Scalable Computing Systems Laboratory
EPFL IC IINFCOM SACS
BC 162 (Bâtiment BC)
Station 14
CH-1015 Lausanne

Master Semester Project

Dynamic Neural Recommender System

Alexander Glavackij

Course of Study: Computer Science

Examiner: Prof. Dr. Anne-Marie Kermarrec

Supervisor: Dr. Othmane Safsasi

Commenced: September 21, 2021

Completed: January 07, 2022

Abstract

Today's recommenders need to work on dynamic datasets, i.e., datasets where users, items, and ratings change frequently. Our main contribution is a dynamic recommender system that periodically recommends a subset of unseen items to all users in a user base without retraining. Our second contribution is the evaluation of techniques to decrease catastrophic forgetting in neural networks. Our work builds on the work done by Sedhain et al. [SMSX15], a neural Autoencoder that takes a vector of all user ratings for one item and reconstructs it. The Autoencoder imputes missing ratings by using a learned latent representation of the users' feedback behavior. To learn the hidden representation that permits imputation, we implement a custom training procedure in TensorFlow that only considers observed entries in the input and does partial forward- and backward propagation. Our approach achieves an Root Mean Squared Error (RMSE) of 0.86 on the MovieLens-1M dataset [Mov21] and of 0.77 on the Microsoft MIND dataset [WQC+20]. Out of the techniques aimed at decreasing catastrophic forgetting, replaying old training samples emerged as the most effective technique.

Contents

1 Introduction **8**
1.1 Motivation for Recommender Systems . . . 8
1.2 Challenges in Recommender System . . . 8
1.3 Existing Methods in Recommendation . . . 10
1.4 Our Contribution . . . 11
1.5 Key Findings . . . 12

2 Related Work **13**
2.1 Neural Approaches to Recommendation . . . 13
2.2 Catastrophic Forgetting . . . 14

3 Implementation **16**
3.1 Implementation of our Autoencoder . . . 16
3.2 Implementation of Partial Training in Keras . . . 17

4 Evaluation **21**
4.1 MovieLens Dataset . . . 21
4.2 MIND Dataset . . . 27

5 Conclusion and Outlook **33**

Bibliography **35**

List of Figures

1.1 Item-user matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ showing ratings between 1 and 5 given by $n = 6$ users to $m = 6$ items and the factor matrices $\mathbf{W}$ and $\mathbf{Z}^\mathbf{T}$. Not every user gave a rating to every item, thus, some entries of the matrix are empty, i.e. unobserved. Matrix factorization factorizes $\mathbf{X}$ into $\mathbf{W} \in \mathbb{R}^{n \times k}$ and $\mathbf{Z} \in \mathbb{R}^{m \times k}$. The latent dimension is $k = 2$. 9
1.2 Autoencoder architecture proposed in [SMSX15]. Input is a row of the item-user matrix with all ratings for one item. Edges to unobserved entries (e.g. $u_{in,2}$) in the input are deactivated (dotted arrows), their weights are not updated during training. Effectively, every input has its own neural network, with m total neural networks in total (indicated by plate notation $1 \dots m$). 10
1.3 The problem setting of this work. We recommend q items per day per user out of m' items arriving every day to n users. 11

2.1 Xiangnan et al. [HLZ+17] propose to replace the dot product in Matrix Factorization (MF) with a neural network to model non-linear interactions of user and item representations. 14
2.2 Figure adapted from [KPR+16]. Grey and cream indicate parameter regions leading to good performance for Task A and Task B. θ_A denotes the parameters of the learner after training on Task A. By constraining the parameters in θ_A, the learner finds parameters that optimize performance on both datasets. 15

3.1 Architecture of our Autoencoder. 16
3.2 We implemented a dummy Autoencoder to illustrate our implementation. Figure 3.2a shows the architecture of the dummy Autoencoder: number of users is $n = 10$, the hidden layer size is $k = 5$. The dotted arrows are deactivated weights. Figure 3.2b shows the circled weights and their respective gradients in Keras. The gradients of deactivated weights (in our example weight row 0, 2, and 5 in the middle plot in 3.2b) are 0, showing that our implementation works. 20

4.1 Screenshot of the format of the MovieLens dataset. 21
4.2 Training and validation loss of the Autoencoder on the MovieLens-1M dataset using the Trade-off split. Final validation RMSE is around 0.86 which is state-of-the-art performance. 23
4.3 RMSE on the test set with 100 movies arriving every day, which is the problem setting of this work. The movies in the dataset are not ordered chronologically; thus, no preference change is visible, indicated by an increasing RMSE on the test set. 24
4.4 The trade-off for number of epochs spent on retraining on the new 100 samples versus the RMSE on training, test, and validation set. The RMSE on the training and validation set increase very quickly, while decreasing very quickly on the test set. This indicates massive overfitting on the new 100 test samples. 25

4.5 Retraining on 1,000 instead of 100 new samples reduces the overfitting seen in Figure 4.4. 25
4.6 Learning-rate versus the RMSE on training, test, and validation set during retraining on the new 100 samples. The RMSE on training and validation set increase gradually, while first decreasing and then increasing on the test set. Note the logarithmic scale. 26
4.7 In the replay trade-off, the RMSEs on the training and validation set decrease monotonically, while increasing monotonically on the test set, which suggests less overfitting and more generalizability. 26
4.8 Screenshot of the format of the dataset. Each user has a history of read articles. Each impression has a timestamp, a user who saw the impression, and the articles shown in the impression. 27
4.9 Training and validation loss of the Autoencoder on the MIND dataset using the Trade-off split. Final validation RMSE is around 0.77. 29
4.10 RMSE on the MIND Forgetting test set with 100 movies arriving every day. The RMSE increases on the test set as the days go by, suggesting that there is a preference shift in the dataset. In this case, it would make sense to retrain the Autoencoder to capture this shift. 30
4.11 The trade-off for number of epochs spent on retraining for the MIND dataset. The RMSE on the training set increases very quickly while decreasing very quickly on the validation and test set. Patterns learned on the test set translate to the validation set and reduce its RMSE. However, overfitting still occurs, indicated by the increasing training RMSE. 31
4.12 The learning-rate trade-off on the MIND dataset. A learning rate of 1×10^{-3} seems to be optimal to incorporate new patterns from the test set. Higher learning rates lead to underfitting on all datasets; at lower learning rates, the Autoencoder does not learn. Note the logarithmic scale. 32
4.13 In the replay trade-off, the RMSEs on the test and validation set increase, while decreasing monotonically on the training set, which suggests less overfitting and more generalizability. 32

List of Tables

3.1 Overview of the implementation of the Autoencoder in Keras. 17

4.1 Hardware and OS Parameters. 22
4.2 Autoencoder Evaluation Parameters for the MovieLens-1M dataset. 22
4.3 The two different Train/Test splits of the MovieLens-1M dataset we evaluate in this work. 22
4.4 Each technique and the values we analyze in this work for the MovieLens-1M dataset. The values for the Replay Trade-off refer to the fraction of the training data replayed during retraining on new samples. When not varied, epochs, learning rate, and replay fraction were kept at 5, 1×10^{-2}, and 0 respectively. 23
4.5 Autoencoder Evaluation Parameters for the MIND dataset. 28
4.6 The two different Train/Test splits of the MIND dataset we evaluate in this work. . 28
4.7 Each technique and the values we analyze in this work for the MIND dataset. The value ranges did not change compared to the MovieLens dataset, except for the Epoch Trade-off, where we go up to 100 epochs maximum. 29

Acronyms

EWC Elastic Weight Consolidation. 15

ISM Item Similarity Model. 9

MF Matrix Factorization. 4

MSE Mean Squared Error. 18

RMSE Root Mean Squared Error. 2

1 Introduction

The rise of the Internet and online services profoundly impacted our daily lives. Today, we can find a product that exactly fits our preferences from thousands of others that do not. Deciding between thousands of items would be impossible without Recommender Systems. They help users choose from large sets of products or services: products on Amazon, movies on Netflix, or songs on Spotify.

1.1 Motivation for Recommender Systems

Recommender systems recommend *items* to *users* based on their preferences. Users make their preferences visible by providing feedback on items. Service providers collect this feedback in numerous ways - e.g., by logging clicks, or ratings of an item, or the time spent on a website.

Well-built recommender systems are essential for a business's success - e.g., they increase the Click-through-rate of online newsletters or the sales of products. Some companies are even known for their addictive recommender systems, e.g., TikTok. Furthermore, their positive effect on businesses is easily measurable by A/B-testing, another reason for their popularity.

1.2 Challenges in Recommender System

While providing value to businesses, recommender systems also generate two general issues for their users: biases and security issues. Their recommendations might suffer from several biases: amplification bias, and dataset biases to name two examples. The recommendations of systems with an amplification bias lose in diversity as the system learns user preferences [MAP+20], which can lead to echo chambers, a phenomenon criticized by the public [Gri17]. Recommender systems might also amplify biases of the dataset they are trained on [ETK+18].

Privacy issues are inherent to recommender systems as they work on collected user data. The more data the system has on a given user, the more relevant its suggestions are - thus, companies are eager to collect more and more information on users to profile them accurately. However, this is at odds with user privacy. It has been shown that recommender systems can reveal private information about users [RKM+01] [AÁK12].

Recommender systems can be segmented in different ways. In this work, we choose to distinguish between static and dynamic recommender systems. Static systems make recommendations on static datasets, i.e., datasets that change very little. Even though the datasets are static, these systems still face challenges. The amount of data available increases day by day; nevertheless, they need to train

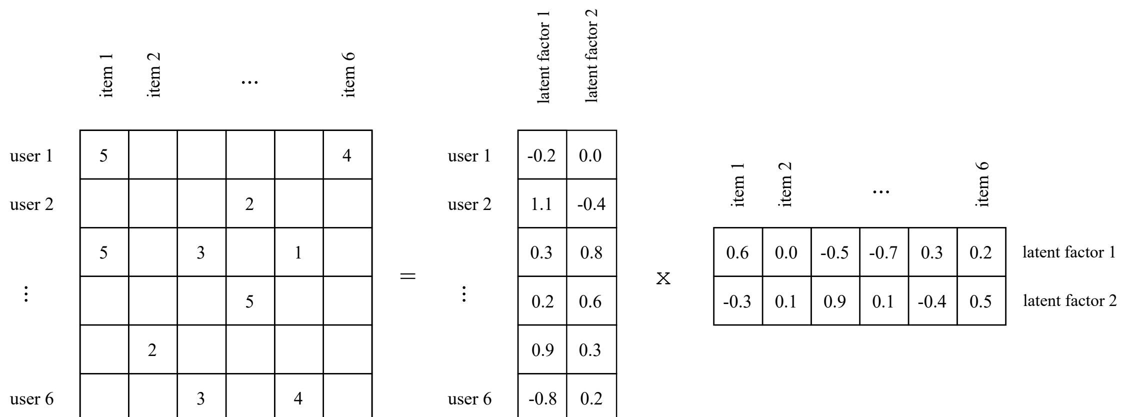


Figure 1.1: Item-user matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ showing ratings between 1 and 5 given by $n = 6$ users to $m = 6$ items and the factor matrices $\mathbf{W}$ and $\mathbf{Z^T}$. Not every user gave a rating to every item, thus, some entries of the matrix are empty, i.e. unobserved. Matrix factorization factorizes $\mathbf{X}$ into $\mathbf{W} \in \mathbb{R}^{n \times k}$ and $\mathbf{Z} \in \mathbb{R}^{m \times k}$. The latent dimension is $k = 2$.

in a reasonable amount of time. An example of a static dataset is the Netflix prize [KBV09]: all users and items are known beforehand; thus, the system only recommends known movies to known users.

The datasets used in recommender systems can change in three dimensions: new users or new items can be added, or a user's feedback for an item changes. User preferences change slowly; therefore, when referring to dynamic datasets, we refer to datasets where new items and new users are added frequently.

Dynamic recommender systems operating on dynamic datasets bring their own set of challenges. First and foremost, they need to make recommendations despite a lack of data - the system has no prior knowledge of new users or items, which renders making relevant recommendations difficult. Second, dynamic systems need to quickly incorporate new item and user preferences - i.e., they need to retrain on new data quickly. Third, they need to make their recommendation fast; users should not wait for them. An example of a dynamic setting is news recommendation. New articles are written every day and need to be recommended to an ever-changing user base.

Today's systems need to increasingly work on dynamic datasets. The amount of data collected increases every day as more and more users use more and more online services. Thus, systems need to handle a constant influx of new and changing data - Netflix alone gained 18.2 mill. new subscribers [Sto21a] and released around 370 movies in 2019 [Sto21b]. Previous techniques used for recommendation - e.g., Matrix Factorization (MF) [KBV09] or Item Similarity Models (ISMs) [LSY03] - operate only on static datasets, i.e., items need to be included in a training phase to be recommended. Therefore, the research community set out to develop novel methods that assume datasets' dynamic properties.

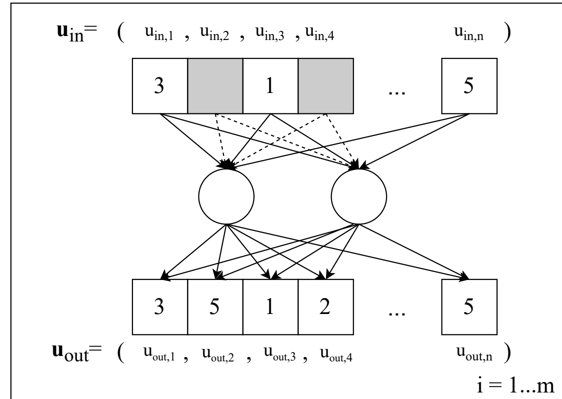


Figure 1.2: Autoencoder architecture proposed in [SMSX15]. Input is a row of the item-user matrix with all ratings for one item. Edges to unobserved entries (e.g. $u_{in,2}$) in the input are deactivated (dotted arrows), their weights are not updated during training. Effectively, every input has its own neural network, with m total neural networks in total (indicated by plate notation $1 \dots m$).

1.3 Existing Methods in Recommendation

Orthogonally to the segmentation of recommender system presented in the previous section, [Agg16] makes the segmentation by the kind of data that is used to make recommendations: item-user interactions, such as ratings, and attribute information about items and users, such as user profiles or item descriptions. Approaches belonging to the former category are called *collaborative filtering* methods, whereas methods of the second category are referred to as *content-based filtering* methods. Collaborative filtering methods leverage item-user interactions data to find patterns and recommend items [Agg16]. We can view the users' feedback on items in a item-user matrix, cf. Figure 1.1. These matrices are sparse: users typically only provide feedback on a small fraction of available items. Thus, the majority of item-user combinations are *unobserved*. Collaborative filtering methods assume that some pattern in the observed ratings can be exploited to impute unobserved interactions.

One method that has proven to be particularly useful is *Matrix Factorization* (MF) [KBV09]. It factorizes the item-user matrix shown in Figure 1.1 into two matrices: $\mathbf{X} = \mathbf{W} \cdot \mathbf{Z}^\mathbf{T}$, where $X \in \mathbb{R}^{n \times m}$, $W \in \mathbb{R}^{n \times k}$, $Z \in \mathbb{R}^{m \times k}$. $\mathbf{W}$ and $\mathbf{Z}$ represent users and items in latent spaces of dimension k. High correspondence of item and user factors leads to a high rating, and thus, a recommendation. However, there is one particular problem arising with MF: inclusion of new users or items. For every new item matrix $\mathbf{Z}$ needs to be expanded by one row and retrained; the same holds for a new user and $\mathbf{W}$. Retraining becomes computationally expensive if the dataset is dynamic. Additionally, new items and users lack ratings, MF cannot calculate the latent vectors without any ratings. Making relevant recommendations is difficult in this case. This phenomenon is referred to as the *Cold-Start problem*.

The recent successes of deep-learning methods prompted researchers to employ them in recommender systems. One fundamental approach for this work, presented by Sedhain et al. [SMSX15], uses an *Autoencoder* to impute missing ratings. The input to the Autoencoder is a row- or column vector of the item-user matrix, and the output is the reconstructed vector. Autoencoders exhibit

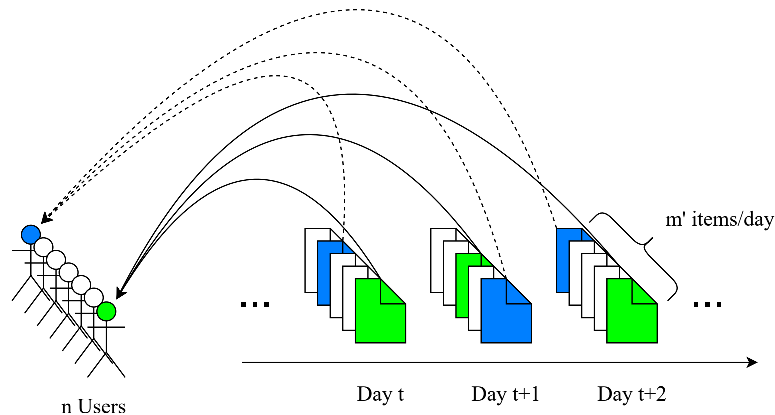


Figure 1.3: The problem setting of this work. We recommend q items per day per user out of m' items arriving every day to n users.

specific properties that are useful for recommendation. Much like MF, they project their input into a lower dimension, thereby learning a lower-dimensional representation of it. However, vanilla neural networks cannot handle the unobserved entries in the input. This is the key contribution of Sedhain et al. [SMSX15], cf. Figure 1.2. For every input, edges associated with unobserved entries are deactivated, such that they do not contribute to training. Thus, only weights associated with active edges are trained. Effectively, every input gets its personal neural network. Even though not all input values are observed, using the latent representation, they still produce a prediction for those values as an output.

1.4 Our Contribution

Our main contribution is a dynamic recommender system that periodically recommends a subset of unseen items to the complete user base. This setting stems from news article recommendation in newsletters. New articles are written and published every day. Daily newsletters advertise a selected subset of these new articles to subscribed readers. The selection of which news articles to include in the newsletters is essential for a high readership of new articles. Thus, online news publishers personalize each newsletter to its receiver to preferably advertise articles that could interest the reader. This work assumes that m' items are added to the dataset every day. Every day, we need to recommend q of these unseen items to every user of a stable user base. We assume that some users interact with new items between recommendations. Thus, there is little user feedback available for new items, which we can use to make recommendations.

Our second contribution is the evaluation of techniques to decrease catastrophic forgetting in neural networks. We retrain our recommender system periodically on new items to capture user preference drift over time. Neural networks are prone to catastrophic forgetting: when retrained on new data, they perform much worse on old data. We present different techniques to combat forgetting, their effectiveness, and their trade-offs.

1.5 Key Findings

We evaluate our autoencoder approach on two datasets: MovieLens-1M [Mov21], achieving a state-of-the-art RMSE of 0.86, and MIND [WQC+20], achieving an RMSE of 0.77 on *unseen* items. Out of the techniques to reduce catastrophic forgetting, replaying old training data proved the most effective with a minor training time penalty.

To develop and evaluate our recommender system, we employ the following structure: In the next chapter, we present relevant research to this work. Here, we show papers implementing neural approaches for recommendation and how their approaches differ from ours. Then, we give the implementation of our recommender system, showing how we implement the training to only consider observed values in the input and our evaluation setup. Next, we evaluate our system on multiple datasets. This chapter presents the accuracy results of our approaches and the influence of retraining strategies on catastrophic forgetting. Lastly, we conclude our work and give a brief outlook on future work.

2 Related Work

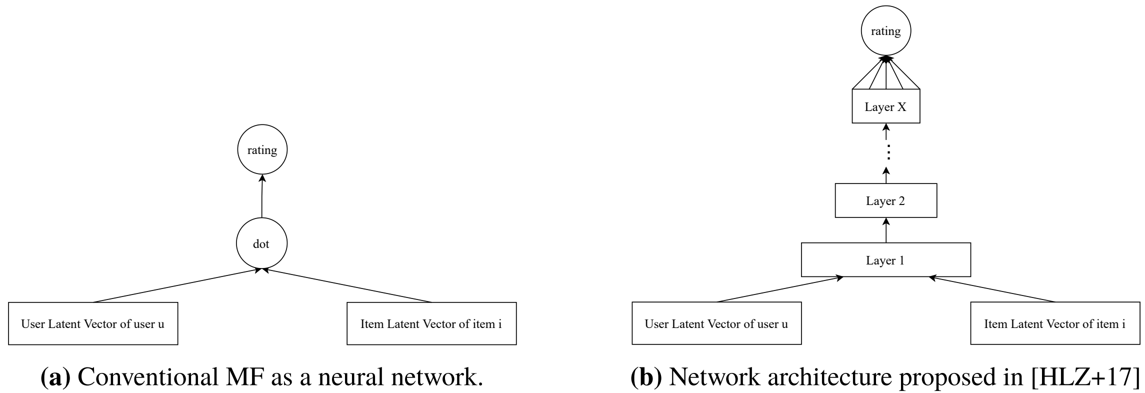
The research of recommender systems is fragmented. It is an umbrella term for a class of systems that uses different methods to recommend items. The method used depends heavily on the context of the recommendation system and the dataset. For instance, MF lends itself to a static context where predictions of ratings are wanted, whereas other approaches recommend items sequentially without predicting a rating. Neural networks are one of the methods that gathered traction recently due to their success on other problems, e.g., in image classification [KSH12]. The main contribution of this work is a recommender using a neural approach; therefore, in Subsection 2.1, we present related research on recommender systems using neural networks. Dynamicity of datasets implies retraining on new data, and retrained neural networks suffer from catastrophic forgetting. Catastrophic forgetting is relevant to our work since we retrain our system on new data and we explore techniques to combat forgetting; therefore, we present research on catastrophic forgetting in Subsection 2.2.

2.1 Neural Approaches to Recommendation

Generally, neural approaches learn a numerical representation of users and items and combine the representations to predict the user feedback or which item to suggest next. The advantage of neural methods to established methods is that they can model non-linear interactions between users and items [HLZ+17]. Furthermore, they can be trained using fast, robust, and readily-available backpropagation methods.

In [KM18], the authors develop a recommender system that considers sequence dynamics to recommend items sequentially. Their key contribution is to employ self-attention to capture short- and long-term patterns in item sequences. Input to their model is a sequence of items chosen by the user. The system creates item and positional embeddings. The positional embeddings are especially important to capture the order of items. The self-attention then weights the importance of elements in the input sequence to find out how strongly an element is correlated to other elements. The authors evaluate their recommender on three datasets: Amazon dataset containing product review, Steam dataset containing reviews of games and user profiles, and the MovieLens-1m dataset containing user ratings of movies. They evaluate the hit rate and entropy of top-10 recommendations [KM18]: hit rate "counts the fraction of times that the ground-truth next item is among the top 10 items, while entropy is a position-aware metric which assigns larger weights on higher positions." Their work improves these two metrics significantly compared to previous sequential neural models. However, all items must be known at training time. The recommender uses item embeddings, which must be pre-trained for all items. As a result, we cannot apply this result to our problem setting.

In [HLZ+17], the authors design a framework for neural networks in collaborative filtering and present a neural network architecture to model latent feature vectors of users and items. The dot-product in conventional MF can only model linear interactions between users and items. They



(a) Conventional MF as a neural network.

(b) Network architecture proposed in [HLZ+17]

Figure 2.1: Xiangnan et al. [HLZ+17] propose to replace the dot product in MF with a neural network to model non-linear interactions of user and item representations.

hypothesize that replacing the dot-product with a neural network can improve the accuracy of recommendations since neural networks can also model non-linear functions. Figure 2.1b shows their network architecture. The network learns item and user embeddings and combines them in a neural network to produce a rating. The authors map MF back to their architecture: MF can be thought of as a particular neural network consisting of one perceptron without an activation function. The authors evaluate their approach on the MovieLens-1M dataset and Pinterest dataset. They transform the ratings in both datasets to be implicit, i.e., 0 or 1, and evaluate the hit rate and entropy of the top-10 recommendations. Their approach shows improved results over MF and ISM. However, Dacrema et al. [DCJ19] have shown that well-tuned ISMs outperform the author's approach significantly on the MovieLens dataset. Like the previous approach, this recommender uses item embeddings. Therefore, all items need to be known during training, making this recommender unsuitable for our problem setting.

Generally, neural approaches use embeddings to represent users and items in a latent space. However, this always implies that all users and items must be known at training time. One requirement for the recommender system is that it can make fast predictions for new items. Fast predictions are impossible if the recommender needs to be retrained for every new item. As such, recommenders employing embeddings to represent items do not fit this requirement. This is what makes the approach of Sedhain et al. [SMSX15] attractive to our work: their neural network does not use item embeddings, and thus, can predict for new items as long as *some* ratings for it exist.

2.2 Catastrophic Forgetting

If not explicitly constrained, neural networks forget old patterns if trained on new samples. The weights will be updated to better fit the new data, which inevitably increases the loss of old data. One method to reduce catastrophic forgetting is changing the training procedure to reduce the loss of old and new training samples. Another approach modifies the network architecture to reduce forgetting.

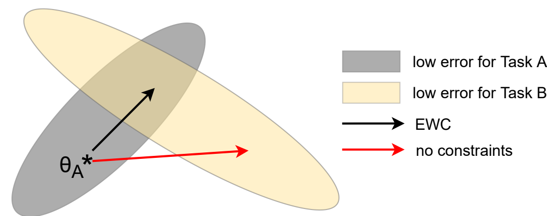


Figure 2.2: Figure adapted from [KPR+16]. Grey and cream indicate parameter regions leading to good performance for Task A and Task B. θ_A denotes the parameters of the learner after training on Task A. By constraining the parameters in θ_A, the learner finds parameters that optimize performance on both datasets.

In [KPR+16] Kirkpatrick et al. introduce a new training procedure called Elastic Weight Consolidation (EWC) that reduces catastrophic forgetting. They compute the importance of every parameter of a neural network with regard to tasks that the network has already been trained on. During the training of new tasks, EWC constrains the important old weights to stay close to their original values. The more important parameter is to previous performance, the less it can change during retraining. They show that their augmented training procedure improves performance on old tasks when the learner is trained on new tasks while not drastically reducing performance on new data. However, Kutalev et al. [KL21] show that EWC does not translate well to more complex network architectures [KL21].

Goodfellow et al. investigate in [GMX+15] the influence of activation functions and Dropout on forgetting. They find that larger neural nets and nets using Dropout are less prone to forgetting, hypothesizing that they are more resistant to forgetting since they take longer to converge. No activation function did show consistent improvements to forgetting; choosing the best function depends on the problem and needs to be cross-validated.

3 Implementation

The Autoencoder approach to recommendation by Sedhain et al. [SMSX15], introduced in Section 1.3, receives vectors of all user ratings for one item as input and reconstructs the vector as an output with all unobserved entries imputed. Effectively, every input has its own neural network. Their key contribution is that unobserved entries in the input are ignored during forward- and backward propagation.

Although the authors provided their code, we implemented our own training routine. Their paper dates back to 2015 when machine learning libraries were not as developed as they are now. State-of-the-art research uses TensorFlow or PyTorch to implement custom neural networks and training procedures. Therefore, we decided to re-implement their approach in today's TensorFlow. We decided to use TensorFlow [Ten21] through the Keras API [Ker21] due to its ease of use and wide usage in research and production environments.

We divide the implementation into two parts. First, we describe how we implement the Autoenocer in Keras. Second, we present our implementation of a partial training routine in Keras that only considers observed entries in forward- and backward propagation during Stochastic Gradient Descent.

3.1 Implementation of our Autoencoder

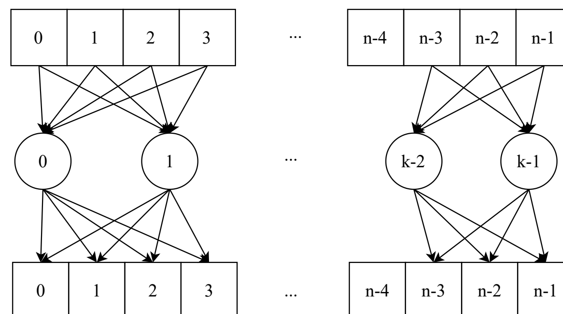


Figure 3.1: Architecture of our Autoencoder.

The implementation of the neural network is straightforward in Keras. We define the input- and output layer as size n, which is the number of users. The hidden layer is a dense layer of size k with a kernel regularizer and an activation function. We leave the exact choice of k, the activation function, regularization parameters, and other hyperparameters for the Evaluation Chapter 4.

Layer	Output Shape
Input	n
Dense	k
Activation	k
Dense/Output	n

Table 3.1: Overview of the implementation of the Autoencoder in Keras.

3.2 Implementation of Partial Training in Keras

We segment the implementation of partial training in Keras into two parts: partial forward propagation and partial backward propagation. In partial forward propagation, we want unobserved entries in the input not to influence the output. We achieve this by constraining unobserved entries in the input to 0. Furthermore, we want the gradients of weights corresponding to unobserved entries to be zero in partial backward propagation. We achieve this by calculating the loss only on observed values and that unobserved entries are constrained to be 0. In the subsequent paragraphs, we explain both segments in further detail.

We introduce notation to present our implementation. Let $x^{(l)}$ denote the output of the l-th layer of a neural network. Then $x^{(0)}$ denotes the first layer, i.e. the input to the Autoencoder. Let $x_j^{(0)}$ refer to one entry in the input, $x_j^{(l)}$ refers to the output of node j in Layer l. $w_{i,j}^{(l)}$ indicates an entry i, j in the weight matrix $\mathbf{W}^{(l)}$, $b_j^{(l)}$ refers to the bias of node j in Layer l. The activation function is denoted by Φ.

Forward Pass

The notion of unobserved values in the input does not exist in neural networks, a neural network's input needs to be well-defined.

$$x_j^{(1)} = \Phi\left(\sum_i w_{i,j}^{(1)} \cdot x_i^{(0)} + b_j^{(0)}\right) \quad (3.1)$$

The variable $x_j^{(1)}$ in Equation 3.1 shows the output at Node j in Layer 1, $x^{(0)}$ denotes the input. However, we want the output to take the following form:

$$x_j^{(1)} = \Phi\left(\sum_{i \in \Omega} w_{i,j}^{(1)} \cdot x_i^{(0)} + b_j^{(0)}\right) \quad (3.2)$$

where Ω denotes the observed entries of $x^{(0)}$. A hidden node should consider only weights corresponding to observed values. Instead of only calculating the sum over observed values, which would be computationally expensive, we employ the following trick: we set all unobserved entries in the input to be 0. We define $x^{(0)\prime}$ to be the same vector as $x^{(0)}$, except for unobserved entries:

$$x_i^{(0)\prime} = \begin{cases} 0, & \text{if } x_i^{(0)} \text{ unobserved} \\ x^{(0)}, & \text{otherwise} \end{cases} \quad (3.3)$$

Then, calculating Equation 3.2 falls back to known vector multiplication of Equation 3.1:

$$(3.4)\quad x_j^{(1)} = \Phi\left(\sum_{i\in\Omega} w_{i,j}^{(1)} \cdot x_i^{(0)} + b_j^{(0)}\right) = \Phi\left(\sum_{i} w_{i,j}^{(1)} \cdot x^{(0)\prime} + b_j^{(0)}\right).$$

Using this trick, we do not need to change the forward-pass procedure in Keras; we set all unobserved entries to 0. Setting to 0 may cause problems in datasets where it occurs as a value but can be dealt with by, e.g., remapping the 0 to another value.

Backward Pass

Backwards propagation works by computing a gradient of the loss function with reference to the weights in the neural network, cf. Equation 3.5:

$$(3.5)\quad \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(1)}} = \delta_j^{(1)} \cdot x_i^{(0)}$$

where $\delta_j^{(1)} = \mathbf{W}^{(2)} \cdot \delta^{(2)} \odot \Phi'(z^{(1)})$. Starting at the ouput node, the gradient is passed backwards through the network. For weights associated to unsobserved weights not to train, we want their gradient to be zero.

$$(3.6)\quad \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(1)}} = 0 \text{ for } i \in \Omega$$

The trick of setting unobserved values to 0 also ensures that the corresponding gradients are 0.

$$(3.7)\quad \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(1)}} = \delta_j^{(1)} \cdot x_i^{(0)\prime} = \delta_j^{(1)} \cdot 0 = 0 \text{ for } i \in \Omega$$

Taking the gradient leaves $x_i^{(0)\prime}$, however, since $x_i^{(0)\prime} = 0$ the resulting gradient is also 0. We achieved that deactivated weights do not train; however, the loss is still computed on all input entries, including the zero entries of unobserved values.

A conventional loss function computes the loss on all input values. In this work, we use the Mean Squared Error (MSE) as a loss function during training, cf. Equation 3.8.

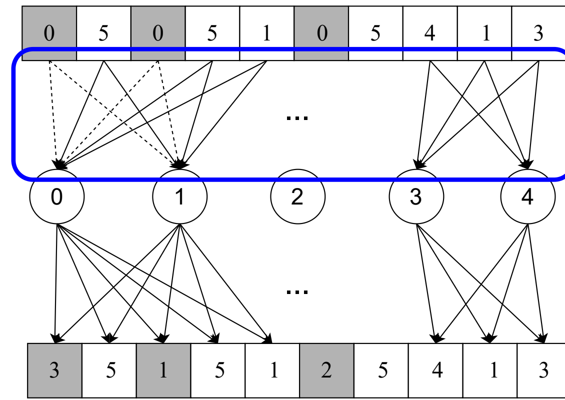
$$(3.8)\quad \mathcal{L} = \frac{1}{2N} \cdot \sum_{i=0}^{n-1} (\hat{x}_i - x_i)^2$$

However, we want to loss function to only consider observed values:

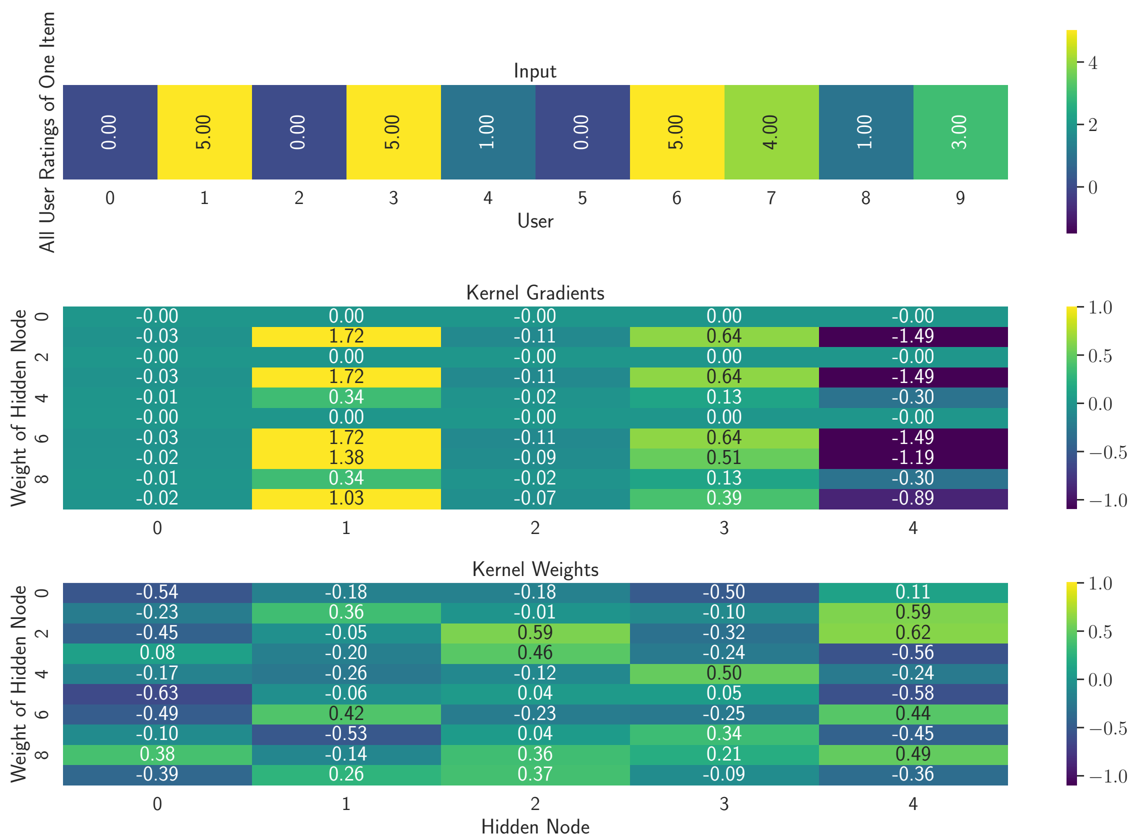
$$(3.9)\quad \mathcal{L} = \frac{1}{2N} \cdot \sum_{i\in\Omega} (\hat{x}_i - x_i)^2$$

We achieve this in Keras by creating a loss function that accepts a mask as an additional input. Instead of calculating the loss on all values, the loss function first uses the passed mask to mask the input and output and then calculates the loss on unmasked values. We create the mask by masking all values which are equal to 0. This way, unobserved entries are masked, and the loss is calculated only over observed entries.

We can see that setting unobserved entries to 0 serves two mechanisms: First, entries with a value of 0 do not contribute in a calculation. Second, 0 entries are masked so that the loss function is only calculated on observed entries. Figure 3.2b shows the results of our implementation in Keras.



(a) Example Autoencoder Architecture



(b) Input, weights and their gradients of our example Autoencoder in Keras

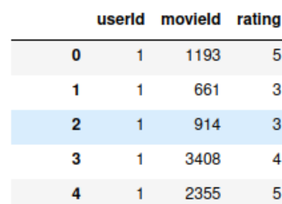
Figure 3.2: We implemented a dummy Autoencoder to illustrate our implementation. Figure 3.2a shows the architecture of the dummy Autoencoder: number of users is $n = 10$, the hidden layer size is $k = 5$. The dotted arrows are deactivated weights. Figure 3.2b shows the circled weights and their respective gradients in Keras. The gradients of deactivated weights (in our example weight row 0, 2, and 5 in the middle plot in 3.2b) are 0, showing that our implementation works.

4 Evaluation

We choose to present our evaluation segmented by dataset, as the evaluation approach and results differ significantly between the two. We evaluate our work on two datasets: the MovieLens-1M dataset [Mov21] and the Microsoft MIND dataset [WQC+20]. For each dataset, we present our evaluation setup and our results.

4.1 MovieLens Dataset

MovieLens is a widely-used set of datasets maintained by the University of Minnesota. The MovieLens-1M dataset [Mov21] is especially popular among researchers in recommender systems research. It contains 1,000,209 ratings of $m = 3{,}952$ movies by $n = 6{,}040$ users on MovieLens from 2000 to 2003, with a sparsity of 95.81%. For one item, 5,787 entries are unobserved and 273 are observed. Users made ratings on a 5-star scale; each user has at least 20 ratings. The dataset is provided in the form shown in Figure 4.1, where each row is one rating given by one user. We transform the dataset into the known item-user form, where a row refers to one movie



	userId	movieId	rating
0	1	1193	5
1	1	661	3
2	1	914	3
3	1	3408	4
4	1	2355	5

Figure 4.1: Screenshot of the format of the MovieLens dataset.

with all observed ratings for it. In contrast to Sedhain et al. [SMSX15] we construct the training, validation, and test sets by splitting the dataset along the item axis. I.e., the test set is movies which the Autoencoder has not seen during training. The authors split the dataset along the individual ratings; thus, the test set contains only unseen ratings, not completely unseen movies.

4.1.1 Evaluation Setup

We run our evaluation on widely-available server hardware, cf. Table 4.1.

Table 4.2 shows the parameters for the Autoencoder architecture and its training procedure. We determined the parameters using a hyperparameter search.

We split the dataset in two different ways to run two different evaluations. Table 4.3a shows the first split to which we refer to as the *Forgetting Split*, Table 4.3b shows the *Tradeoff Split*. The splits differ primarily in the test set size.

CPU	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Memory	8 x 16 GiB DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
OS	Ubuntu 20.04.2 LTS
Kernel	5.4.0-77-generic

Table 4.1: Hardware and OS Parameters.

Layer	Output Shape
Input	$n = 6{,}040$
Dense	$k = 100$
Activation	ReLU
Dense/Output	$n = 6{,}040$

(a) Autoencoder Architecture Parameters.

Batch Size	64
Training Epochs	3,500
Kernel Regularizer	1×10^{-3}
Optimizer	Default Keras Adam
Learning Rate	1×10^{-2}

(b) Autoencoder Training Parameters.

Table 4.2: Autoencoder Evaluation Parameters for the MovieLens-1M dataset.

The *Forgetting Split* simulates the settings where 100 new items arrive every day, defined in Section 1.4. The test size is 50% of the dataset size; thus, we can simulate 19 days of items arriving every day. We use it to quantify the decrease of the RMSE (forgetting) when retraining on new samples.

We use the *Trade-off Split* to determine the accuracy of our Autoencoder and to explore the trade-offs of three techniques aimed at fighting catastrophic forgetting: reducing epochs spent on retraining, reducing the learning rate, and replaying old training samples. The number of epochs spent on retraining influences the accuracy of the network on new training samples: more epochs during retraining lead to lower RMSE on the new data; however, also to overtraining on these new samples. Here, a compromise needs to be chosen between a low RMSE on new data and forgetting the old data. The intuition behind reducing the learning is that a reduced learning rate also reduces the decrease of the RMSE on old samples. However, reducing the learning rate also reduces learning on the new samples. Instead of augmenting training parameters, we can replay old samples when retraining on new ones. By replaying old samples, the network should forget old patterns to a lesser extent.

Dataset	Size
Training Set	45%
Validation Set	5%
Test Set	50%

(a) Forgetting Train/Test Split.

Dataset	Size
Training Set	90%
Validation Set	10%
Test Set	100 samples

(b) Tradeoff Train/Test Split.

Table 4.3: The two different Train/Test splits of the MovieLens-1M dataset we evaluate in this work.

All three techniques introduce a trade-off between RMSEs on old- and new samples. Thus, we analyze the RMSE on the training, validation, and test set. The training set consists of 100 new items, which simulates one day of new items arriving. Therefore, we limit the forgetting in the trade-off analyses to one day. To evaluate the epoch trade-off, we retrain the Autoencoder on the

100 new samples for an increasing number of epochs. We evaluate the learning rate trade-off by retraining the Autoencoder using different learning rates. We vary the fraction of training data we replay during retraining for the replay trade-off. We record the RMSE on the training, validation, and test sets and average over the last ten epochs of retraining for each technique. Table 4.4 shows the range of values we analyze for each technique.

Technique	Values Analyzed
Epoch Trade-off	1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1,000
Learning Rate Trade-off	1×10^{-6}, 5×10^{-6}, 1×10^{-5}, 5×10^{-5}, 1×10^{-4}, 5×10^{-4}, 1×10^{-3}, 5×10^{-3}, 1×10^{-2}
Replay Trade-off	0, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1

Table 4.4: Each technique and the values we analyze in this work for the MovieLens-1M dataset. The values for the Replay Trade-off refer to the fraction of the training data replayed during retraining on new samples. When not varied, epochs, learning rate, and replay fraction were kept at 5, 1×10^{-2}, and 0 respectively.

4.1.2 Evaluation Results

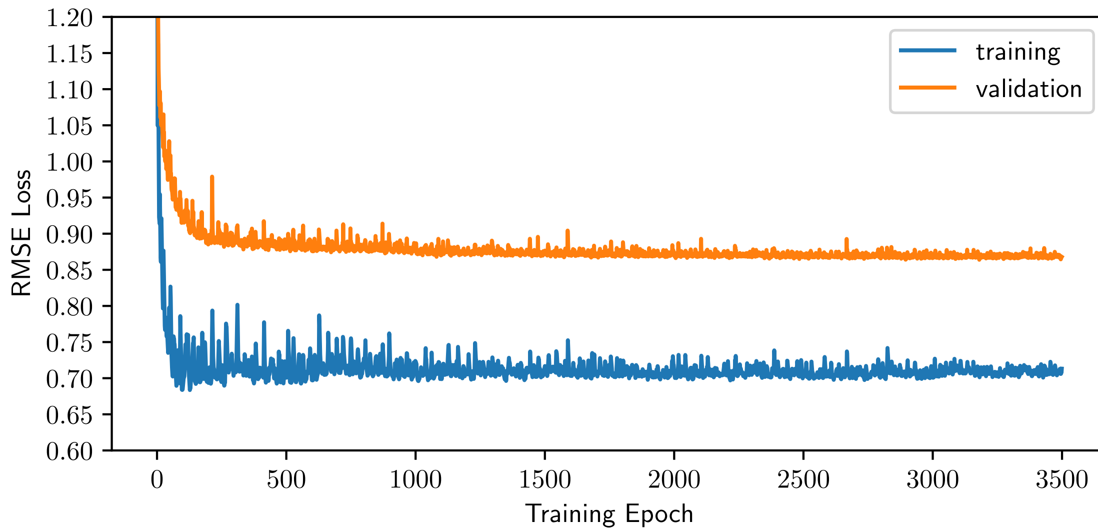


Figure 4.2: Training and validation loss of the Autoencoder on the MovieLens-1M dataset using the Trade-off split. Final validation RMSE is around 0.86 which is state-of-the-art performance.

The Autoencoder trained in 16 minutes and achieved a state-of-the-art RMSE of 0.86 on movies that were not seen during training, cf. Figure 4.2. The Autoencoder trains regularly without overfitting.

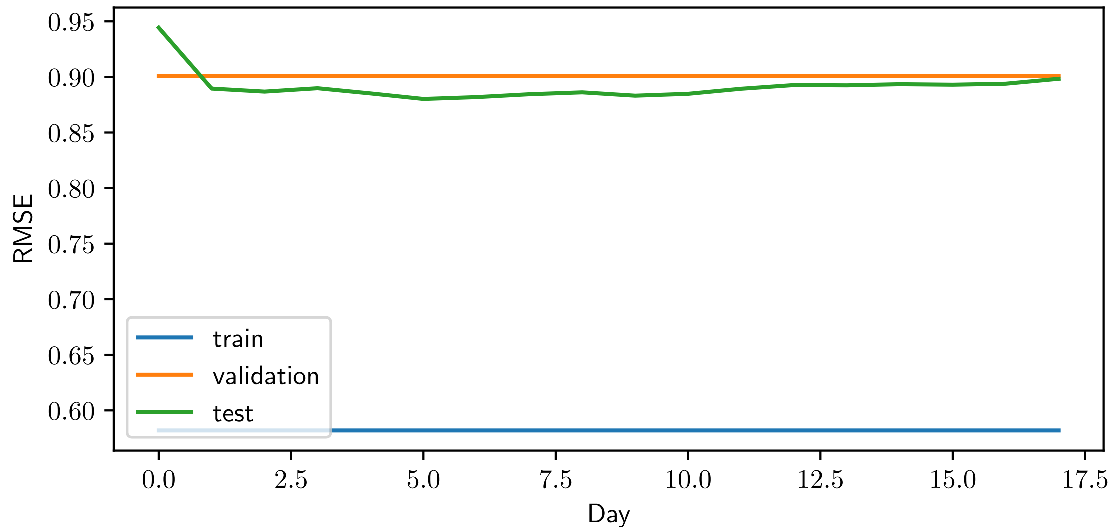


Figure 4.3: RMSE on the test set with 100 movies arriving every day, which is the problem setting of this work. The movies in the dataset are not ordered chronologically; thus, no preference change is visible, indicated by an increasing RMSE on the test set.

Each rating given by a user has a timestamp. We could have used the timestamps to order the movies chronologically by their appearance in the dataset. However, we decided against this for two reasons: First, MovieLens advised against using the timestamps since they do not correspond to any real action such as watching the movie. Second, most movies existed for a long time when users gave ratings; thus, the movies cannot be thought of as new. Therefore, the test set of the Forgetting Split consists of 3,000 randomly sampled movies. Figure 4.3 shows the accuracy on 100 new items arriving each day. The accuracy on the test set hovers around the validation set as the days increase. So, in reality, we would not need to retrain the Autoencoder. However, in the remaining paragraphs, we still retrain on new items as an academic exercise to analyze the catastrophic forgetting techniques.

Increasing the number of epochs spent on retraining on new data decreases RMSE on new samples but increases RMSE monotonically for old samples, cf. Figure 4.4. Additionally, the Autoencoder overfits to the new samples very quickly, which is indicated by the RMSE of 0.1 starting at around 100 epochs. This is not surprising since we retrain the Autoencoder on 100 new samples only. In contrast, the training set comprises around 5,400 samples. The RMSE on the training and validation set converge to 1.0. The convergence happens very quickly, after 50 epochs both RMSEs already reached 1.0.

Increasing the number of new samples to retrain on, i.e., the test set size increases RMSE on the test set and decreases it on the training and validation set, indicating less overfitting, cf. Figure 4.5. So the overfitting observed previously is indeed rooted in the small number of new samples (100).

Increasing the learning-rate increases the RMSE on the training and validations sets while showing changing behavior on the test set. The optimal learning rate on the test set is at 1×10^{-3}, at other learning rates, RMSE increases. However, training and validation RMSE already increased

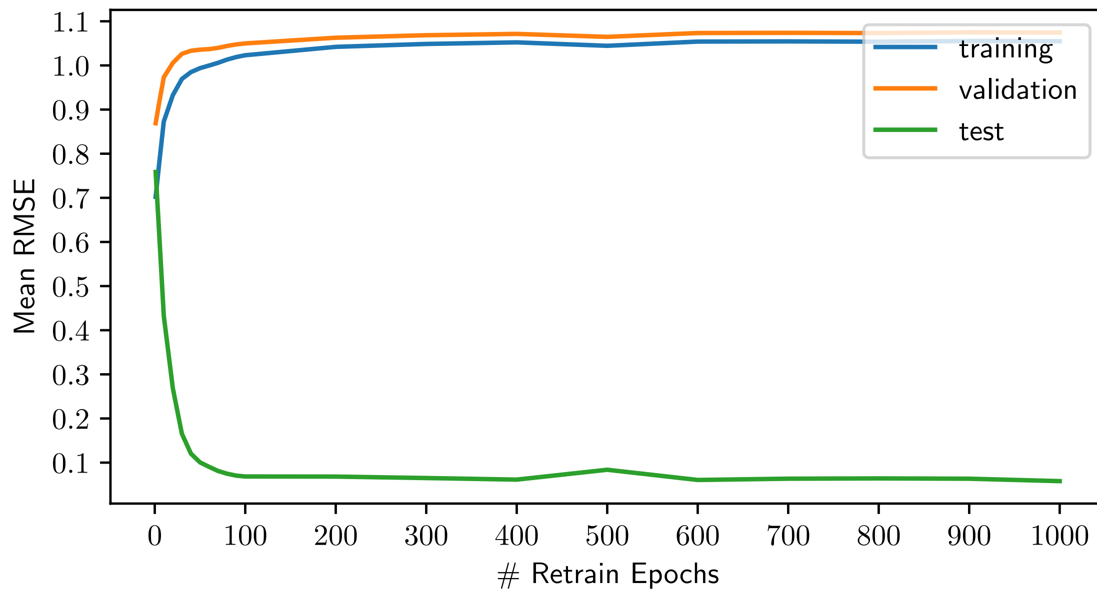


Figure 4.4: The trade-off for number of epochs spent on retraining on the new 100 samples versus the RMSE on training, test, and validation set. The RMSE on the training and validation set increase very quickly, while decreasing very quickly on the test set. This indicates massive overfitting on the new 100 test samples.

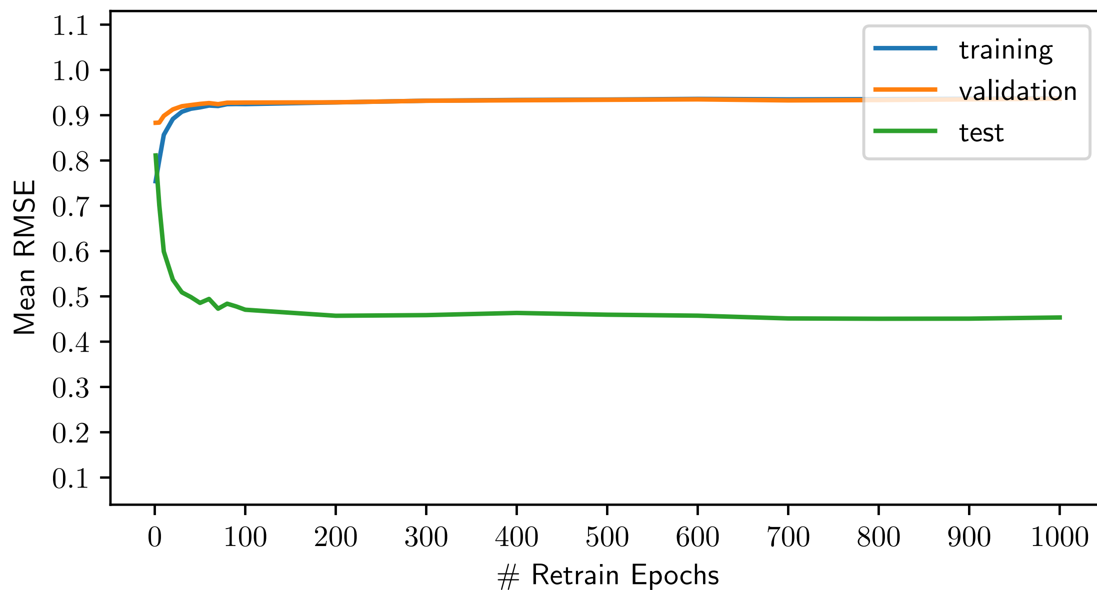


Figure 4.5: Retraining on 1,000 instead of 100 new samples reduces the overfitting seen in Figure 4.4.

significantly at that 1×10^{-3}, suggesting that increasing learning rate increases overfitting. At 5×10^{-3} the test set RMSE increases again, suggesting that the learning rate is too high and the Autoencoder is not learning anymore.

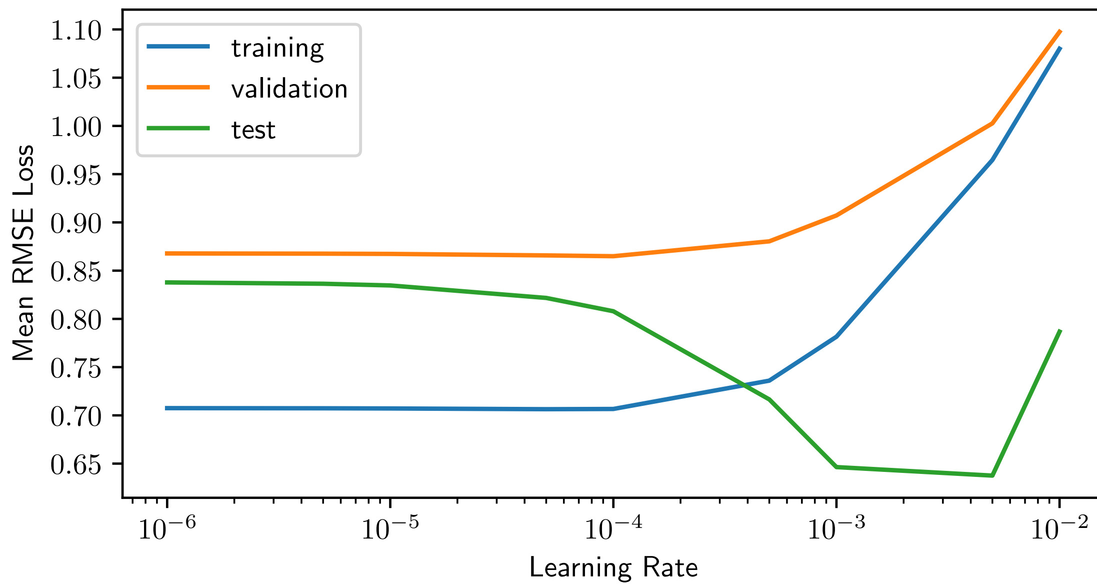


Figure 4.6: Learning-rate versus the RMSE on training, test, and validation set during retraining on the new 100 samples. The RMSE on training and validation set increase gradually, while first decreasing and then increasing on the test set. Note the logarithmic scale.

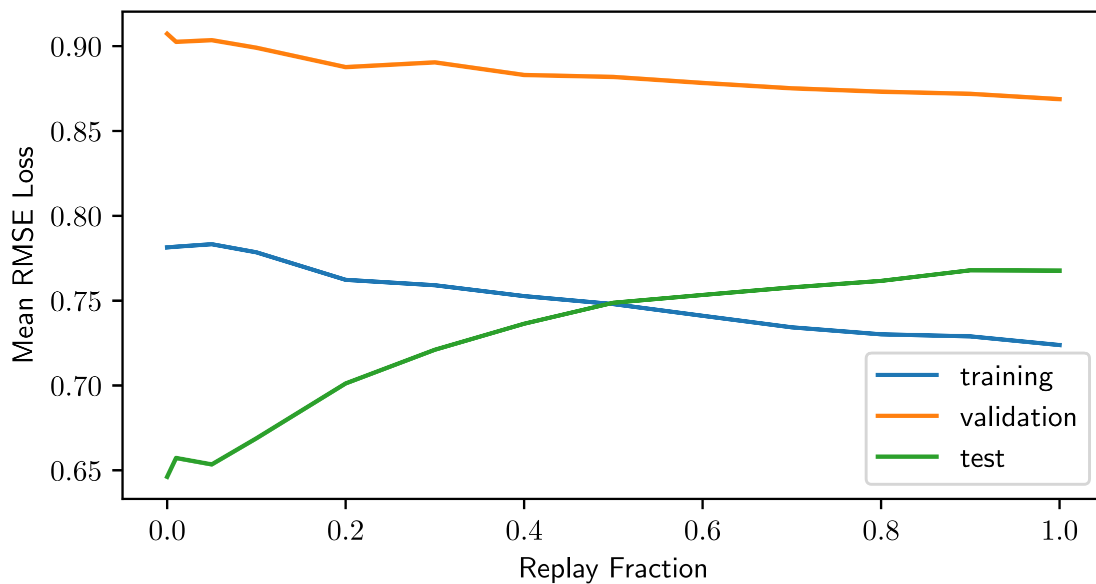


Figure 4.7: In the replay trade-off, the RMSEs on the training and validation set decrease monotonically, while increasing monotonically on the test set, which suggests less overfitting and more generalizability.

The replay fraction has a positive effect on all RMSEs, reducing the overfitting of the Autoencoder. The RMSEs on the training and validation set decrease monotonically. At a replay fraction of 1.0, the validation RMSE dropped back to 0.86. In contrast, the RMSE increases monotonically on the

test set. The increased RMSE on the test set suggests less overfitting and more generalizability. It also converges to about 0.77, indicating that the five epochs spent on retraining are enough to incorporate the new samples into the Autoencoder. A replay fraction of 1.0 seems to be best for the Autoencoder performance. Retraining does take 5 seconds since we retrain for five epochs; however, choosing this trade-off seems reasonable considering its impact on Autoencoder performance.

The Autoencoder exhibited an RMSE of 0.86 on new movies, which is a state-of-the-art performance for the MovieLens-1M dataset. The accuracy did not decrease for the 100 new movies arriving every day. However, this is not surprising since it was impossible to introduce the movies in chronological order by their appearance in the dataset. Out of the catastrophic forgetting techniques, a replay fraction of 1.0 is the most effective in combatting forgetting and reducing overfitting on new items. However, there is a slight increase in training time due to replaying the whole dataset during the five training epochs. Increasing the epochs for retraining makes the Autoencoder forget fast, thus, it is best to keep the number of epochs low at around 5. Increasing or decreasing the learning rate does not seem to combat catastrophic forgetting. At learning rates of 1×10^{-4} or lower, the Autoencoder does not learn at all. The Autoenocder is not forgetting, however, neither is it incorporating the new data. A too high learning rates makes it underfit severely on all datasets. Neither effect reduces forgetting, so it is best to keep the learning rate at its optimal value to allow learning during the replay technique.

4.2 MIND Dataset

The MIND dataset is a large dataset published by Microsoft to facilitate news recommendation research. It contains anonymized behavior logs of users on a Microsoft News website. The large versions contains 160,000 english news articles in more than 15,000,000 impression logs generated by 1,000,000 users. The smaller version contains 51,282 articles in 156,965 impression logs generated by 50,000 users. This corresponds to a sparsity of 99.77%; in contrast, the MovieLens dataset had a lower sparsity of 95.81%. We use the smaller version of the dataset in this work, as the number of users is already one magnitude larger than in the MovieLens-1M dataset. An impression is a selection of articles that Microsoft presented to a user on their news website. Ideally, the impression should be catered to the user, i.e., created with the user's preferences in mind. Users give feedback implicitly by clicking on articles. A clicked article is a 1; non-clicked articles get a 0.

	impression_id	user_id	time	history	impressions
0	1	U13740	2019-11-11 09:05:58	N55189 N42782 N34694 N45794 N18445 N63302 N104...	N55689-1 N35729-0
1	2	U91836	2019-11-12 18:11:30	N31739 N6072 N63045 N23979 N35656 N43353 N8129...	N20678-0 N39317-0 N58114-0 N20495-0 N42977-0 N...
2	3	U73700	2019-11-14 07:01:48	N10732 N25792 N7563 N21087 N41087 N5445 N60384...	N50014-0 N23877-0 N35389-0 N49712-0 N16844-0 N...
3	4	U34670	2019-11-11 05:28:05	N45729 N2203 N871 N53880 N41375 N43142 N33013 ...	N35729-0 N33632-0 N49685-1 N27581-0
4	5	U8125	2019-11-12 16:11:21	N10078 N56514 N14904 N33740	N39985-0 N36050-0 N16096-0 N8400-1 N22407-0 N6...

Figure 4.8: Screenshot of the format of the dataset. Each user has a history of read articles. Each impression has a timestamp, a user who saw the impression, and the articles shown in the impression.

To feed the dataset as an input to the Autoencoder, we have to distinguish between non-clicked movies shown in impressions and non-clicked movies never shown to a user. We have to make this distinction for two reasons. First, the Autoencoder could always predict 1s for unobserved entries to minimize loss. Thus, we must introduce at least one more value to force the Autoencoder to decide between two values. Second, it makes sense to separate the non-clicked events because their user feedback has a different meaning. A user made a conscious decision to not click on a movie shown in an impression, while he could not have made that decision on a movie not shown to him. Thus, we assign non-clicked articles not shown in impressions 0's and non-clicked articles shown in impressions -1's.

In contrast to the MovieLens dataset, we chronologically order the articles by their first appearance in the dataset. Microsoft published news articles more frequently than new movies were added to MovieLens; thus, the chronological ordering makes sense. We transform the dataset into item-user form and create the training, validation, and test splits chronologically along the item dimension.

4.2.1 Evaluation Setup

Table 4.5 shows the parameters for the Autoencoder architecture and its training procedure. We determined the parameters using a hyperparameter search.

Layer	Output Shape
Input	$n = 50{,}000$
Dense	$k = 1{,}000$
Activation	ReLU
Dense/Output	$n = 50{,}000$

(a) Autoencoder Architecture Parameters.

Batch Size	256
Training Epochs	500
Kernel Regularizer	1×10^{-3}
Optimizer	Default Keras Adam
Learning Rate	1×10^{-2}

(b) Autoencoder Training Parameters.

Table 4.5: Autoencoder Evaluation Parameters for the MIND dataset.

Like for the MovieLens dataset, we split the MIND dataset into the *Forgetting Split* and *Tradeoff Split*, cf. Table 4.6.

Dataset	Size
Training Set	45%
Validation Set	5%
Test Set	50%

(a) Forgetting Train/Test Split.

Dataset	Size
Training Set	90%
Validation Set	10%
Test Set	100 samples

(b) Tradeoff Train/Test Split.

Table 4.6: The two different Train/Test splits of the MIND dataset we evaluate in this work.

We again analyze the three catastrophic forgetting techniques introduced in Section 4.1.1, Table 4.7 shows the values we analyze.

Technique	Values Analyzed
Epoch Trade-off	1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Learning Rate Trade-off	1×10^{-6}, 5×10^{-6}, 1×10^{-5}, 5×10^{-5}, 1×10^{-4}, 5×10^{-4}, 1×10^{-3}, 5×10^{-3}, 1×10^{-2}
Replay Trade-off	0, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1

Table 4.7: Each technique and the values we analyze in this work for the MIND dataset. The value ranges did not change compared to the MovieLens dataset, except for the Epoch Trade-off, where we go up to 100 epochs maximum.

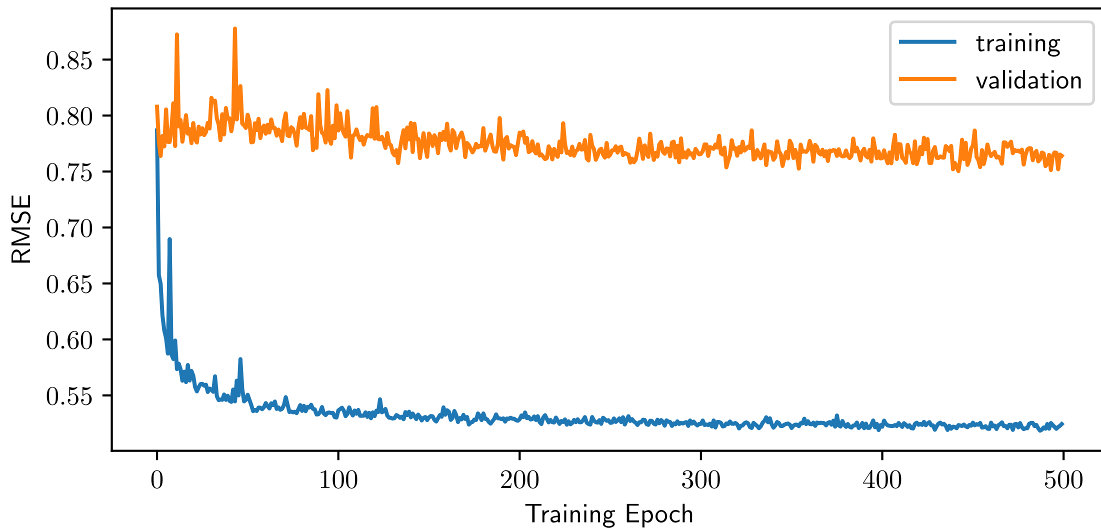


Figure 4.9: Training and validation loss of the Autoencoder on the MIND dataset using the Trade-off split. Final validation RMSE is around 0.77.

4.2.2 Evaluation Results

The Autoencoder trained in 20 hours on the MIND dataset and achieved a validation RMSE of 0.77. The increased training set RMSE compared to MovieLens is due to the higher sparsity of the MIND dataset: Out of 50,000 users, only 115 have given feedback on one item on average. On the MovieLens dataset, 253 out of 6,040 have given feedback. The higher sparsity makes learning a hidden representation of user preferences harder since a high proportion of edges in the neural network graph is deactivated, and thus, their gradients are 0. The RMSE difference on the MIND dataset is 0.27, compared to 0.16 on MovieLens, even though the rating range (from -1 to 1) is smaller on MIND.

Changed user preferences cause the high RMSE difference between training and validation set on the MIND dataset. Since the validation set is split chronologically from the training set, it contains 10% of news articles that appeared last in the dataset. If the hidden representation learned on the training set does not translate to the validation set, the RMSE increases. The sparsities are equal in both datasets, meaning that the difference cannot be caused by the validation set containing less user

feedback. Furthermore, only five users appearing in the validation set did not give any feedback in the training set. The difference cannot be caused by different user bases giving feedback. Thus, we conclude that a user preference shift causes this difference.

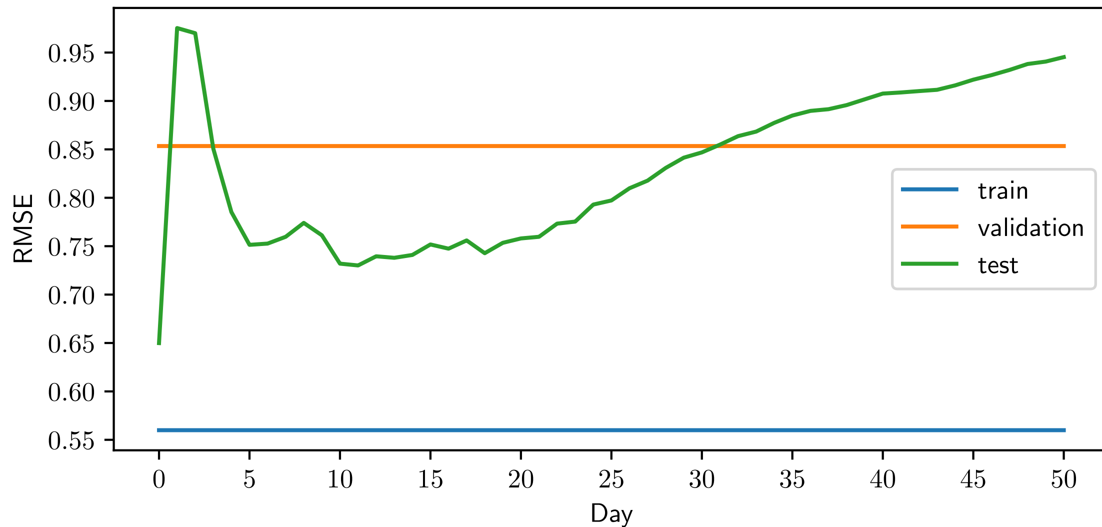


Figure 4.10: RMSE on the MIND Forgetting test set with 100 movies arriving every day. The RMSE increases on the test set as the days go by, suggesting that there is a preference shift in the dataset. In this case, it would make sense to retrain the Autoencoder to capture this shift.

Figure 4.10 reinforces our intuition that the proportionally high validation RMSE is caused by a user preference shift. Apart from two outliers, the test RMSE is below the validation RMSE at a value of 0.75 on Day 20. The test RMSE then increases up to value of 0.95 by Day 50. The Autoencoder's learned hidden representation cannot capture the shifted user preferences, and thus, the RMSE increases gradually. In this case, retraining the Autoencoder would capture this shift and reduce the error.

The epoch trade-off shows different behavior compared to the MovieLens dataset, caused by the previously explained preferences shift. The training set RMSE increases quickly and converges after 20 retrain epochs. In contrast to the MovieLens dataset, the validation RMSE now decreases with the test set RMSE and converges to a value of 0.55. We attribute this behavior to the change in user preferences between the validation and training set. The preferences in the validation and test set are closer than in training and validation set. As a result, retraining on the test set also reduces the error on the validation set. However, the Autoencoder still forgets patterns learned on the training data, as indicated by the high training RMSE.

Increasing the learning rate increases forgetting on the training and validation sets while improving the learning new patterns on the test set, cf. Figure 4.12. Again, the learning rate of 1×10^{-3} seems to be optimal to incorporate new patterns from the test set. However, at this learning rate, the RMSE on the training and validation set already increased significantly, indicating forgetting. Higher learning rates lead to underfitting on all datasets; the Autoencoder does not learn at lower learning rates.

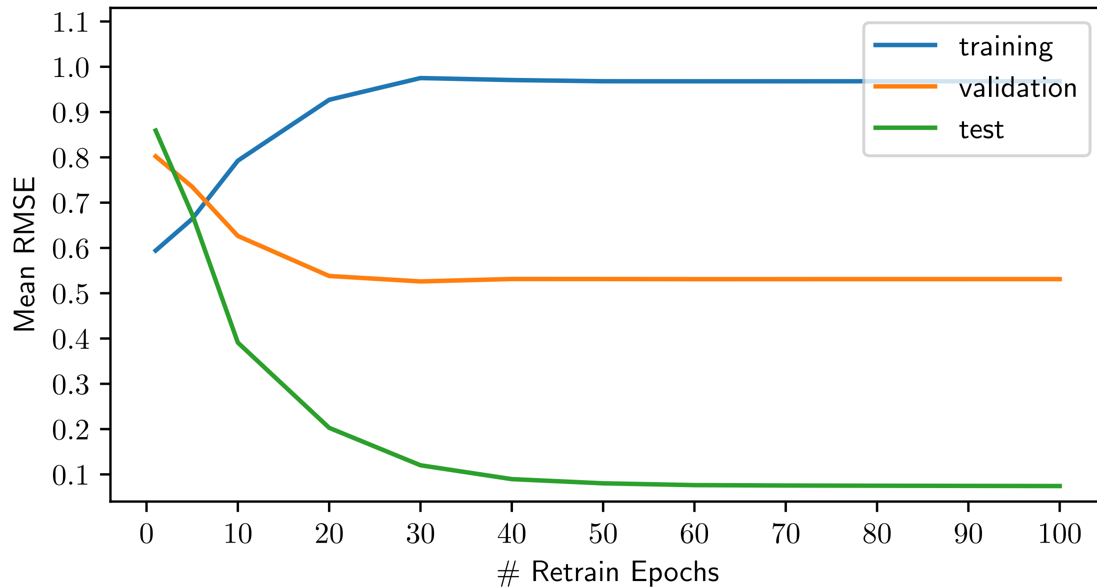


Figure 4.11: The trade-off for number of epochs spent on retraining for the MIND dataset. The RMSE on the training set increases very quickly while decreasing very quickly on the validation and test set. Patterns learned on the test set translate to the validation set and reduce its RMSE. However, overfitting still occurs, indicated by the increasing training RMSE.

Increasing the fraction of old samples reduces catastrophic forgetting but hinders learning new preferences for the MIND dataset. The training RMSE decreases until it converges to 0.55, the value that was achieved during training. The validation set RMSE increases with the test set RMSE, which we explain with the preference shift. The increase indicates that the Autoencoder has trouble learning the new preferences. This is not surprising since the size of the test set is 0.2%. The Autoencoder needs more new samples to incorporate the new patterns.

The Autoencoder achieves an RMSE of 0.77 on the validation set. It has more difficulties generalizing because the preferences between the training and validation set shift, further demonstrated by the increasing RMSE on the Forgetting Split. Like for the MovieLens dataset, the replay techniques turn out to be the most effective at hindering catastrophic forgetting, but at a penalty of retraining time. ´

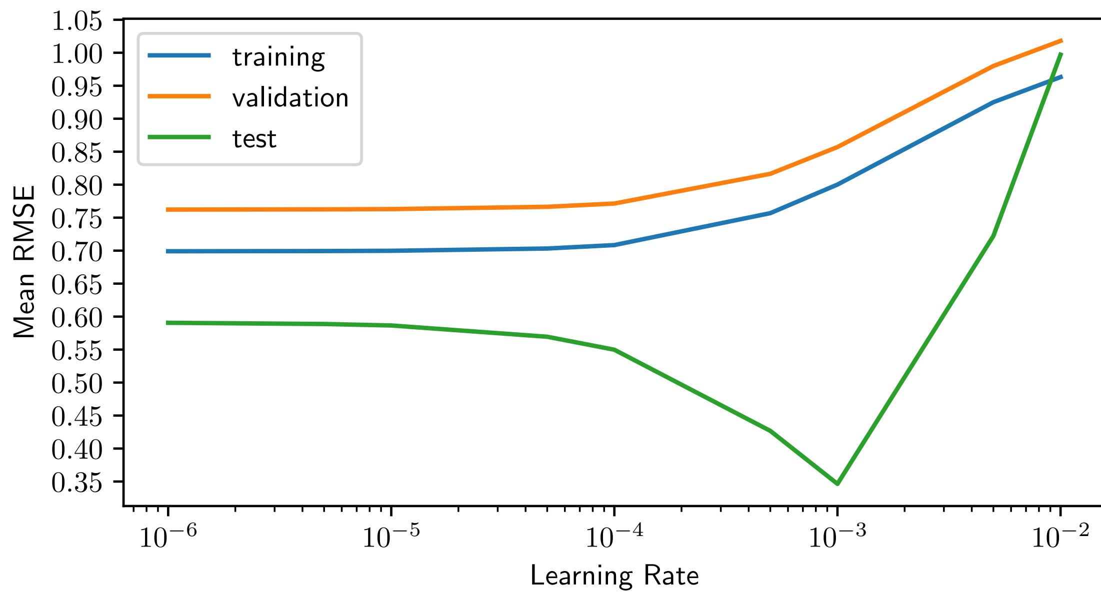


Figure 4.12: The learning-rate trade-off on the MIND dataset. A learning rate of 1×10^{-3} seems to be optimal to incorporate new patterns from the test set. Higher learning rates lead to underfitting on all datasets; at lower learning rates, the Autoencoder does not learn. Note the logarithmic scale.

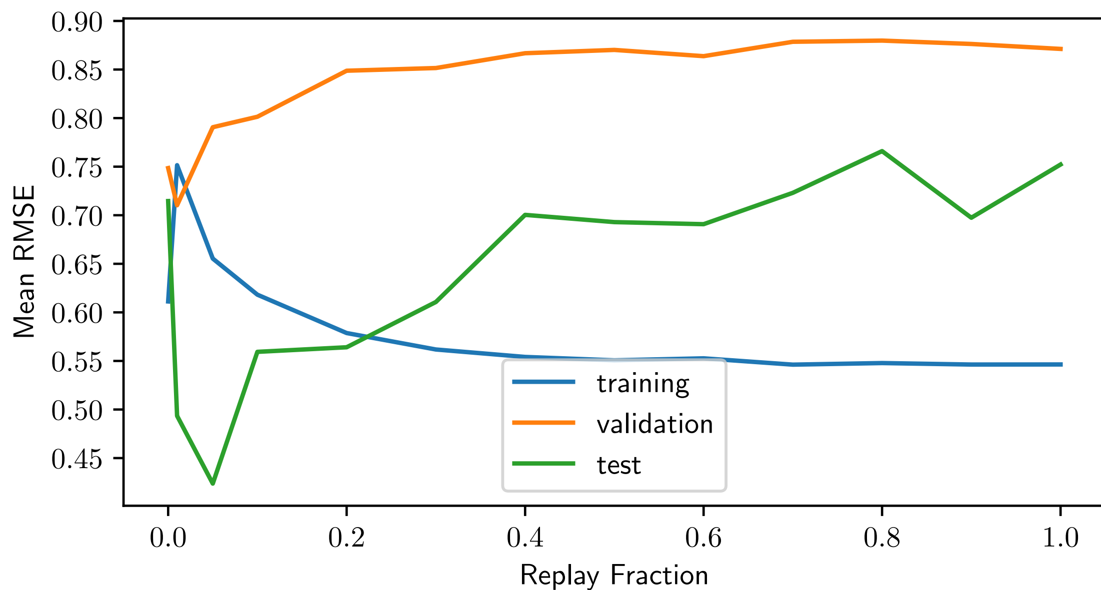


Figure 4.13: In the replay trade-off, the RMSEs on the test and validation set increase, while decreasing monotonically on the training set, which suggests less overfitting and more generalizability.

5 Conclusion and Outlook

Recommender systems recommend *items* to *users* based on their preferences. Well-built recommender systems are essential for a business's success. Today's recommenders need to work on dynamic datasets, i.e., the users, items, and their ratings change frequently. Most techniques, like Matrix Factorization (MF) [KBV09], need to be continually retrained to give recommendations on items that were not included in the training set.

Our main contribution is a dynamic recommender system that periodically recommends a subset of unseen items to the complete user base without retraining. Our second contribution is the evaluation of techniques to decrease catastrophic forgetting in neural networks. Our work builds on the work done by Sedhain et al. [SMSX15]. We implement a neural Autoencoder that takes a vector of all user ratings for one item and reconstructs it. The key here is that the Autoencoder imputes unobserved entries in the input (some users have not given feedback for an item) by using a learned latent representation of the users' feedback behavior. Effectively, the Autoencoder is predicting missing entries.

We implement the Autoencoder and a custom training procedure in TensorFlow. The Autoencoder, a neural network, requires the input to have no unobserved entries. However, our implementation only considers observed entries by ignoring weights of the neural network associated with unobserved entries during forward- and backward propagation. Every input has its own set of activated weights and thus its own neural network. We implement this by setting unobserved values to 0. This way, deactivated weights are multiplied with 0 during forward propagation and do not influence the result. This trick also leads the corresponding gradients to be 0. Lastly, we implemented a custom loss function that only calculates observed entries' loss.

Our Autencoder achieves an RMSE of 0.86 on the MovieLens-1M dataset [Mov21]. It achieves this RMSE on movies that were not included in the training. We then retrained the Autoencoder on new movies to analyze three techniques to combat catastrophic forgetting: reducing epochs spent on retraining, reducing the learning rate, and replaying old training samples. Out of all techniques, replaying old training samples emerged as the most effective technique to stop catastrophic forgetting of patterns learned on old data while at the same time inhibiting overfitting to the new samples.

Our approach attains an RMSE of 0.77 on the Microsoft MIND dataset [WQC+20]. The test and validation sets contain articles that appeared later in the dataset, whereas the training set contains early appearing articles. When simulating 100 new articles arriving each day with the test set, the Autoencoder can predict user behavior with an RMSE of 0.75 on Day 0. However, the RMSE increases gradually until 0.95, indicating a user preference shift in the dataset. In this case, retraining would be advantageous to incorporate this shift into the Autoencoder's hidden representation. Again, replaying proved to be the most effective technique against catastrophic forgetting.

Outlook

While showing promising results on the two evaluated datasets, evaluation on other datasets should be part of future work. Further evaluation on other datasets is necessary to ensure that our approach is feasible in many applications.

Further work could go into improving the scalability of our approach. The training time and memory requirements scale with the number of users. While the scalability did not pose problems in our work, it could become a problem in datasets with a substantial user base in the millions. One approach could be partitioning the users into batches and training a network separately for each batch. Handling user batches separately would reduce memory requirements but not training time. Another approach could be to cluster similar users and give one representative user of each cluster as input to the Autoencoder.

Currently, our approach only works on a stable user base; future work could expand our approach to work in settings where new users arrive. The network architecture depends on the number of users. If a new user joins, the network would need to be expanded by one node and then retrained to include them. Our approach can also work on user vectors instead of item vectors by simply passing a vector of one user and all his feedback as input. We imagine that combining two Autoencoders, one to recommend new items and one to recommend old items to new users, could cover datasets where new movies and new users arrive.

Bibliography

[AÁK12] C. Abdelberi, G. Ács, M. A. Kâafar. "You are what you like! Information leakage through users' Interests". In: *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. URL: https://www.ndss-symposium.org/ndss2012/you-are-what-you-information-leakage-through-users-interests (cit. on p. 8).

[Agg16] C. C. Aggarwal. *Recommender Systems - The Textbook*. Springer, 2016. ISBN: 978-3-319-29657-9. DOI: 10.1007/978-3-319-29659-3. URL: https://doi.org/10.1007/978-3-319-29659-3 (cit. on p. 10).

[DCJ19] M. F. Dacrema, P. Cremonesi, D. Jannach. "Are we really making much progress? A worrying analysis of recent neural recommendation approaches". In: *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019*. Ed. by T. Bogers, A. Said, P. Brusilovsky, D. Tikk. ACM, 2019, pp. 101–109. DOI: 10.1145/3298689.3347058. URL: https://doi.org/10.1145/3298689.3347058 (cit. on p. 14).

[ETK+18] M. D. Ekstrand, M. Tian, M. R. I. Kazi, H. Mehrpouyan, D. Kluver. "Exploring Author Gender in Book Rating and Recommendation". In: *CoRR* abs/1808.07586 (2018). arXiv: 1808.07586. URL: http://arxiv.org/abs/1808.07586 (cit. on p. 8).

[GMX+15] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, Y. Bengio. *An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks*. 2015. arXiv: 1312.6211 [stat.ML] (cit. on p. 15).

[Gri17] D. R. Grimes. *Echo chambers are dangerous – we must try to break free of our online bubbles*. 2017. URL: https://www.theguardian.com/science/blog/2017/dec/04/echo-chambers-are-dangerous-we-must-try-to-break-free-of-our-online-bubbles (visited on 04/12/2017) (cit. on p. 8).

[HLZ+17] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, T.-S. Chua. "Neural Collaborative Filtering". In: *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. Ed. by R. Barrett, R. Cummings, E. Agichtein, E. Gabrilovich. ACM, 2017, pp. 173–182. DOI: 10.1145/3038912.3052569. URL: https://doi.org/10.1145/3038912.3052569 (cit. on pp. 13, 14).

[KBV09] Y. Koren, R. M. Bell, C. Volinsky. "Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42.8 (2009), pp. 30–37. DOI: 10.1109/MC.2009.263. URL: https://doi.org/10.1109/MC.2009.263 (cit. on pp. 9, 10, 33).

[Ker21] Keras. *Keras. Simple. Flexible. Beautiful.* 2021. URL: https://keras.io/ (cit. on p. 16).

[KL21] A. Kutalev, A. Lapina. “Stabilizing Elastic Weight Consolidation method in practical ML tasks and using weight importances for neural network pruning”. In: *CoRR* abs/2109.10021 (2021). arXiv: 2109.10021. URL: https://arxiv.org/abs/2109.10021 (cit. on p. 15).

[KM18] W.-C. Kang, J. J. McAuley. “Self-Attentive Sequential Recommendation”. In: *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*. IEEE Computer Society, 2018, pp. 197–206. DOI: 10.1109/ICDM.2018.00035. URL: https://doi.org/10.1109/ICDM.2018.00035 (cit. on p. 13).

[KPR+16] J. Kirkpatrick, R. Pascanu, N. C. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, R. Hadsell. “Overcoming catastrophic forgetting in neural networks”. In: *CoRR* abs/1612.00796 (2016). arXiv: 1612.00796. URL: http://arxiv.org/abs/1612.00796 (cit. on p. 15).

[KSH12] A. Krizhevsky, I. Sutskever, G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, K. Q. Weinberger. 2012, pp. 1106–1114. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (cit. on p. 13).

[LSY03] G. Linden, B. Smith, J. York. “Amazon.com recommendations: item-to-item collaborative filtering”. In: *IEEE Internet Computing* 7.1 (2003), pp. 76–80. DOI: 10.1109/MIC.2003.1167344 (cit. on p. 9).

[MAP+20] M. Mansoury, H. Abdollahpouri, M. Pechenizkiy, B. Mobasher, R. Burke. “Feedback Loop and Bias Amplification in Recommender Systems”. In: *CoRR* abs/2007.13019 (2020). arXiv: 2007.13019. URL: https://arxiv.org/abs/2007.13019 (cit. on p. 8).

[Mov21] MovieLens. *MovieLens 1M Dataset*. 2021. URL: https://grouplens.org/datasets/movielens/1m/ (visited on 10/22/2021) (cit. on pp. 2, 12, 21, 33).

[RKM+01] N. Ramakrishnan, B. J. Keller, B. J. Mirza, A. Grama, G. Karypis. “When being Weak is Brave: Privacy in Recommender Systems”. In: *CoRR* cs.CG/0105028 (2001). URL: https://arxiv.org/abs/cs/0105028 (cit. on p. 8).

[SMSX15] S. Sedhain, A. K. Menon, S. Sanner, L. Xie. “AutoRec: Autoencoders Meet Collaborative Filtering”. In: *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Florence, Italy, May 18-22, 2015 - Companion Volume*. Ed. by A. Gangemi, S. Leonardi, A. Panconesi. ACM, 2015, pp. 111–112. DOI: 10.1145/2740908.2742726. URL: https://doi.org/10.1145/2740908.2742726 (cit. on pp. 2, 10, 11, 14, 16, 21, 33).

[Sto21a] J. Stoll. *Number of Netflix paid subscribers worldwide from 3rd quarter 2011 to 2nd quarter 2020*. 2021. URL: https://www.statista.com/statistics/250934/quarterly-number-of-netflix-streaming-subscribers-worldwide/ (visited on 10/22/2021) (cit. on p. 9).

[Sto21b] J. Stoll. *Number of original content titles released by Netflix in the United States from 2012 to 2019*. 2021. URL: https://www.statista.com/statistics/883491/netflix-original-content-titles/ (visited on 10/08/2021) (cit. on p. 9).

[Ten21] TensorFlow. *An end-to-end open source machine learning platform*. 2021. URL: https://www.tensorflow.org/ (cit. on p. 16).

[WQC+20] F. Wu, Y. Qiao, J.-H. Chen, C. Wu, T. Qi, J. Lian, D. Liu, X. Xie, J. Gao, W. Wu, M. Zhou. "MIND: A Large-scale Dataset for News Recommendation". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Ed. by D. Jurafsky, J. Chai, N. Schluter, J. R. Tetreault. Association for Computational Linguistics, 2020, pp. 3597–3606. DOI: 10.18653/v1/2020.acl-main.331. URL: https://doi.org/10.18653/v1/2020.acl-main.331 (cit. on pp. 2, 12, 21, 33).

All links were last followed on December 24, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Lausanne, 07.01.22, A. Glavackij

place, date, signature