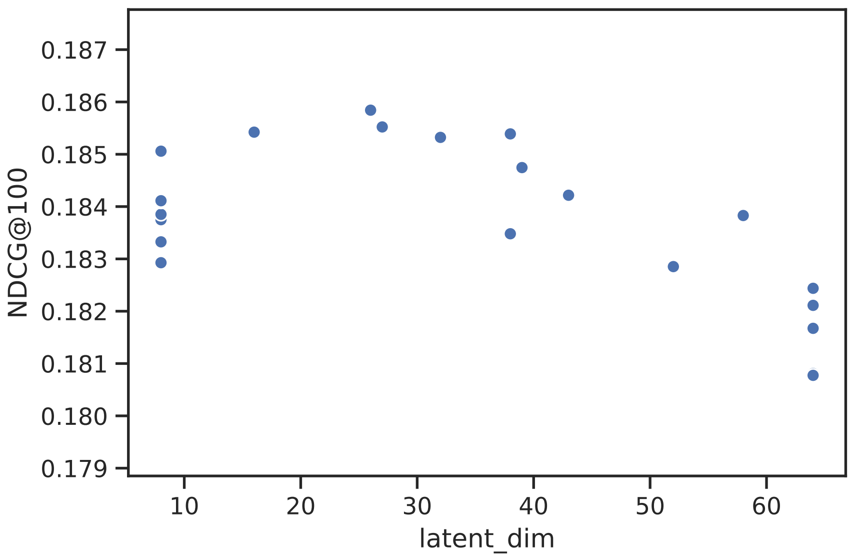


Figure 4.9: NDCG@100 for different size of the latent vectors (each component)

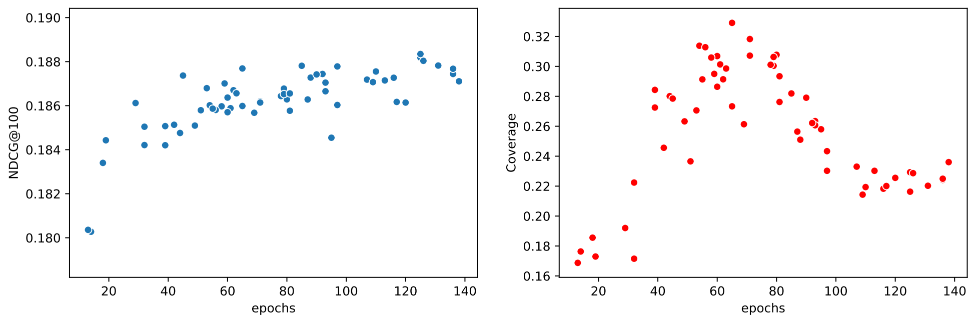


Figure 4.10: NDCG@100 and coverage for different number of epochs

Graph 4.10 shows the dependence of NDCG@100 and coverage on the number of epochs used to train the model. This dependence is interesting, because NDCG@100 and coverage tend to be negatively correlated.

The optimal value seems to be around 60. NDCG@100 value doesn't improve significantly past this point, but coverage begins to decrease significantly. This is most likely caused by over-fitting. Since generally the goal is to improve the value of coverage as well if possible, there is no point in training the model past approximately 60 epochs.

Table 4.5 shows the metrics for models trained on both datasets with the best parameters. The values are generally worse than for the standard autoencoder with no hidden layers. Only the values of coverage have improved.

Dataset	Recall@20	Recall@50	NDCG@100	Coverage@50
MovieLens	0.18608	0.31365	0.18549	0.24724
Netflix	0.14410	0.26022	0.17178	0.33873

Table 4.5: Results of VAE with no hidden layers applied to two datasets

4.4.2 VAE with one hidden layer

To improve potential performance of the model, a hidden layer can be added. This should allow the training process to generate a more complex model, but it also increases the training time significantly. In order to use the hidden layer efficiently, appropriate size needs to be selected.

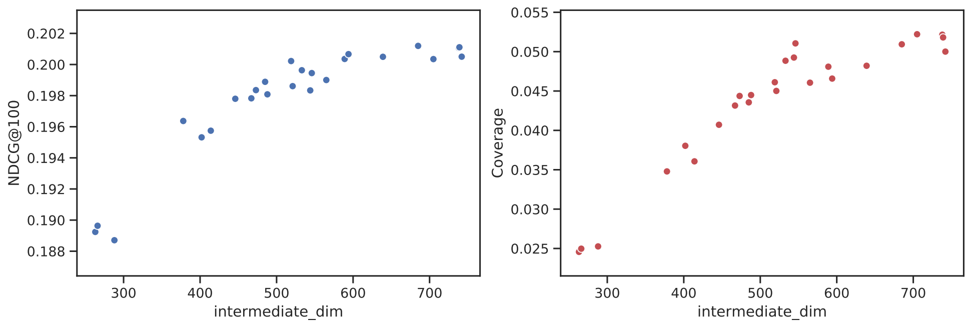


Figure 4.11: NDCG@100 and coverage for different number of neurons in the hidden layer

Graph 4.11 shows how NDCG@100 and coverage change with the number of neurons in the hidden layer. It is interesting to see both metrics growing with the number of neurons. It is much more common for them to be negatively correlated.

That observation suggests, that the models with higher number of neurons can predict user preferences better without over-fitting. For further experiments the size of the hidden layer was set to 640, because the improvements past this point don't warrant the increased time and complexity of training.

4.4.2.1 Activation function

As was already discussed, when appropriate activation function for the standard autoencoder was chosen, different neural networks can benefit from different choices. Since variational autoencoder can behave very differently from the standard one, the options should be evaluated separately.

Graph 4.12 shows the results of the experiment. ReLU was the best option for the standard autoencoder. In this case it also performed very well, but it was outperformed by sigmoid function. As a result, the sigmoid function will be used in further experiments.

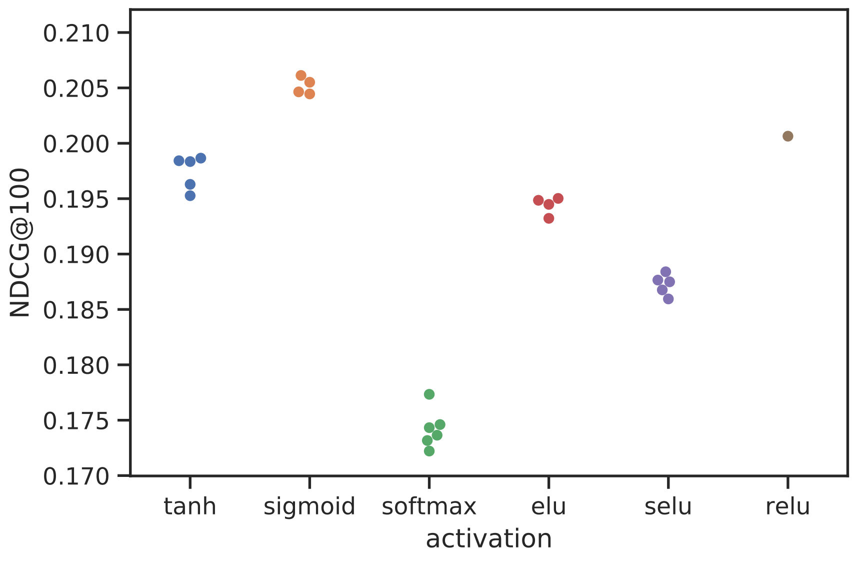


Figure 4.12: NDCG@100 for different activation functions in the hidden layer

4.4.2.2 Dropout

In keeping with the optimization methodology used for standard autoencoders, dropout was also experimented with. Unlike in the previous case, the best values (according to graph 4.13) seem to be relatively low. Since the NDGC@100 stays approximately the same as the dropout rate increases and coverage only decreases, the optimal value seems to be around 0.25.

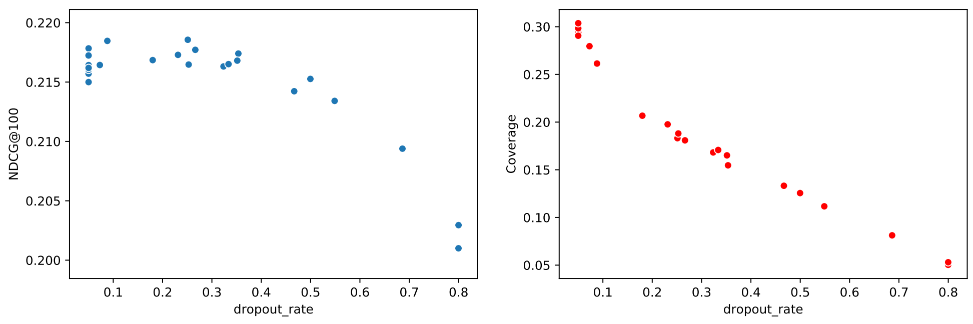


Figure 4.13: NDCG@100 and coverage for values of dropout

4.4.2.3 Loss function

The loss function is composed of two parts. One is called the reconstruction loss, which is the standard loss function penalising the difference between expected and actual output. The other is the Kullback–Leibler divergence. It leads the model to find a probability distribution, which matches the distribution of the underlying data. The loss function in graph 4.14 represents the reconstruction loss.

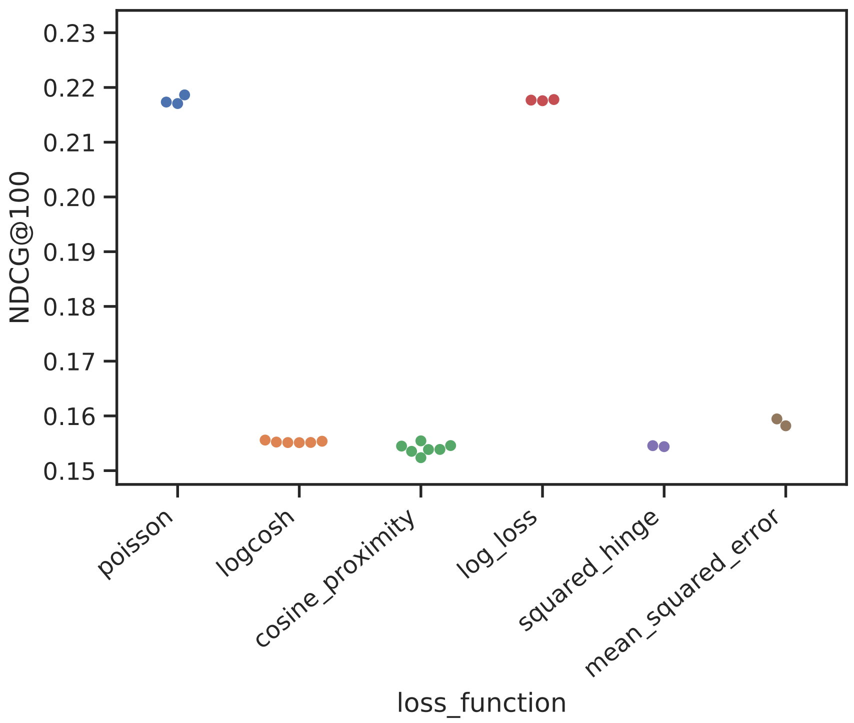


Figure 4.14: NDCG@100 for different loss functions

Loss functions behave very similarly in the case of the variational autoencoder and they did for the standard autoencoder. This seems reasonable, because loss function processes the expected and the actual output of the model. And those are the same for both models. Therefore, the choice of a loss function depends more on the data used for the experiments, then the models used.

Poisson and log loss function were the most successful in both cases. Interestingly enough they also produce models with high coverage compared to most other tested functions.

Paper [10] proposes interesting extension of the loss function of the variational autoencoder. It interprets the reconstruction loss as the loss function and the KL-divergence as a regularisation mechanism. This perspective then naturally leads to adding parameter β between the two components of the loss function, so that the extent of the regularisation can be regulated. Further reasoning behind this modification can be found in the paper itself.

Experimenting with this parameter leads to an interesting result. It is the

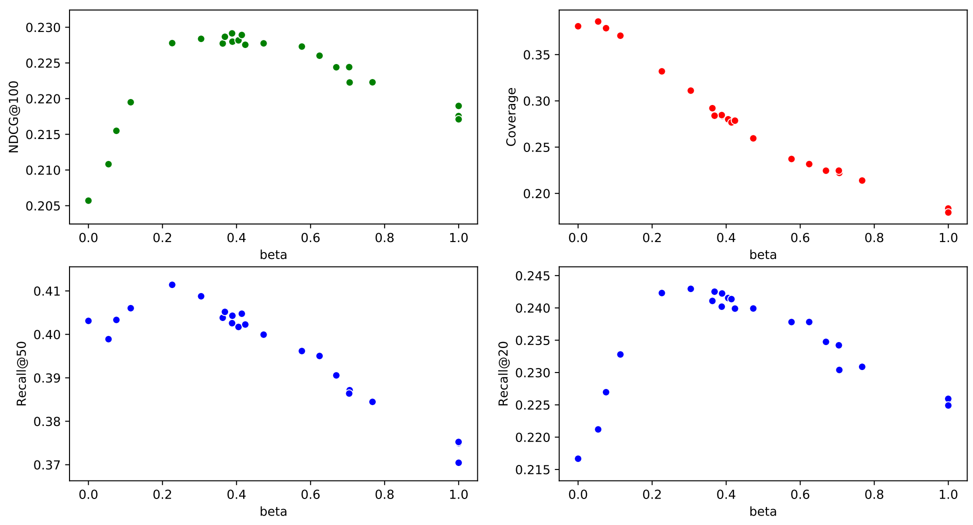


Figure 4.15: Metrics for different values of the β parameter of the loss function

first parameter in these experiments, which influences each of the used metrics differently. Even for Recall@20 and Recall@50 the correlation isn't as high as usual. The best value seems to be around 0.25. Ultimately, introducing this parameter improved the results by several percents.

4.4.2.4 Results

After the optimal parameters found in the previous section were used to train models on both datasets, the measured results are summed up it the following table 4.6.

Dataset	Recall@20	Recall@50	NDCG@100	Coverage@50
MovieLens	0.24138	0.40965	0.22654	0.32164
Netflix	0.20216	0.35102	0.21503	0.58889

Table 4.6: Results of VAE with one hidden layer applied to two datasets

The three metrics related to accuracy of the predictions are all consistently about 5-10 % worse than the best results of the standard autoencoder. However, coverage is significantly better. It is more than a double of the previous values.

This is an interesting result. It suggests that these two models could be used differently under different circumstances. In some use cases the accuracy of the predictions might be more important, but in some the slightly worse accuracy could be well compensated by the ability to recommend much larger subset of the overall catalogue.

4.4.3 VAE with multiple hidden layers

In some cases adding extra layers to a model can help improve performance. In section 4.3.3 it was shown, that in the case of the standard autoencoder that wasn't true. The same experiment was performed with the variational autoencoder.

Dataset	Recall@20	Recall@50	NDCG@100	Coverage@50
MovieLens	0.25658	0.42078	0.23977	0.29387
Netflix	0.20855	0.35151	0.22172	0.49630

Table 4.7: Results of VAE with two hidden layers applied to two datasets

Table 4.7 shows the results for two hidden layers in the encoder and decoder. The accuracy related metrics were improved by a few percent compared to one hidden layer.

As could have been expected, coverage drops by several percents. These results are very interesting. The accuracy is approaching the best results of the standard autoencoder, but the coverage is significantly higher. This means there is a real potential for trade off based on the demands for specific use case.

Adding more layers leads to a significant decrease in all metrics. It is possible that further tuning of these complex models would make them competitive, but the potential added value appears to be limited.

4.5 Latent space visualizations

This section presents visualizations of the latent space for both types of models described in the previous sections. The visualizations have been created by training a model and generating the latent vectors for each input vector in the test set.

The latent vectors are high-dimensional and need to be projected to a 2D space in order to be plotted. Visualizations in this section have been created using the t-SNE [18] method.

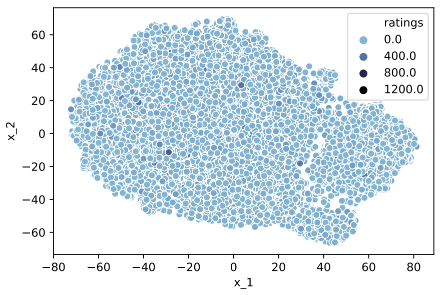
One of the main goals of this method is to place vectors, which are close in the original high-dimensional space, close in the new low-dimensional space. That means, that if vectors tend to cluster in the original space, they should also cluster in the new space.

4.5.1 User-based models

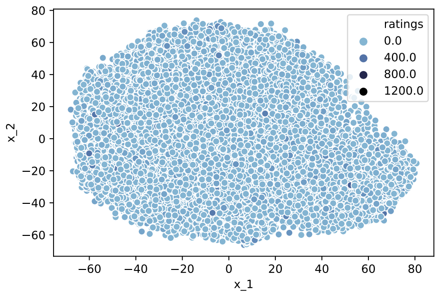
The following visualizations show latent representation of the user ratings vectors from the MovieLens dataset. The models used in this section have been trained with the best parameters found in the previous sections.

The dataset doesn't contain any information about the users. That is why the visualizations only show the number of ratings created by a given user.

4.5.1.1 AE



(a) Standard autoencoder



(b) Variational autoencoder

Figure 4.16: Projections of latent space representation of users and their number of ratings

The latent space of the autoencoder doesn't seem to contain any clearly defined clusters. There is one subset of the latent vectors on the right side of the visualization 4.16a, which seems partially separated from the rest.

This potential cluster doesn't exhibit any straightforward reason for being distinct. The users in this part of latent space have created about the same number of ratings as the rest.

There could be a certain subset of movies, which these users have rated significantly differently from the rest. However, the original user ratings vectors don't exhibit any obvious differences from the rest.

When PCA was applied to the original high-dimensional data instead of t-SNE, no clearly defined clusters appeared. That means that either the subset visible in 4.16a forms more complex manifold, which PCA didn't transform well into the low-dimensional space, or it is a property of the t-SNE algorithm itself.

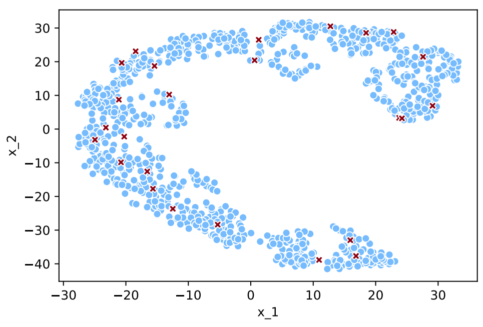
4.5.1.2 VAE

The visualization 4.16b of the latent space of the variational autoencoder differs from the previous one in the general shape. The latent vectors are ”huddled” together.

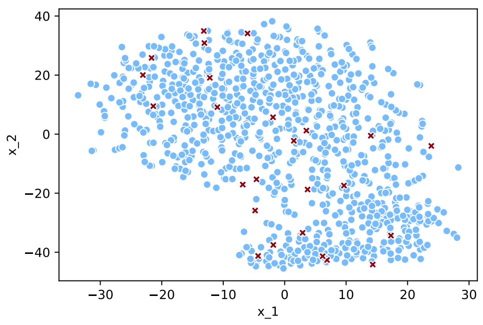
This is commonly seen in visualizations of the latent space of variational autoencoders. It can be explained in the following way. If only one input vector was sent through the network repeatedly, the resulting latent vectors would be normally distributed. This would plot as a circular cluster with more vectors near the centre.

If several similar vectors were used as the input, it would lead to multiple clusters laid over each other with slightly different centres. This effect partially persists when more and more input vectors are used. That is why the visualizations tend to be more circular.

4.5.2 Movie-based models



(a) Standard autoencoder



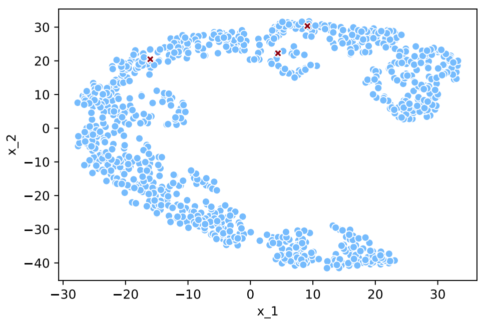
(b) Variational autoencoder

Figure 4.17: Projections of the latent space representation of movies. Red crosses represent the Animation genre.

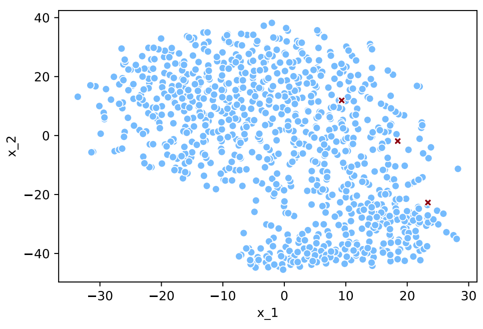
Models in this thesis were focused on the user ratings vectors. However, the MovieLens dataset doesn't contain any information about the users, but it does contain some information about movies – their titles and genres.

This section presents visualizations of the latent space for the movie vectors. The input matrix has been inverted. The models used to generate the visualizations haven't been tuned nearly as well as the ones in the previous section, but they did produce some interesting patterns.

4.5.2.1 AE



(a) Standard autoencoder



(b) Variational autoencoder

Figure 4.18: Projections of the latent space representation of movies. Red crosses represent the Three Colours series.

The first visualization for the standard autoencoder (4.17a) highlights animated movies. Similar patterns could be shown for all the other genres. This could have been expected, because it is not typical for consumers to choose movies strictly based on genres.

The hope was that animated movies are much more popular among children and their pattern of ratings could be different. This may or may not be true, but children likely use their parents' accounts and therefore the ratings would be mixed with others anyway.

The second visualization (4.18a) highlights a trilogy of movies named Three Colours. This result does match the expectation. Their corresponding latent vectors are close together in the latent space, because users are likely to watch all of them or none.

4.5.2.2 VAE

Similarly to visualizations of the user vectors, the variational autoencoder produces more circularly arranged latent vectors. The situation here is very similar to the standard autoencoder.

The animated movies don't form any distinct cluster. However, the latent vectors of the Three Colours trilogy are again relatively close to each other.

4.6 Discussion of results

The previous sections have focused on improving two basic models (standard and variational autoencoder) as much as possible. For each model a configuration was found, which produced the best results.

This "simplicity" of finding the clearly "best" configuration for each model may be partially caused by the limitations of the metrics used to evaluate the results. As was discussed earlier, recall and NDCG are usually highly correlated.

Coverage tends to be inversely correlated. At least for very similar models. Seemingly simple conclusion could be the following: As accuracy of the predicted ratings increases, the number of recommended items decreases.

However, this is only true for very similar models. When the configuration is changed more significantly (for example new hidden layer is added), all metrics can improve together. This change can then be considered clear improvement (within the limited relevancy of these metrics).

The following tables also include matrix factorization results.

Model	Recall@20	Recall@50	NDCG@100	Coverage@50	Time (hh:mm)
SVD	0.16036	0.30237	0.15459	0.04686	39:16
AE	0.28019	0.45730	0.25018	0.13520	00:19
VAE	0.25658	0.42078	0.23977	0.29387	00:35

Table 4.8: Results of the best models for MovieLens dataset

Comparison of the models used on the MovieLens dataset shows, that the autoencoders outperform matrix factorization. It is important to point out, that the SVD method wasn't tuned nearly as much as the autoencoders. Its result could likely be improved.

The most interesting part of the comparison with SVD is very small coverage demonstrated by SVD. This is an example of how all metrics can increase or decline together, when the models are significantly different.

Between the standard and variational autoencoder, the recall and NDCG are similar. The most interesting part is the more than twice as large coverage generated by the variational autoencoder.

That strongly suggests that in practice the user experience of these two models would be very different. The variational autoencoder would likely recommend significantly larger range of items.

Table 4.8 also shows the time it took to train each model. SVD is a time-consuming operation especially with a large matrix. Standard autoencoder is simpler to train than variational autoencoder and it also requires fewer epochs.

Times for the Netflix dataset (4.9) reflect the fact, that the dataset is larger. The SVD was faster, because it used only a fraction of the entire dataset. The full dataset would be too memory-consuming.

Model	Recall@20	Recall@50	NDCG@100	Coverage@50	Time (hh:mm)
SVD	0.17015	0.31099	0.17333	0.05984	06:19
AE	0.23253	0.38753	0.23734	0.20995	00:51
VAE	0.20855	0.35151	0.22172	0.49630	01:44

Table 4.9: Results of the best models for Netflix dataset

Results of all three models applied to Netflix dataset lead to the same conclusions. This supports the results. The observed differences are very likely caused by the models themselves.

Conclusion

During the work on this thesis, I have surveyed the relevant literature and analysed potential algorithms for generating recommendations. Based on the state-of-the-art literature, I have chosen the most promising models.

I have prepared data for evaluation using publicly available datasets and chosen the metrics, which allowed me to analyse behaviour of different models.

I have implemented the two basic types of models and experimented with them to improve their performance. By the end both models reached comparable performance, but the specifics of their generated recommendations were different.

Standard autoencoder is easier to train and becomes accurate in predicting user preferences. It learns faster and requires fewer computational resources. However, ultimately variational autoencoder can be trained to a similar level of accuracy, but it also maintains the ability to recommend much larger fraction of the catalogue.

This, in my opinion, makes it a better choice for many use cases, because the larger variability of the recommended items increases the chance, that some of the items will be interesting to the user.

In order to gain better understanding of how the two models differ in realistic situations, it would be beneficial to perform A/B testing with a real online service and real users.

This thesis has focused on the problem of processing the user-item ratings matrix, specifically on implicit feedback. I believe that in practical application it could be very interesting to incorporate other available data about users and items into the recommendation process.

Neural networks provide a good framework for integrating different types of data. More complex networks or an ensemble of different models could be used for this purpose.

Bibliography

[1] Ghosh, S. Simple Matrix Factorization example on the Movielens dataset using Pyspark. `https://medium.com/@connectwithghosh/simple-matrix-factorization-example-on-the-movielens-dataset-using-pyspark-9b7e3f567536`, [cit. 2019-02-04].

[2] Axon. Jan 2019. Available from: `https://en.wikipedia.org/wiki/Axon`

[3] Spinner, T.; Körner, J.; et al. Towards an Interpretable Latent Space – An Intuitive Comparison of Autoencoders with Variational Autoencoders. Sep 2018. Available from: `https://thilospinner.com/towards-an-interpretable-latent-space/`

[4] Scrapehero. How Many Products Does Amazon Sell Worldwide - January 2018. Feb 2018. Available from: `https://www.scrapehero.com/how-many-products-amazon-sell-worldwide-january-2018/`

[5] Aggarwal, C. C.; et al. *Recommender systems*. Springer, 2016.

[6] 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*, CSCW '94, New York, NY, USA: ACM, 1994, ISBN 0-89791-689-1, pp. 175–186, doi: 10.1145/192844.192905. Available from: `http://doi.acm.org/10.1145/192844.192905`

[7] Netflix Prize homepage. `http://www.netflixprize.com/index.html`, [cit. 2019-02-03].

[8] Töscher, A.; Jahrer, M.; et al. The bigchaos solution to the netflix grand prize. *Netflix prize documentation*, 2009: pp. 1–52.

[9] Rehorek, T.; Kordik, P.; et al. Comparing Offline and Online Evaluation Results of Recommender Systems. In *In Proceedings of RecSyS conference (RecSyS'18)*, New York, NY, USA: ACM, 2018.

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

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

[12] Russell, S. J.; Norvig, P. *Artificial intelligence: a modern approach*. Prentice Hall, 2010.

[13] Ng, A.; et al. Sparse autoencoder. *CS294A Lecture notes*, volume 72, no. 2011, 2011: pp. 1–19.

[14] MovieLens. Jan 2019. Available from: `https://grouplens.org/datasets/movielens/`

[15] Frazier, P. I. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[16] Scikit-Optimize library. Available from: `https://scikit-optimize.github.io/`

[17] Pedamonti, D. Comparison of non-linear activation functions for deep neural networks on MNIST classification task. *CoRR*, volume abs/1804.02763, 2018, `1804.02763`. Available from: `http://arxiv.org/abs/1804.02763`

[18] Maaten, L. v. d.; Hinton, G. Visualizing data using t-SNE. *Journal of machine learning research*, volume 9, no. Nov, 2008: pp. 2579–2605.

Appendix A

Contents of CD

```
README.md..........................the file with CD contents description
 data.............................................the data files directory
   results................the examples of recorded parameter searches
     *.csv .......................................... the stored results
   datasets................the preprocessed datasets as sparse matrices
 src.....................................the directory of source codes
   python..............................the directory with python code
   thesis...............the directory of LaTeX source codes of the thesis
     documents ........................ the thesis documents directory
     images ..............................the thesis figures directory
     *.tex .................... the LaTeX source code files of the thesis
 text ......................................... the thesis text directory
   DP_Bajer_Michal_2019.pdf.......the Diploma thesis in PDF format
```