Radboud University Nijmegen

Faculty of Science

Long- and short-term dependencies for sequential recommendation

Master's Thesis

Author:
J.M. van Hulst

Supervisor:
Prof. dr. ir. Arjen P. de Vries

Second reader:
Dr. Faegheh Hasibi

July 2019

Abstract

In this thesis, we study the influence of long- and short-term dependencies for sequential recommendation. The problem of recommendation stems from the enormous amount of content that is provided by companies such as Spotify and Netflix. By treating consumption behaviour of a user as a sequence, the task of recommendation turns into a problem of predicting the next movie and or song in a sequence. Previous work has focused on the usage of various recurrent models and personalized recommendations by integrating user information in a Gated Recurrent Unit (GRU)[1]. Our first contribution consists of the extension of this previous work by performing a thorough analysis of the hyperparameters and methods of preprocessing that were used. We find that the integration of user information is not as significant as was portrayed in previous work. For our second contribution we studied the influence of long- and short-term dependencies on our models. As some recurrent models are said to have difficulties with handling longer sequences [2], we employ a Differentiable Neural Computer, which uses an external memory component. We find that long-term dependencies have less of an influence on recommendation effectiveness than anticipated. As we decrease the size of our training set, while keeping it temporally close to the evaluation set, we find that the decrease in Recall@k scores is not linear, meaning that most of the important information is temporally closest to our evaluation set. When evaluating various sequence lengths, and even when distinguishing between popular, semi-popular and unpopular items, there does not seem to be a clear relation between the resulting scores and the corresponding sequence lengths. We conclude that if runtime is key, one might consider using a smaller dataset and shortening sequence lengths. This could be of importance to companies, such as Netflix, which deal with a tremendous amount of data.

Contents

Chapter 1

Introduction

In this current day and age, people are flooded with media. Platforms such as Netflix and Spotify offer thousands of songs and videos to their following. As such, it becomes increasingly hard for users to choose what they would find entertaining. Recommender systems can ease this pain by suggesting songs and/or movies, which we will refer to as items, by learning from historical user behavior. If we see user behavior as a sequence of actions, where a single action would be the consumption of an arbitrary item, then recommendation becomes a problem of predicting the next item in a particular sequence.
Recurrent Neural Nets (RNNs) are commonly used in sequence prediction tasks, such as NLP studies, where a RNN is used to e.g. predict the next word in a sentence. Just like predicting the next word in the sentence, it can also be used to predict the next item in a sequence for a particular user.
Traditional RNNs suffer from the vanishing gradient problem, which is a problem where the gradient becomes too small for the weights to be updated effectively. The Gated Recurrent Unit (GRU), originally introduced by Cho et al. [3] aims to solve this problem. Donkers et al. [1], in their work, focused on the integration of user information to personalize the task of recommendation. Through their experiments they show that the user-based GRU outperforms the vanilla GRU and other more traditional approaches such as Matrix Factorization or K-Nearest Neighbours (KNN). We first analyze and extend their work by performing additional experiments.
In previous work, the sequences were of length 20, meaning that one assumes that an item that was consumed 20 timesteps prior to predicting the next to-be consumed item, still influences the prediction of the model. We are keen to ascertain if short- or long-term behaviour of users influences the capabilities of our models. We perform experiments with various sequence lengths (i.e. 5, 10, 20 and 50), which allows us to capture the importance of long- and short-term dependencies for the task of recommendation. According to Graves et al. [2], RNNs have difficulties with storing and accessing information over longer time

periods and in their work they show that for the copy-task[1], the LSTM fails to perfectly reproduce sequences with a sequence length that is larger than 20. As we are experimenting with longer sequence lengths than previous work, we are unsure if the GRU will be able to perform adequately when dealing with longer sequences as it is similar to the LSTM[2]. To supplement this, we experiment with the Differentiable Neural Computer (DNC), which is a recurrent network with an additional external memory component that is used to store sequence information. Our experiments show whether or not various sequence lengths influence the predictive prowess of our models and if a model that is developed to handle longer sequences has a positive impact on the given task.
To summarize, the contribution of this work is twofold. First, we extend the work, performed by Donkers et al. to further investigate the LastFM dataset, the hyperparameters that were found for the GRU and the importance of the user dimension for the task of recommendation. Secondly, we find whether long- and short-term dependencies are of importance for the task of sequential recommendation and how well each sequence length fares with respect to the popularity of items. As GRU networks are known to have difficulties with longer sequences [2], we also employ a DNC.
The remainder of this report is divided into several chapters. Chapter 2 discusses related work for the task of recommendation and the models that we employ. Section 3 elaborates on our experimental setup. Section 4 discusses our results, which are divided into the extension of previous work and our additional work regarding variable sequence length. Finally, in chapter 5 and 6 we conclude our work and discuss future work. The code for this work can be found on Github[3].

[1]A task where the model gets fed a sequence, after which it is expected to reproduce the sequence perfectly.

[2]The main difference here is that the LSTM has an additional Sigmoid gate that governs the inner workings of the model.

[3]`https://github.com/mickvanhulst/long-short-term-dependencies-recommender-systems`

Chapter 2

Related work

This chapter introduces related literature, divided into two sections. The first section discusses previous work related to sequence prediction for the task of recommendation. The second section discusses previous work related to the DNC.

2.1 Recommender task

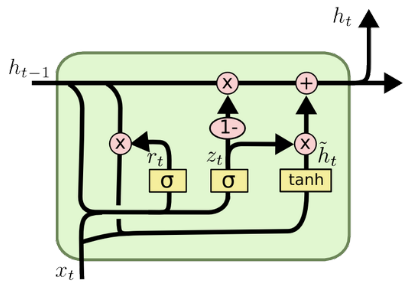
As mentioned in the introduction, this work turns the recommendation problem into a sequential prediction problem. This approach bases itself on the temporal dimension, which carries with it, an inherent ordering, consisting of the consumption behaviour of a particular user.
When training RNNs with long sequential time series data, the vanishing gradient problem may arise. This problem arises, if the gradient of the error with respect to the model's parameters at early time steps goes to zero (as a result of multiplying too many numbers that are < 1). Cho et al. [3] and Hochreiter et al. [4] introduced the Gated Recurrent Unit (GRU) and Long short-term memory (LSTM) respectively, which both aim to solve this problem.
Devooght et al. [5], in their work, focused on using vanilla LSTM networks and solely used this inherent ordering to recommend items based on sequences. Donkers et al. [1] performed similar experiments, but they used GRU networks, rather than LSTM networks. Donkers et al. also experimented with a GRU which incorporated user information and showed that by incorporating such information, the measured scores improved. Lastly, Wu et al. [6] implemented a RNN for the Netflix challenge such that they could predict ratings for user/item combinations. A point of critique here is that the inputs for their network consisted of tuples with values (user, movie, timestamp). Looking at the data that was used, it is clear that there was a temporal overlap between training and test data. As such, part of the timestamp information that was used for training was also available in the test data. In their report, they claim to be able to 'predict future trajectories', but this seems a bit far fetched as future information was

used. In real world scenarios, this information would not be available.
The aforementioned previous work all focused on as coined by Villatel et al. [7], short-term predictions. Villatel et al. their work focused on extending that research by also considering long-term predictions. In their work, short-term predictions were defined as predicting the next item in the sequence, while long-term predictions were defined as a set of most probable next occurring items in a particular sequence occurring further along the consumption behaviour of a user. Although this work focuses on short-term predictions, meaning the prediction of the next item in the sequence, future work could include the enhancement of this work by performing experiments regarding long-term predictions.

2.2 GRU

The GRU, originally introduced by Cho et al. [3] aims to solve the vanishing gradient problem. To understand how Donkers et al. [1] have altered the standard GRU by adding a user-dimension, it is important to understand the inner workings of the vanilla GRU and how the gates that govern the GRU interact. The GRU has two gates, namely the update and the reset gate. As can be seen in Figure 2.1 [8], they both receive a concatenation, consisting of the previous hidden state (zero if the current item is the first item of the sequence) and an item originating from a sequence. After this concatenation is processed by a linear layer, the output is fed to the Sigmoid gate. These Sigmoid functions squash the inputs between zero and one to determine how much of an influence the current input has over updating the current hidden state. To find candidate values to add to the hidden state, the tanh activation function is used. The output of this function can be positive or negative, which allows for an increase or decrease of the state. The last step updates the hidden state, which consists of a multiplication of the update gate with the output of the tanh activation, thus determining how much of a change will occur and if it will be positive or negative. This result is added to a multiplication of the reset gate and the current hidden state[1], which in turn returns the new hidden state.

[1]If the current hidden state is zero, which occurs if the item being processed is the first item out of a sequence, then the hidden state becomes the output of the multiplication of the update gate with the output of the tanh activation.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2.1: GRU cell with equations

Donkers et al. [1] proposed three methods for incorporating user information. This work focuses on the one that performed best, namely the attentional user-based GRU. This model incorporates an additional (Sigmoid) gate which determines the influence of item-based and user-based information. Figure 2.2 shows this process. One can observe that the user- and item-input complement each other, meaning that if the influence of the user-input is low, then the influence of the item-input is high (and vice-versa). After the aforementioned process, the remainder of the internal workings of the attentional user-based GRU is identical to that of the vanilla GRU.

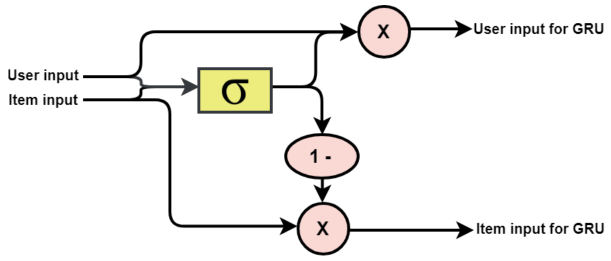


Figure 2.2: Attentional user gate

As we have now explained the internal workings of both the GRU and the user-based GRU, we feel it is of importance to elaborate on the intuition behind why these models are used for the task of recommendation. The GRU is a RNN, meaning that it processes sequential information. As we have stated prior to this section, consumption behaviour of a particular user is the result of the inherent ordering of the temporal dimension. By using a recurrent model, we can utilize this dimension and process the information in the same order as the user consumed it. The usage of the user-based information, which was proposed by Donkers et al. [1], comes from the intuition of personalization, meaning that we may find recurrences in the consumption behaviour of a specific user.

2.3 DNC

As stated by Graves et al. [2], RNNs have difficulties storing and accessing information over longer time periods. A type of model to address this issue, is called a Memory Augmented Neural Network (MANN), which uses an external memory-component. This allows for more ‘slots’ to store information. This research focuses on a differentiable MANN, namely the DNC, which was originally introduced by Graves et al. [9]. The intuition behind using this model for the recommender problem is similar to that of the GRU. The difference here is that the DNC has the capability to store additional information in its memory component, which we expect to be of importance when dealing with longer sequences.
The sections below briefly introduce the DNC, its components, why these components might be suitable for the task of recommendation and improvements proposed in the literature. As a brief introduction to the model might not be sufficient to fully grasp the mechanics of the DNC, we encourage readers to read the original paper by Graves et al. [9] and the bit-by-bit guide to the equations of the DNC [10]. These sources grant further insight into the foundations of the DNC.

2.3.1 Overview

A DNC consists of three main components, namely a controller, the memory component and the output of the DNC. As we are working with sequences, the operations are recurrent. This means that the model does not only use an input item, but also prior information which is stored by the memory component. An important note here is that the memory is not global, but is reinitialized for each sequence that is processed. The only global components are the weights that are improved iteratively, which is why it is highly important that the memory component is also differentiable. As the DNC gets fed a single input item, which originates from a sequence, it combines this with the current memory state and then feeds this to the controller. The controller, in our case, is either a vanilla or user-based GRU. This allows for a comparison between both models with and without the extra memory capacity. The output of the controller is used twice. First, the output is used to update the memory component, whose inner workings are explained in a separate section due to its complexity. Second, the output of the controller is added to the output of the memory to form the final output of the DNC. This process is repeated for each item in the sequence up until the entire sequence is processed. Figure 2.3 illustrates this process.

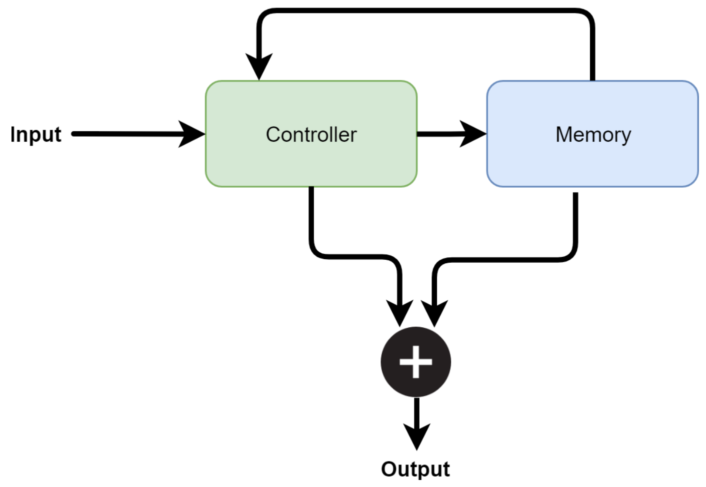


Figure 2.3: Illustration process DNC

2.3.2 Memory component

The memory component is composed out of memory cells. The number of memory cells and the length of each cell is predefined by the user. The DNC performs two operations on the memory component, namely a read and a write operation.
The *read operation* is performed by a read head. A read head is a set of weights that is used to address the memory component. The DNC has at least one read head, where each read head has its own respective weights. These weightings are used to access memory slots. The read head generates a distribution over all the memory locations and will thus gain partial access to each memory cell.
The *write operation* is performed by a write head. The DNC has a singular write head, and, as with the read heads, the write head consists of a set of weights that is used to determine to which degree each memory location can be written to at a certain timestep. Just as the read head, the write head generates a distribution over all the memory locations.
Both the read and write operations require weightings. These weightings are obtained by using three memory addressing schemes.
Content-based addressing: The first memory addressing scheme is called content-based addressing. This allows for access to memory locations which are similar to a given lookup key $\mathbf{k}$. The lookup key $\mathbf{k}$ is the result of processing the output of the controller. Equation 2.1 is used to determine how similar the output of the controller is to a particular memory cell. Besides the lookup key $\mathbf{k}$, $\boldsymbol{M}$ denotes the memory matrix, β denotes a scalar with $\beta \in [1, \infty)$ and $\mathcal{D}$ is defined as the cosine similarity.

$$\mathcal{C}(\mathbf{M}, \mathbf{k}, \beta)[i] = \frac{\exp\left(\mathcal{D}(\boldsymbol{k}, \boldsymbol{M}[i, \cdot])\right)^{\beta}}{\sum_{j=1}^{N} \exp\left(\mathcal{D}(\boldsymbol{k}, \boldsymbol{M}[j, \cdot])\right)^{\beta}} \tag{2.1}$$

The cosine similarity can be seen in Equation 2.2. If two vectors have the same orientation, then the angle between these vectors is zero degrees. This results in the maximum value of the function, which is 1. If they have the exact opposite direction, then $\mathcal{D}$ returns the minimum value, which is -1. An inherent property of the cosine similarity is that the resulting score does not base itself on the magnitude of the vector, but bases its score solely on the orientation of two vectors [10].

$$\mathcal{D}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|} \tag{2.2}$$

If one were to only use content-based addressing, then this would restrict the capability of the write and read heads to access memory. An example here is that the task of copying a sequence, such that the DNC repeats the sequence completely once the input sequence has finished, requires a form of incremental addressing. This cannot be done by content-based addressing, and, as such, the read and write heads combine content-based addressing with other memory addressing schemes. Content-based addressing is combined with dynamic memory allocation and temporal memory linkage for reading and writing operations respectively [10].
For the task of recommendation this component's importance comes from the fact that it is able to find memory cells that are similar to previous inputs, meaning that it finds similarities between the current input and information that has already been stored, which offers a form of generalisation.
Temporal memory linkage: The second memory addressing scheme is called temporal memory linkage. Temporal memory linkage bases its approach on the temporal linkage matrix. The temporal linkage matrix stores probabilities that indicate if a certain memory location was written to after another memory location. If we denote i as the rows and j as the columns of the matrix, then the probability that location i was written to after location j is stored as $L_t(i, j)$, where L_t is the temporal linkage matrix at timestep t. Using this matrix, we can then move backward or forward in time by shifting attention to locations written before or after a given weight matrix[2] for a given timestep. The resulting weightings are called backward and forward weightings respectively. As mentioned previously, the weightings that are generated by the temporal memory linkage scheme are combined with the content-based weightings to compute the final read weighting. This weighting is used to produce the output of the memory component that is combined with the output of the controller to produce the final output of the DNC for a particular timestep [10].
For the task of recommendation this component's importance comes from the fact that it is able to perform incremental addressing. If, for example, several items often occur subsequently, then the temporal linkage matrix is responsible

[2]This weight matrix consists of the previous write weighting for $t-1$.

for capturing this phenomenon.
Dynamic memory allocation: The third and last memory addressing scheme is called dynamic memory allocation. The objective of this scheme is to find an allocation weighting that indicates to what degree each location can be written to and does this by expressing how much an arbitrary memory cell is being used by the model. To compute this allocation weighting, the model first computes a usage vector where each value is between zero and one, indicating how much a position in the memory cell is 'in use'. This vector is sorted in ascending order, which in turn grants a list, consisting of indices that indicate to what extend memory cells are being used. Based on the previous calculations, the allocation weighting is calculated which in turn is combined with the results of the content-based addressing scheme to generate the write weighting. This write weighting is used to compute the new memory matrix at a particular timestep [10].
For the task of recommendation this component's importance is not as clear as the other memory-addressing schemes as it is responsible for writing, rather than reading. Reading operations directly influence the output and aid us in obtaining a prediction. Dynamic-memory allocation is solely responsible for which memory cells will be altered. It does, however, prevent that memory cells that carry important information for subsequent reading operations, are altered. This indirectly results in better predictions.

2.3.3 Proposed improvements

After the initial publication by Graves et al. Csordas et al. [11] proposed three changes with respect to memory addressing and supplied a PyTorch implementation of the DNC which is used and altered accordingly for this work. The first proposal is regarding *content-based addressing*, this component is used to find memory cells that are similar to a given look-up key. To find these similar cells, a similarity score is calculated (see Equation 2.1). Here the entire key and entire cell value are compared to produce a similarity score. This means that the cell value is also used in the normalization part of the cosine similarity measure, while it is unknown during search time, resulting in an unpredictable score. To solve this, Csordas et al. propose the masking of the part that is unknown and should not be used in the query, which results in more dynamic key-value separation. The second proposal is regarding *deallocation and content-based look-up*. Within the DNC, the usage of all memory cells are recorded. The computation of this usage vector is based on two opposite forces acting in different directions. The decremental force acts when a certain location has been read at the previous timestep as that may be an indication that the contents of a certain location are no longer required. The incremental force acts when a location was written to at the previous timestep as the location has not yet had a chance to be read. When allocating memory, the cell with the lowest usage is chosen for allocation [10]. Deallocation of such a cell is done by element-wise multiplication with a retention vector. This indicates how much of the current memory should be kept. The problem here is that it does not affect the actual memory, but just the usage vector, allowing the content-based look-up to find the deallocated data. To solve this, Csordas et al. propose to zero out the memory contents by multiplying every cell of the memory matrix by the corresponding element of the retention vector. The third and last proposal is regarding the *Sharpness of the Temporal Link Distributions*. The temporal linkage matrix is used to read memory cells in the same order as they were written. On every write, all elements of this matrix are updated, meaning that the links related to previous writes are weakened and the new links are strengthened. If the weight that is used for the temporal linkage matrix is not one-hot, links for all non-zero addresses will be reduced and the noise from the current write will be included. When performing multiple iterations, the linkage matrix is multiplied by itself, making the problem worse as the noise is also included. This ultimately results in a flat distribution for the long-term-present cells. Csordas et al. propose the exponentiation and re-normalization of the temporal linkage distribution, such that the model is able to control the importance of non-dominant elements.

Chapter 3

Experimental setup

3.1 Data

The experiments in this thesis use the freely available LastFM dataset, consisting of approximately 19.5 million song played by a set of nearly one thousand users[1]. The dataset contains tuple values of the format (user, timestamp, artist, song). This dataset was also used by Donkers et al. [1], making it a fitting dataset to compare results. Another reason for choosing this dataset is mentioned in Donkers et al. their work, where they mention that in one session, users generally listen to more than one song in a row, e.g. they listen to entire playlists and/or albums in one sitting. Factors such as the mood of a user determine which title will be listened to next. Donkers et al. compare it to the underlying grammar in NLP studies, where there exist dependencies between succeeding events. Such relations are much less obvious when recommending movies as people in this case generally do not watch multiple items in one sitting.
The dataset can be divided into ten disjoint datasets, which will be used separately for our experiments. Each subset, consisting of ten percent of the total dataset, is divided into a training, validation and test set such that the dataset sizes match the previous work performed by Donkers et al. [1]. This means that for each subset, 90 percent of the user's sequence is used for training and the remaining 10 percent is equally divided among the validation and test set. Figure 3.1 shows how the different data subsets are used. All data folds are preprocessed by removing items (i.e. songs) that occur less than 20 times, meaning that if an item does not occur 20 times in a fold, it is removed. Items that do not occur in the training set, are removed from the test and validation set.

[1]`https://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/`, last accessed date: 22-06-2019

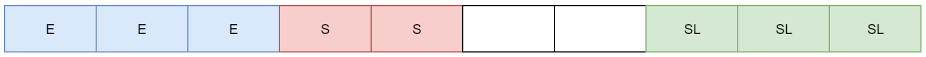


Figure 3.1: Subsets for various experiments (E: Embedding size experiment, S: Various sizes training set, SL: Various sequence lengths). Note that the two empty boxes depict 20% of the unused data of the dataset, with the exception of the first experiment, which uses the entire dataset.

3.2 K-Fold Cross Validation

To ensure that the results are not solely the result of choosing a specific dataset, we perform K-Fold cross validation for our experiments. As aforementioned, the dataset can be divided into ten disjoint dataset, which we utilize for cross validation. Figure 3.2 shows our training process of a singular. The number of epochs are found by training a model on the training set and observing when the model starts to overfit by using the validation set. As soon as this occurs, the model stops the training process and stores the number of epochs, after which a new model is trained on a merged dataset, consisting of the training and validation set. This procedure is often referred to as EarlyStoppage. The final model is then used to predict on the test set.

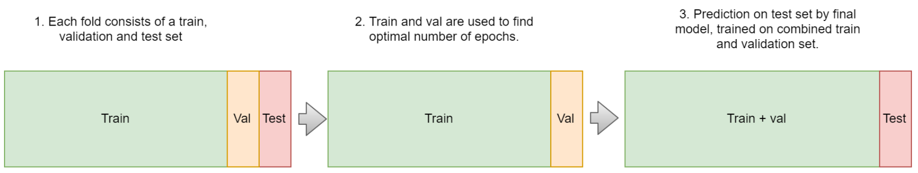

1. Each fold consists of a train, validation and test set
2. Train and val are used to find optimal number of epochs.
3. Prediction on test set by final model, trained on combined train and validation set.
Train
Val
Test
Train
Val
Train + val
Test

Figure 3.2: Training process per fold.

3.3 Hyperparameters

The hyperparameters for the GRU, are copied from Donkers et al. [1] their work. This means that our GRU and thus also the controller consists of one layer with a hidden dimension of 1000. The embedding layer has the same dimension as the hidden layer. To prevent our model from overfitting, we apply dropout to both the user and item dimension, with a keep probability of 0.8 and 0.5 respectively. Furthermore, we apply a gradient cap of 5.0 and apply weight decay to the parameters of the user-based dimension, which is set to 0.01. After experiment 3, we fix our hyperparameters due to computation limitations. This means that we have not performed a hyperparameter optimization step for the DNC. We set the number of read heads, the memory size and the memory width and to 1, 128 and 128 respectively. Additionally, the size of the embedding and hidden layer is set to 128. The set of hyperparameters is summarized in Table 3.1 and 3.2. Similar to Donkers et al. their work, we employ the Adam optimizer with a learning rate of 0.001 for both models.

Table 3.1: Hyperparameters models

Component	**Hyperparameter**	**Exp 1-3**	**Exp 4**
GRU/Controller DNC	Embedding Size	1000	128
	Hidden size	1000	128
	User drop-out	0.5	0.5
	Item drop-out	0.8	0.8
	Layers	1	1
Memory DNC	Memory size	-	128
	Memory width	-	128
	Number of read heads	-	1

Table 3.2: Hyperparameters for training

Hyperparameter	**Value**
Batch size	1000
Learning rate	0.001
User weight decay	0.01
Gradient cap	5

3.4 Evaluation

We evaluate our test set by calculating the Recall@k score, with $k = 20$ and $k = 1$. The Recall@20, in the case of this work, measures if the true class is in the 20 most-likely recommended items by the model for a particular data point. The Recall@1 measures if the most-likely recommended item was in fact the true class. In Information Retrieval studies, the measure that is used is also referred to as the Success Rate (SR). We will, however, refrain from using this reference, such that our work aligns with previous work.
In comparison to previous work performed by Donkers et al. [1], we do not report the Mean Reciprocal Rank (MRR). The reason for this is described by Fuhr in [12], where he describes that the difference between the ranks 1 and 2 is the same as between the rank 2 and ∞. Reciprocal Rank is not an interval scale, but an ordinal scale and one cannot meaningfully compute the mean for an ordinal scale.
We employ the Categorical cross-entropy loss function for the optimization of our models.

Chapter 4

Results & Discussion

This chapter is divided into two sections. The first section focuses its attention on the extension of previous work performed by Donkers et al. [1]. Here we do not strive to improve the work, but rather find meaning behind the choices that were made by the respective work. The second section focuses on exploring the influence of long- and short-term dependencies for the recommender task.

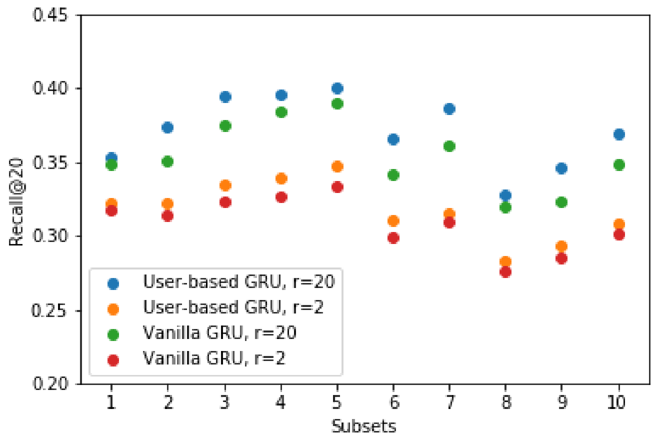
The results that are reported in this chapter are also documented in table format in Appendix A, which is for readers which want to have insight in the specific scores that were found during our experiments.

4.1 Extension previous work

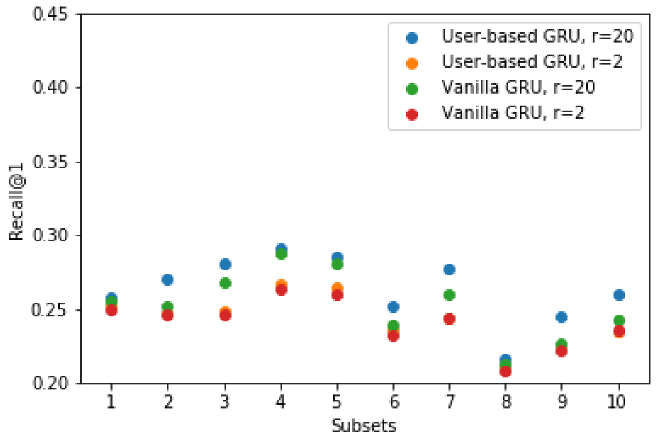
4.1.1 Experiment 1: Dataset sample

The work of Donkers et al. [1] introduced a user-based GRU. As previously described, we focus on the attentional user-based GRU, which is a GRU with an additional attentional gate that determines the importance of the user information for a particular user-item combination. In their work, the authors took a subsample of 10 percent of the original training data, which suggests a random subsample. A brief exchange by email with the authors clarified that rare items were also removed, but it remains unclear how their sample was taken exactly, and what the threshold was for removing rare items. It is, however, highly important to know how a sample was taken. If temporal boundaries were not respected, meaning that the sample was random, then user consumption patterns were broken. To explore the dataset, we perform a 10 fold operation with the user-based GRU and vanilla GRU over the entire dataset and report the results. Figure 4.1 shows the results for the aforementioned models on the 10 disjoint subsets of the LastFM dataset. We observe a variance between the results of the two models for a particular subset, and between the various subsets. We also observe that our scores are much higher than those reported in [1] and that the difference between the scores of the user-based and vanilla GRU is not as large, although one can observe that the user-based GRU always outperforms

the vanilla GRU.
We performed some preliminary tests such as refraining from the usage of EarlyStoppage and training for the same number of epochs as reported in [1], which had little impact on the resulting Recall@k scores. We hypothesized that the higher scores might also be the result of our threshold setting for removing items that occur scarcely. As we decrease the threshold setting, it becomes increasingly difficult for the model to make correct predictions, making the problem more complex. We expected the importance of user information to increase in this setting as the model is expected to make predictions for items that occur rarely, resulting in a decrease in Recall@20 scores, but most importantly in a larger gap between the scores of the two models. This would show that with a lower threshold, the importance of the user-based model increases. Figure 4.1 shows the scores with the threshold of removing items, which we coined as $r = 2$ and $r = 20$, meaning that items had to occur at least two or twenty times respectively. We observe that the scores for $r = 2$ are higher than those reported in [1] and that the significance of the user-based model does not increase with a lower threshold. This leads us to the conclusion that our hypothesis was false. After these extensive preliminary tests, we conclude that we are unable to reproduce the scores of previous work, meaning that our scores indicate that the performance of the attentional user-based GRU is less significant in comparison to previous work, which is most likely due to a different method of preprocessing the data.



(a) Recall@20

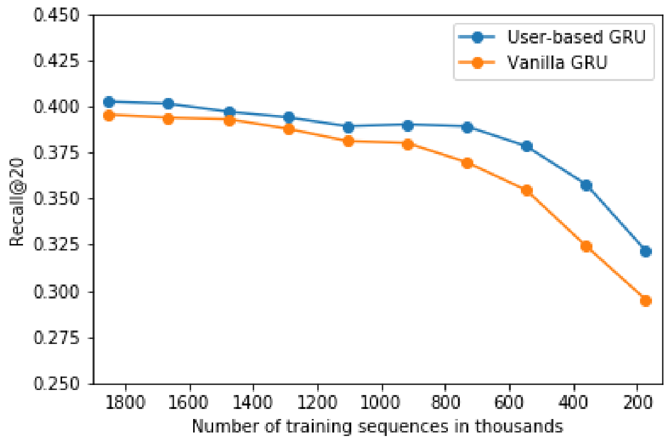


(b) Recall@1

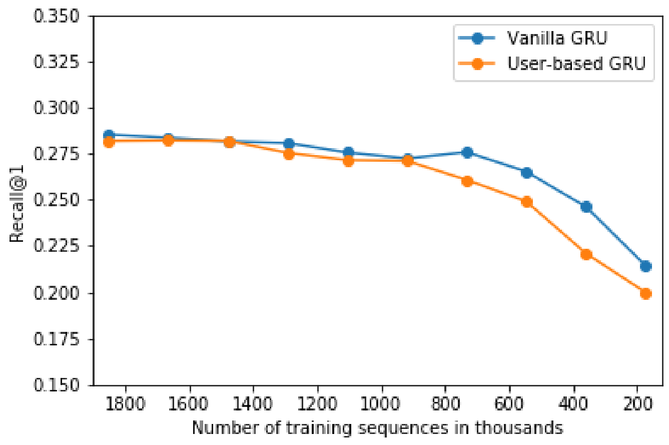
Figure 4.1: Recall@k for ten disjoint folds in temporally increasing order.

4.1.2 Experiment 2: Subsample size

Previous work used a subsample of 10 percent for their work [1], but did not vary the amount of data for training. For this experiment, we measure the impact of the amount of training data on the predictive performance of our models. To perform this experiment, we took 20 percent of the dataset (see Figure 3.1) and carried out 10 different experiments where for each experiment the length of the training sequence for each user was decreased by 10 percent, while the validation and test set remained unchanged for all experiments. To make this experiment as fair as possible, the 20 percent subsample was preprocessed as two separate samples of 10 percent each, such that we could remove rare items, after which they were merged and processed equally for the remainder of the experiments. Figure 4.2 shows the resulting Recall@20 and Recall@1 scores on the test set. We observe that for this respective subsample, the Recall@20 does not seem to decrease linearly, indicating that the most important data is hidden in the part temporally closest to the test set. This leads us to two observations, first being that if runtime is of importance, then one might decide to decrease the dataset further as, for example, the difference between 1.8 million and 1 million training sequences only resulted in a decrease of around 0.01 Recall@20. Second being that this shows that it is of importance to merge the validation and training set as the validation set consists of the sequences temporally closest to the test set.



(a) Recall@20



(b) Recall@1

Figure 4.2: Recall@k scores as training data decreases.

4.1.3 Experiment 3: Embedding size

Another finding that peaked our interest was the size of the embedding layer. Google [13], in one of their articles, mentions that it is common for NLP tasks to use $\sqrt[4]{Z}$ to determine the size of the embedding layer, with Z being the number of unique items in a dataset. Per data fold, our number of unique items equals approximately 18 thousand, which means that an embedding size of 1000 is much higher than $\sqrt[4]{18000} \approx 12$. Although this is not an NLP task, it is interesting to see whether or not the general consensus regarding the size

of an embedding layer generalizes to the recommender task. We performed an experiment to test various embedding settings with a subset of our dataset (see Figure 3.1). Figure 4.3 displays the average validation loss over the three folds. We report the validation loss as this is commonly used for hyperparameter optimization, while Recall@k is used as a measure of success. We observe that the loss decreases as the embedding size increases. In comparison to what is advised for NLP tasks, the size of the embedding layer is much higher, which leads us to think that there is less of an obvious relation between the items in the sequences than with, for example, sentences (which are common in NLP tasks). In NLP tasks, embedding is used such that items that occur in the same 'context' are represented near each other in the embedding space. For the task of recommendation, the model does not seem to benefit as much from such a method as one might expect.
Before analyzing our results, we hypothesized that the model might be creating its own one-hot encoding. We, however, observe that for the user-based GRU, the embedding size of 4096 results in a lower validation loss than for the embedding size of 8192. For the vanilla GRU, we also calculated the validation loss when setting the embedding size equal to the number of classes, which grants the GRU the opportunity to create its own one-hot encoding. This resulted in a validation loss of $\approx \mathbf{6.178}$, which shows that the model is not just creating its own one-hot encoding, but is finding some relationship between the items. We were unable to perform the same test for the user-based GRU due to limitations in the available computational resources. We, however, find this additional test unnecessary as we already determined that a lower embedding size resulted in a lower validation loss. These results show that our hypothesis was false and that there is a relation between the items, but as aforementioned, it is most likely not as present as for NLP tasks, resulting in a higher optimal value for the size of the embedding layer.

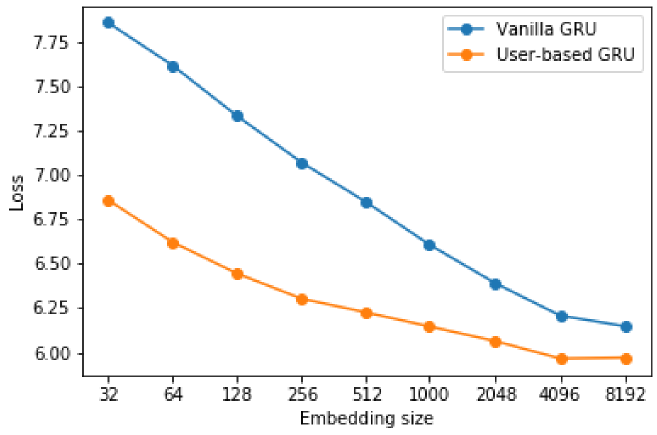
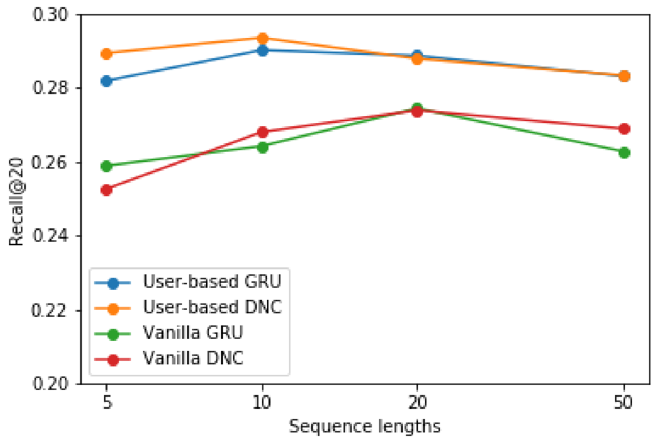


Figure 4.3: Validation loss as embedding size increases.

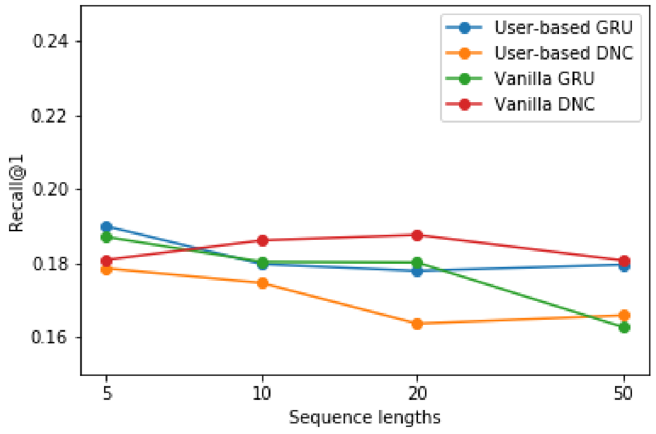
4.2 Experiment 4: Variable sequence lengths

We experimented with various sequence lengths, to find whether or not short- and long-term dependencies influence the predictive prowess of our model. Contrast to the previous experiments, we also employ the DNC for this experiment as it was mentioned to more effectively memorize longer sequences than other RNNs[2].
Figure 4.4 shows the results for sequence lengths 10, 20, 50. At first we were tempted to also test sequence length of 100. During the training process for the user-based model GRU, however, we noticed that EarlyStoppage was triggered early, causing a huge drop in the Recall@k. This lead us to assume that for the user-based GRU, the validation set of sequence length 100 was not a good representation of the training set, causing it to halt the training process early. An explanation for this could be that if we have e.g. 700 users and want to create sequences of length 100, then we only have 700 data rows, while with a sliding window approach, we would have 66.500 rows for a sequence length of 5. Enlarging the dataset might further solve this problem, but we leave this for future work. The resulting Recall@k scores are the averaged scores over a 3-Fold operation (see Figure 3.1 for the folds that were used). We observe, that in general the sequence length does not seem to influence the resulting Recall@k scores tremendously. This seems to be an indication that longer sequences and thus long-term dependencies do not result in large increases in the Recall@k scores, meaning what users listened to a longer time ago, does not seem to influence the short-term behaviour of a user. Besides this conclusion, we do make a few observations.
There is a difference between the models and sequence length combinations that perform best with respect to the Recall@k scores. For the Recall@20, a sequence length of 10 seems to perform best for the user-based models, while for the vanilla models, a sequence length of 20 is more optimal. For Recall@1 the best performing sequence length is a sequence length equal of 5 for all models, except for the vanilla DNC. This leads us to our second observation, which is that there is a difference between sequence lengths that perform best for user-based models and for the vanilla models. We are unsure what causes this result, but there is one finding that seems to be explainable.
The vanilla DNC is outperformed by all user-based models for the Recall@20 scores. However, if we look at the Recall@1 scores, the vanilla DNC outperforms all other models for sequence lengths that are larger than 5. If we look at the original tasks for the DNC, like the copy-task, it leads us to think that the DNC is particularly good at memorizing sequences perfectly. When reporting the Recall@20 score, a model is not expected to memorize the sequences perfectly as the true class of the sequence only needs to be in the top 20 most likely predicted items. As such, this gives models a higher chance to guess correctly. When reporting the Recall@1 score, there is less room for error in comparison to the Recall@20. If the DNC is able to memorize certain reoccuring sequences perfectly, then it seems likely that the DNC outperforms the other models, which is something that we are tempted to assume is occurring here. As was mentioned

by Graves et al. [2], when sequences get longer, regular RNNs have difficulties memorizing the sequences, which aligns with our thought process that is elaborated above. Our thought process does not align with the Recall@1 scores which were obtained for the user-based DNC. This model seems to perform poorly for longer sequence lengths. We are unsure why this is the case, but it might be due to the underrepresentation of sequence- and user-combinations, causing the user-based DNC to generalize poorly to the test set.



(a) Recall@20



(b) Recall@1

Figure 4.4: Recall@k scores for variable sequence lengths.

Figure 4.5 depicts the item distribution for the training data of fold 7 (see

Figure 3.1). Here we observe a huge item imbalance, which leads to the hypothesis that the recommender system is primarily good in capturing the listening behaviour from popular items and would thus perform poorly in a (realistic) setting where it is expected to also recommend items that are less popular. To test this hypothesis, we divide our classes into three categories: popular, semi-popular and unpopular. To achieve this, we determine the item occurrences of items for the training set of a particular fold, after which we create three equal sized bins, which we base on the inherent ordering of the list of item occurrences. This means that the weights are unique for each respective fold.

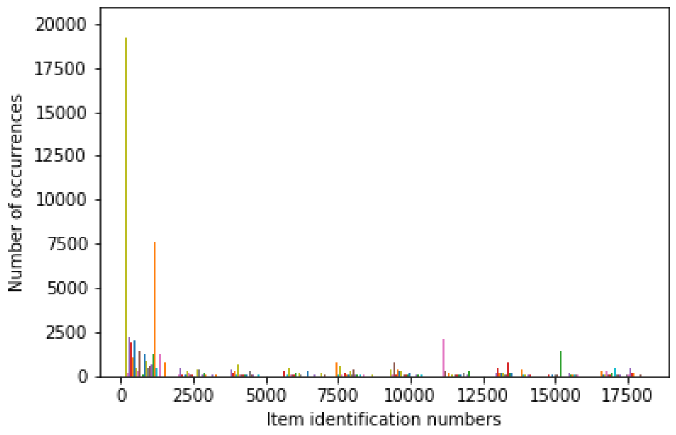
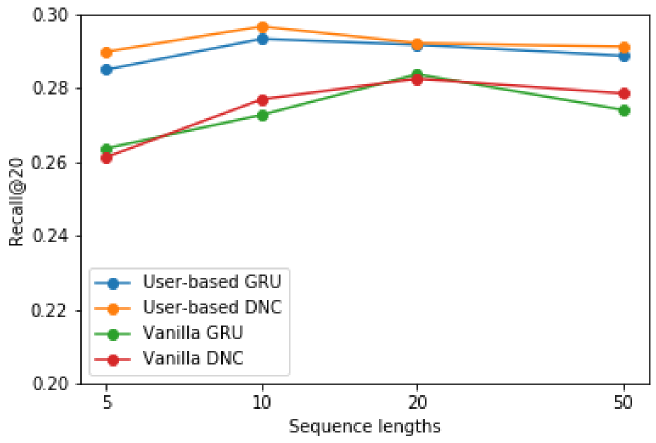
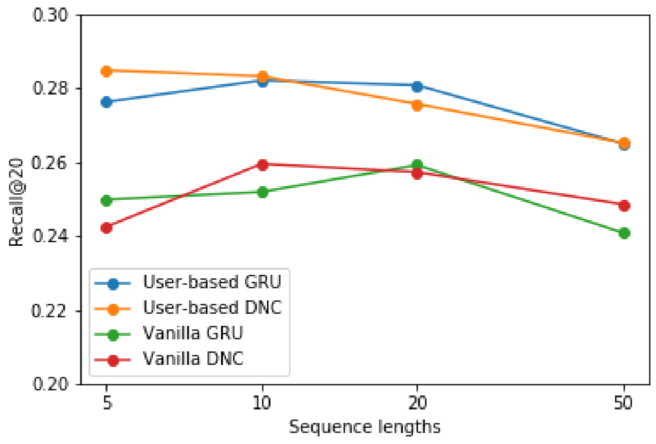


Figure 4.5: Item distribution training set of singular fold

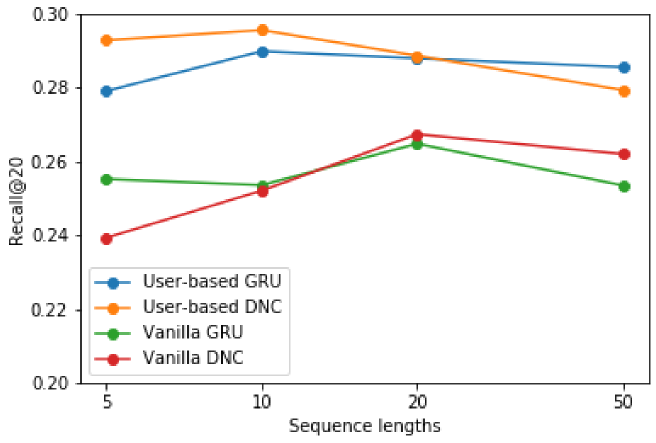
Figures 4.6 and 4.7 show the respective Recall@20 and Recall@1 scores for popular, semi-popular and unpopular items. Our hypothesis, however, regarding the differences in results for the aforementioned categories, seems to be false. The recommender system returns similar results for the aforementioned categories of popularity. Our aforementioned observations seem to also be present for the different categories as, for example, the vanilla DNC outperforms the other models, with respect to the Recall@1, for sequences lengths longer than five. To conclude, even when looking into different categories of popularity, it does not seem like the sequence length has a tremendous influence on the resulting Recall@k scores.



(a) Popular

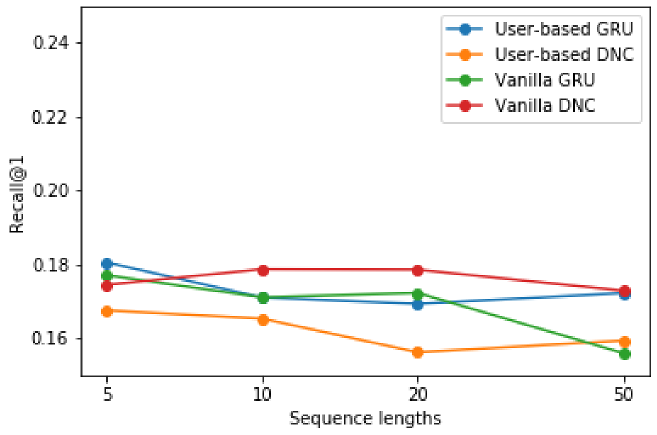


(b) Semi-popular

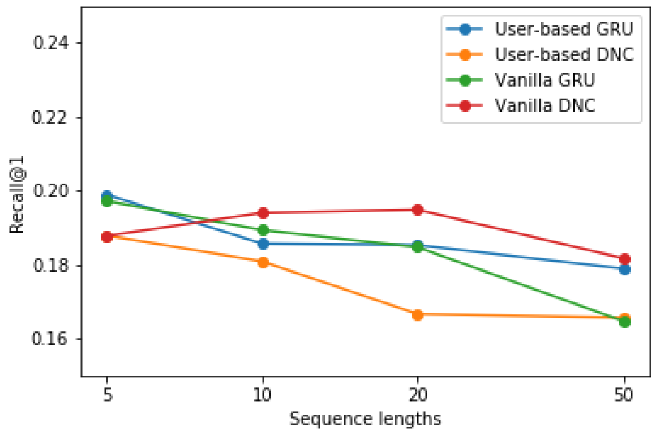


(c) Unpopular

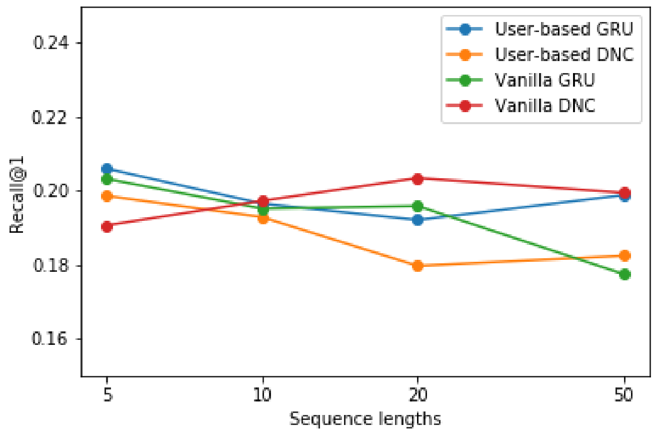
Figure 4.6: Recall@20 scores for popular, semi-popular and unpopular items



(a) Popular



(b) Semi-popular



(c) Unpopular

Figure 4.7: Recall@1 scores for popular, semi-popular and unpopular items

Chapter 5

Conclusion

In this work, we extended previous work [1] by exploring the LastFM dataset and the settings that were used for their experiments. After which we experimented with various sequence lengths, the usage of a model that was shown to generally perform better than other (recurrent) models when dealing with longer sequences [2] and the difference in the models their performance on popular, semi-popular and unpopular items.

We found that our approach reduced the significance of the user-based GRU, which in comparison to previous work, is most likely due to a different method of preprocessing the data.

As the embedding size increased, we found that the respective validation loss also increased, which seems to be an indication that, in comparison to e.g. NLP tasks, there is less of an relationship between items in sequences for the task of recommendation.

Long-term dependencies did not seem to be an important factor for our models. As we decreased the size of the training set, while keeping it temporally close to the evaluation set, the scores did not decrease linearly. This is an indication that a big portion of the data that results in our Recall@k scores is present in the data temporally closest to the test set (i.e. short-term dependencies). Furthermore, when increasing sequence lengths and when reporting several categories with respect to popularity, we found that there does not seem to be a clear relation between different sequence lengths and an increase or decrease in the resulting Recall@k scores. The vanilla DNC was shown to have higher Recall@1 scores than other models for longer sequence lengths. When reporting the Recall@1 score, there is less room for error in comparison to the Recall@20. We explain the results by assuming that the DNC was able to memorize the reoccuring sequences perfectly, resulting in the vanilla DNC outperforming other models.

We conclude that when working with sequential data, it is highly important to respect the temporal boundaries as failing to do so, might result in scores that cannot be used to evaluate recommender systems. There seems to be an indication that a large portion of the information that is responsible for the resulting Recall@k scores is temporally closest to the evaluation set, meaning that if runtime is key, one could consider using a smaller dataset and shortening sequence lengths. This could be of importance to companies, such as Netflix, which deal with a tremendous amount of data.

An important note here is that the results for experiment 4 are subject to discussion as we fixed our hyperparameters due to the limited computation resources available to carry out this study.

Chapter 6

Future work

In future work, we plan to extend our research by performing a full hyperparameter search, which may be the reason why we found that longer sequences did not seem to clearly increase the performance of our models. This does, however, mean that if the models for longer sequence lengths become more complex, then training time would increase even more then it already has[1].

Besides the improvements of our work, we also see room for improvement by incorporating the temporal dimension in our models, which was suggested in Donkers et al. their work[1]. An example of integrating this dimension could be the time difference between consuming items by users. This could perhaps teach the model to recognize when sessions end, and therefore better anticipate the likely changes in consumption behaviour.

[1]As aforementioned, training models took increasingly longer as we increased the sequence length

Bibliography

[1] T. Donkers, B. Loepp, and J. Ziegler, "Sequential user-based recurrent neural network recommendations," in *Proceedings of the Eleventh ACM Conference on Recommender Systems*, pp. 152–160, ACM, 2017.

[2] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[3] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[5] R. Devooght and H. Bersini, "Collaborative filtering with recurrent neural networks," *arXiv preprint arXiv:1608.07400*, 2016.

[6] C.-Y. Wu, A. Ahmed, A. Beutel, A. J. Smola, and H. Jing, "Recurrent recommender networks," in *Proceedings of the tenth ACM international conference on web search and data mining*, pp. 495–503, ACM, 2017.

[7] K. Villatel, E. Smirnova, J. Mary, and P. Preux, "Recurrent neural networks for long and short-term sequential recommendation," *arXiv preprint arXiv:1807.09142*, 2018.

[8] M. Thapliyal, "Deep learning basics: Gated recurrent unit (gru)," Jan 2019.

[9] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.

[10] J. A. Prez-Ortiz, "A bit-by-bit guide to the equations governing differentiable neural computers." `http://github.io/jaspock/dnc`, 2017. Accessed: October 20 2017.

[11] R. Csordás and J. Schmidhuber, "Improved addressing in the differentiable neural computer," 2019.

[12] N. Fuhr, "Some common mistakes in ir evaluation, and how they can be avoided," in *ACM SIGIR Forum*, vol. 51, pp. 32–41, ACM, 2018.

[13] "Feature columns." `https://www.tensorflow.org/guide/feature_columns`. [Online; accessed 22-06-2019].

Appendices

Appendix A

Tables results experiments

A.1 Experiment 1: Dataset sample

Table A.1: Recall@20 scores for 10 different subsets

Fold	**User-based GRU r=20**	**Vanilla GRU r=20**	**User-based GRU r=2**	**Vanilla GRU r=2**
1	0.353	0.349	0.322	0.317
2	0.373	0.350	0.322	0.314
3	0.395	0.375	0.335	0.322
4	0.395	0.384	0.339	0.327
5	0.400	0.390	0.347	0.334
6	0.366	0.341	0.310	0.299
7	0.386	0.361	0.315	0.309
8	0.328	0.319	0.283	0.276
9	0.346	0.323	0.293	0.285
10	0.369	0.348	0.308	0.301

Table A.2: Recall@1 scores for 10 different subsets

Fold	**User-based GRU r=20**	**Vanilla GRU r=20**	**User-based GRU r=2**	**Vanilla GRU r=2**
1	0.257	0.255	0.251	0.249
2	0.270	0.251	0.247	0.245
3	0.280	0.267	0.248	0.246
4	0.291	0.287	0.267	0.264
5	0.285	0.280	0.264	0.260
6	0.252	0.239	0.236	0.233
7	0.277	0.260	0.244	0.244
8	0.216	0.213	0.209	0.208
9	0.244	0.227	0.223	0.222
10	0.259	0.243	0.235	0.235

A.2 Experiment 2: Subsample size

Table A.3: Recall@20 scores for different sizes for the training set

User-based GRU	**Vanilla GRU**	**Size training set in thousands**
0.402	0.395	1851
0.401	0.394	1665
0.397	0.393	1478
0.394	0.388	1292
0.389	0.381	1106
0.390	0.380	919
0.389	0.370	733
0.378	0.355	547
0.358	0.324	360
0.322	0.295	174

Table A.4: Recall@1 scores for different sizes for the training set

User-based GRU	**Vanilla GRU**	**Size training set in thousands**
0.285	0.282	1851.0
0.284	0.282	1665.0
0.282	0.282	1478.0
0.281	0.275	1292.0
0.276	0.271	1106.0
0.272	0.271	919.0
0.276	0.261	733.0
0.265	0.249	547.0
0.246	0.221	360.0
0.214	0.200	174.0

A.3 Experiment 3: Embedding size

Table A.5: Cross-entropy loss for different embedding size

User-based GRU	Vanilla GRU	Embedding size
6.861	7.861	32.0
6.622	7.620	64.0
6.445	7.334	128.0
6.303	7.072	256.0
6.225	6.847	512.0
6.147	6.610	1000.0
6.064	6.390	2048.0
5.966	6.208	4096.0
5.971	6.148	8192.0
-	6.178	Number of unique classes

A.4 Experiment 4: Variable sequence lengths

A.4.1 Base scores

Table A.6: Recall@20 scores for various sequence lengths

Model type	Sequence length 5	Sequence length 10	Sequence length 20	Sequence length 50
Vanilla GRU	0.253	0.268	0.274	0.269
User-based GRU	0.282	0.290	0.289	0.283
Vanilla DNC	0.259	0.264	0.274	0.263
User-based DNC	0.289	0.293	0.288	0.283

Table A.7: Recall@1 scores for various sequence lengths

Model type	Sequence length 5	Sequence length 10	Sequence length 20	Sequence length 50
Vanilla GRU	0.181	0.186	0.188	0.181
User-based GRU	0.190	0.180	0.178	0.180
Vanilla DNC	0.187	0.180	0.180	0.163
User-based DNC	0.179	0.175	0.164	0.166

A.4.2 Popularity scores

Popular

Table A.8: Recall@20 scores for popular items and various sequence lengths

Model type	**Sequence length 5**	**Sequence length 10**	**Sequence length 20**	**Sequence length 50**
Vanilla GRU	0.261	0.277	0.282	0.278
User-based GRU	0.285	0.293	0.292	0.289
Vanilla DNC	0.264	0.273	0.284	0.274
User-based DNC	0.290	0.297	0.292	0.291

Table A.9: Recall@1 scores for popular items and various sequence lengths

Model type	**Sequence length 5**	**Sequence length 10**	**Sequence length 20**	**Sequence length 50**
Vanilla GRU	0.174	0.179	0.179	0.173
User-based GRU	0.181	0.171	0.169	0.172
Vanilla DNC	0.177	0.171	0.172	0.156
User-based DNC	0.168	0.165	0.156	0.159

Semi-popular

Table A.10: Recall@20 scores for semi-popular items and various sequence lengths

Model type	**Sequence length 5**	**Sequence length 10**	**Sequence length 20**	**Sequence length 50**
Vanilla GRU	0.242	0.259	0.257	0.249
User-based GRU	0.276	0.282	0.281	0.265
Vanilla DNC	0.250	0.252	0.259	0.241
User-based DNC	0.285	0.283	0.276	0.265

Table A.11: Recall@1 scores for semi-popular items and various sequence lengths

Model type	**Sequence length 5**	**Sequence length 10**	**Sequence length 20**	**Sequence length 50**
Vanilla GRU	0.188	0.194	0.195	0.182
User-based GRU	0.199	0.186	0.185	0.179
Vanilla DNC	0.197	0.189	0.185	0.165
User-based DNC	0.188	0.181	0.167	0.166

Unpopular

Table A.12: Recall@20 scores for unpopular items and various sequence lengths

Model type	**Sequence length 5**	**Sequence length 10**	**Sequence length 20**	**Sequence length 50**
Vanilla GRU	0.239	0.252	0.267	0.262
User-based GRU	0.279	0.290	0.288	0.285
Vanilla DNC	0.255	0.254	0.265	0.253
User-based DNC	0.293	0.295	0.289	0.279

Table A.13: Recall@1 scores for unpopular items and various sequence lengths

Model type	**Sequence length 5**	**Sequence length 10**	**Sequence length 20**	**Sequence length 50**
Vanilla GRU	0.191	0.197	0.203	0.199
User-based GRU	0.206	0.197	0.192	0.199
Vanilla DNC	0.203	0.195	0.196	0.177
User-based DNC	0.199	0.193	0.180	0.182