NTNU

Norwegian University of
Science and Technology

Inter-/Intra-session Recurrent Neural Network for Session-based Recommender Systems

Ole Steinar Lillestøl Skrede

Master of Science in Computer Science
Submission date: July 2017
Supervisor: Massimiliano Ruocco, IDI

Norwegian University of Science and Technology
Department of Computer Science

Ole Steinar Lillestøl Skrede

Inter-/Intra-session Recurrent Neural Network for Session-based Recommender Systems

Master's Thesis in Computer Science, Spring 2017

Data and Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology and Electrical Engineering
Norwegian University of Science and Technology
Submission date: July 2017
Supervisor: Massimiliano Ruocco

Abstract

Recommender systems are useful to users of a service and to the company offering the service. Good recommendations can help users find what they are looking for faster, and they can help users discover new content. For businesses, the recommendations can improve user engagement. In recent years, research has been done on employing Recurrent Neural Networks (RNNs) within the case of recommender systems. Results have been promising, especially in the session-based setting where RNNs have been shown to outperform state-of-the-art models. In many of these experiments, the RNN could potentially improve its recommendations by utilizing information about a user's past sessions, in addition to his interactions in the current session. A problem for session based recommenders, is how to produce accurate recommendations at the start of a session, before the system has learned much about the user's current interests.

We propose a novel approach that extends the existing RNN-based session-based recommender, making it able to process the user's recent sessions, to improve recommendations. This is done by using a second RNN layer to learn from recent sessions, and predict the user's interest in the current session. By feeding this information to the original RNN layer, the proposed solution is able to improve its recommendations.

Our experiments on three different datasets, show that the proposed approach can significantly improve recommendations throughout the sessions, compared to a single RNN working only on the current session. The proposed model especially improves recommendations at the start of sessions, and is therefore able to deal with the cold start problem within sessions.

Sammendrag

Anbefalingssystem er nyttige både for brukere av en tjeneste og bedriften som tilbyr tjenesten. Gode anbefalinger kan hjelpe brukere med å lettere finne det de leter etter, og de kan hjelpe brukere med å oppdage nytt innhold. Anbefalingene kan også øke brukerengasjemnt i tjenesten til bedriften. De siste årene har det blitt gjort forskning på bruk av rekurrente nevrale nettverk (RNN) innen anbefalingssystemer. Resultatene har vert lovende, spesielt i sesjonsbaserte tilfeller, der RNN har gjort det bedre enn State of the art modeller. I mange av disse eksperimentene, kunne RNNene potensielt ha forbedret anbefalingene sine ved å benytte informasjon om brukerenes tidligere sesjoner, i tillegg til brukerens interaksjoner i den gjeldende sesjonen. Et problem for sesjonsbaserte anbefalingssystmer er hvordan de kan finne gode anbefalinger tidlig i sesjonen, når systemet enda ikke har lært særlig om brukerens gjeldende interesser.

Vi foreslår en ny metode som utvider det eksisterende RNN baserte sesjonsbaserte anbefalingssystemet slik at det kan bruke brukere sine tidligere sesjoner til å forbedre anbefalingene. Dette gjøres ved å bruke et ekstra RNN lag til å lære fra nylige sesjoner, og forutse brukerens interesser i den nåværende sesjonen. Ved å formidle denne informasjonen til det første RNN laget, kan den foreslåtte modellen forbedre sine anbefalinger.

Eksperimentene våre på tre ulike datasett, viser at den foreslåtte modellen oppnår signifikant forbedrede anbefalinger i sesjoner, sammenlignet med et enkelt RNN som kun forholder seg til hver enkelt sesjon. Den foreslåtte modellen oppnår spesielt forbedrede anbefalinger i begynnelsen av sesjonene, og er dermed i stand til å håndtere kald start problemet innad i sesjoner.

Preface

This thesis was written in the spring of 2017 for the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU).

Many thanks to supervisor Massimiliano Ruocco for his guidance, input, and support. Thanks also to Helge Langseth for his contributions.

Contents

1 Introduction **1**
- 1.1 Background and Motivation . . . 1
- 1.2 Goals and Research Questions . . . 3
- 1.3 Research Method . . . 3
  - 1.3.1 Limitations . . . 4
- 1.4 Contributions . . . 4
- 1.5 Thesis Structure . . . 5

2 Background Theory **7**
- 2.1 Recommender Systems . . . 7
  - 2.1.1 Collaborative Filtering . . . 7
    - Matrix Factorization . . . 8
    - Bayesian Personalized Ranking Matrix Factorization . . . 9
    - Item k Nearest Neighbors . . . 10
  - 2.1.2 Content-based Filtering . . . 11
  - 2.1.3 Cold Start Problem . . . 11
  - 2.1.4 Session-based Recommender Systems . . . 12
- 2.2 Neural Networks . . . 12
  - 2.2.1 Activation Functions . . . 14
  - 2.2.2 Training Neural Networks . . . 14
  - 2.2.3 Dropout . . . 15
  - 2.2.4 Minibatches . . . 15
- 2.3 Deep Learning . . . 15
  - 2.3.1 Recurrent Neural Networks . . . 16
    - LSTM and GRU . . . 18
  - 2.3.2 Item Representations . . . 22
    - One-hot Vectors . . . 22
    - Embeddings . . . 22
- 2.4 Evaluating Recommender Systems . . . 23
  - 2.4.1 Recall . . . 23
  - 2.4.2 MRR . . . 24
- 2.5 Next Basket Recommendation . . . 24
  - 2.5.1 Representing Past User Sessions in Next Basket Recommendation . . . 25

3 Related work **27**
3.1 RNN as a Recommender System . . . 27
3.2 Dealing with Large and Dynamic Item Sets . . . 29
3.3 Temporal Dynamics . . . 29
3.4 Context Aware Systems . . . 31
3.5 Other Relevant Techniques and Approaches . . . 31
3.5.1 Dealing with Long Sessions . . . 31
3.5.2 Feeding Rich Input to the Recommender System . . . 32
3.5.3 Training Recommender Systems Consisting of Multiple RNN Layers . . . 32
3.5.4 Data Augmentation and Privileged Information . . . 33

4 Proposed architecture **35**
4.1 Main Idea: the II-RNN . . . 35
4.2 Problem Formulation . . . 36
4.3 Model Description . . . 37
4.3.1 Intra-session RNN . . . 37
4.3.2 II-RNN . . . 38

5 Experimental Setup **41**
5.1 Experimental Setup . . . 41
5.1.1 Datasets . . . 41
Reddit Dataset . . . 41
Last.fm Dataset . . . 42
Instacart Dataset . . . 42
Preprocessing . . . 42
5.1.2 Baselines . . . 43
Most Popular . . . 43
Most Recent . . . 43
Item-kNN . . . 44
BPR-MF . . . 44
5.1.3 Implementation . . . 44
5.1.4 Experiments . . . 45
First-n Recommendations . . . 45
Creating Mini-batches . . . 46

6 Results and Discussion **47**
6.1 Results . . . 47
6.2 Evaluation . . . 56
6.2.1 Baselines . . . 56
6.2.2 RNN and II-RNN . . . 56
6.2.3 BPR-MF . . . 57

6.2.4 Dropout . . . 57
6.2.5 Average-Pooling and Last Hidden State . . . 57
6.3 Discussion . . . 57
6.3.1 Usefulness of the Inter-Session Level RNN . . . 57
6.3.2 Importance of Retraining the Model . . . 58
6.3.3 Artificial and Natural Sessions . . . 58
6.3.4 Declining Performance on the Instacart Dataset . . . 58

7 Conclusion **59**
7.1 Limitations . . . 59
7.1.1 Response time . . . 59
7.1.2 Scalability . . . 59
7.1.3 Dynamic set of items and item with short lifespans . . . 59
7.1.4 Suggestion to Overcome the Limitations . . . 60
7.2 Contributions . . . 60
7.3 Answering Research Questions . . . 60
7.4 Further work . . . 61
7.4.1 Producing recommendations before the first user interaction 62
7.4.2 Session representations . . . 62
7.4.3 Learning Item Embeddings . . . 62
7.4.4 Recommending Items with Short Lifespans . . . 62
7.4.5 Alternating Training . . . 63
7.4.6 Novel Recommendations . . . 63
7.4.7 Utilizing Contextual Information . . . 63

Bibliography **65**

List of Figures

2.1 A feed-forward network. Examples are inserted into the network by setting the values in the input layer. The values of the nodes in each layer are computed as a function of the values in the nodes in the prior layer. 13
2.2 Sigmoid, ReLU, and tanh activation functions. 14
2.3 A simple RNN. Represented as a loop(left), and unrolled to t timesteps(right). 17
2.4 Simple illustration of a Recurrent Neural Network (RNN). 17
2.5 Illustration of a long short-term memory (LSTM), adapted from [20]. Rectangles represent element-wise operations. Circles and ellipses represent Neural Network (NN) layers. 19
2.6 Illustration of a gated recurrent unit (GRU), adapted from [21]. Rectangles represent element-wise operations. Circles and ellipses represent NN layers. 21

3.1 Illustration of the proposed model in [1]. 28
3.2 RNN using multiple GRU cells, unrolled to three time steps. Any number of sequence lengths is of course possible as with a RNN with one layer of cells. 28
3.3 Adapted illustration of the parallel GRU model used in [6]. 33
3.4 Adapted illustration of the data augmentation used in [5]. 34

4.1 Illustration of the intra-session RNN. 38
4.2 Illustration of the inter-/intra-session RNN (II-RNN). This illustration does not show how the session representations are obtained. 39
4.3 Illustration of the II-RNN with average pooling to create session representations from items. 40
4.4 Illustration of the II-RNN where the last hidden state of the intra-session RNN is stored as the session representation. 40

6.1 First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Reddit dataset, with Recall@5 metric. . . 53
6.2 First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Reddit dataset, with MRR@5 metric. . . 53

6.3 First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Last.fm dataset, with Recall@5 metric. . 54
6.4 First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Last.fm dataset, with MRR@5 metric. . . 54
6.5 First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Instacart dataset, with Recall@5 metric. 55
6.6 First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Instacart dataset, with MRR@5 metric. . 55

List of Tables

1.1 Outline of this thesis. 5

2.1 Illustration of collaborative filtering. User4's rating of Item3 is predicted by using the ratings of Item3 (yellow) from similar users (green). 8
2.2 Simple example of average pooling and maximum pooling. Values for max-pooling are indicated with blue color. 26

5.1 Statistics for our three datasets, after preprocessing. 43
5.2 Best found configurations for the RNN models. We found that the configurations that worked well for the II-RNN, worked well for the standalone intra-session RNN as well. Not all configurations are applicable to the standalone intra-session RNN. 45

6.1 Recall and MRR scores for the RNN models and the baselines. Relative scores are given compared to the standalone intra-session RNN. The best results per dataset are highlighted. 48
6.2 Recall and MRR scores for the Bayesian Personalized Ranking Matrix Factorization (BPR-MF) baseline and the RNN models on the hold-one-out version of the dataset. Only the best performing II-RNN model is included for each dataset. Relative scores are given compared to the standalone intra-session RNN. The best results per dataset are highlighted. 49
6.3 First-n-recommendations results on the Reddit dataset. The values for II-RNN are for the last hidden state implementation. . . . 50
6.4 First-n-recommendations results on the Last.fm dataset. The values for II-RNN are for the average-pooling implementation. 51
6.5 First-n-recommendations results on the Instacart dataset. The values for II-RNN are for the last hidden state implementation. . . 52

Glossary

BPR Bayesian Personalized Ranking. 38

BPR-MF Bayesian Personalized Ranking Matrix Factorization. xiii, 8, 9, 27, 44, 47, 49, 57, 58

FFNN Feed-forward Neural Network. 13, 25

GRU gated recurrent unit. xi, 18, 20–22, 27, 28, 32, 33, 37–39, 45, 47, 57, 63

II-RNN inter-/intra-session RNN. xi–xiii, 36–40, 43, 45, 47, 49–63

Item-kNN Item-k nearest neighbors. 8, 10, 12, 27, 44, 47, 56

LSTM long short-term memory. xi, 18–20, 22, 30, 62

MRR mean reciprocal rank. 24, 37

NBRP next basket recommendation problem. 24, 25

NN Neural Network. xi, 8, 12, 13, 15, 16, 19, 21, 22, 25, 30, 32, 62

ReLU rectified linear unit. 14, 20

RNN Recurrent Neural Network. xi–xiii, 1–4, 11–13, 16–18, 20, 23, 25, 27–33, 35–40, 42, 43, 45, 47–49, 53–58, 60–62

1 Introduction

Here, we introduce the domain of our work and what we want to achieve, we describe our contributions, and the reminder of this thesis is presented.

1.1 Background and Motivation

There is a vast amount of information and products available on the web. Even within a single website the number of items can be overwhelming for users. This is true for news sites, streaming services, e-commerce sites, and many other sites on the Internet. Recommender systems can help create a better user experience by helping users find what they are looking for and are interested in. They can also help businesses in different ways, like showing targeted ads, help to offer a better product, and increase user engagement.

Users are generally interested in finding what they are looking for as easy as possible, or to be shown products or content that interests them, but which they normally would not have discovered on their own. E.g. Spotify helps users discovering new music, tailored to the user, through their Discover Weekly [1] playlists. When a user buys something on the e-commerce site Amazon, the site display items that other users bought together with the chosen item.

In a session-based setting, the actions of the user within a session are correlated. This means that a recommender system can observe the user's actions and improve the recommendations as the system learns more about the user's interests. Recently, RNNs has been shown to work well in the session-based setting [1–4]. RNNs are naturally good at working with sequences of data, because they have an internal memory of what they have already seen, and the ability to update and discard information in their memory. Therefore, a RNN will make more accurate recommendations as it learns more about a user. This also means that a simple RNN will struggle to make good recommendations at the start of a session. The advantage a RNN has over many other recommendation/prediction models, is that it naturally considers the order of sequences. Many other models use the relaxed assumption that the order does not matter, or that items are

[1] "Introducing Discover Weekly: your ultimate personalised playlist": https://press.spotify.com/it/2015/07/20/introducing-discover-weekly-your-ultimate-personalised-playlist/

only dependent on the last previous item in the sequence. Solutions that take the whole sequence into account are possible, but RNNs considers the order of sequences in a very natural way that few other models do.

One of the first papers that looked at how a RNN can be used for session-based recommendations, was [1]. Here a straightforward implementation of a session-based RNN recommender system is made. The RNN processes the sequence of items interacted with, and produce a list of recommended items. Items are represented as one-hot vectors at the input stage. The proposed RNN model achieved marked improvements over widely used approaches, on two datasets. The datasets contained sessions of item interactions in the e-commerce and movie domains. Later, [5] looked at how the model from [1] can be improved through data augmentation, pre-training, privileged information, and output embeddings. These techniques are explained in Chapter 3. All the suggested techniques gave improvements over the original RNN model from [1]. In [4] they investigate how contextual information can be used to improve a session-based RNN recommender system. That is, a RNN similar to the one proposed in [1], is improved by considering the temporal difference between events in a session, and the input context of those events. The input context is the external situation, such as the time of day or weather, of a event. In [6], it is shown how information about items can be used to improve a session-based RNN recommendation system. They used existing methods to create feature vectors from images or text related to items. Multiple modified variants of the architecture from [1], that are able to process the additional feature vector, are proposed. The models exploiting the additional feature vector, are shown to outperform the original model.

The aforementioned papers only look at recommendations on a per session basis. That is, given a user with an interaction history consisting of multiple sessions, using information from past sessions to improve recommendations in the next session could be possible. This possibility has not yet been thoroughly researched. Intuitively, there should be a strong dependence between sessions. E.g. for a news site, a user will probably be interested in reading news within the same news categories that he read in previous sessions. Or for an e-commerce site, if a user purchased hiking boots in his last session, he will probably not be interested in another pair in his next session. However, he might be interested in additional hiking equipment such as a primus stove.

In this thesis, we want to investigate how the straightforward implementation of a session-based RNN recommender system can be extended to also make use of previous user sessions, thereby improving recommendations within a session. We will add a second RNN layer to the model in order to process previous sessions, use this to supply the original RNN layer with information about the current session, and thus be able to improve recommendations within the current session.

1.2 Goals and Research Questions

We want to find out how previous sessions can be incorporated into a RNN-based session-based recommender system. The goal of this thesis is therefore to implement a novel RNN-based session-based recommender system, that also considers the user history of users. We want to find out how the model performs as a recommender system.

Intuitively, RNNs improve their sequence predictions throughout a sequence, after having learned from the earlier items. Specifically, it seems reasonable that a RNN is limited to produce sub-optimal predictions at the start of sequences. Since our proposed model supply the RNN with initial information about the session, it should be able to make stronger recommendations from the start of the session.

Research question 1: Can a RNN be used to learn from previous sessions, and thus produce initial information about the next session, which can help improve recommendations from the straightforward session-based RNN recommendation system implementation?

Research question 2: Given that our proposed model performs close to the straightforward RNN model or better, is it able produce improved recommendations at the beginning of sessions, and thereby deal with the cold start problem of session-based recommendations?

1.3 Research Method

After building our model, we will compare it to multiple baselines on different datasets. One of the baselines should be a RNN. First, we know from earlier work [1–4], that it performs very well in the session-based recommendations scenario. Second, our model is an extension to a standard RNN, which means that our model should be able to outperform a standard RNN.

We also need to tweak our model, by adjusting parameters and testing different implementations, to perform as optimal as possible. Then we theorize about its performance, trying to find its strengths and weaknesses, and why it performs the way it does. Finally, we test our hypotheses with experiments on the model.

If time allows it, we perform more iterations of tweaking, theorizing, and experiments, where the tweaking is done based on what we discover in the experiments.

To answer our second research question, we compare our model with the straightforward model on the first recommendations within sessions as well as the overall recommendations.

1.3.1 Limitations

As touched upon earlier, there are at least two kinds of recommendations. One type is novel recommendations. These help the user discover new content that he likes. It is a way of helping the user explore, by guiding him to content that he will find interesting, but maybe would not have found by himself. An example of this is Netflix and Spotify who try to expose the user to new movies/series and music, respectively. The goal of these kind of recommendations is often to get the user to consume more content than he normally would, but also to help the user discover content. Another type is predictive recommendations. These help the users by showing them what they are looking for. The goal is often to provide an improved user experience, by saving the user from the work of finding the desired content. An example of this could be an e-commerce site that helps users find what they are looking for, saving them some trouble of searching and browsing.

It is hard to evaluate the recommendations of a system that produces novel recommendations. This is because it would require actual users to evaluate the recommendations, either directly or indirectly. Predictive recommendations, on the other hand, can be evaluated without human interaction, if the system has access to a dataset. The true actions of the user are in the dataset, so the recommendation problem becomes a sequence prediction problem.

Because of this, the work in this thesis focus on creating a recommender system that tries to predict the actions of users.

1.4 Contributions

We propose an extension to the straightforward session-based RNN recommender system, that makes it able to learn from the sequence of recent sessions leading up to the current session. The model is able to predict a user's current interests based on representations of his recent sessions. We show two simple methods for creating session representations.

Our results on three different datasets shows that our proposed model significantly outperforms the original RNN model, and that both session representation methods used, work well. The proposed model achieves stronger overall recommendations. The biggest improvement is observed at the start of sessions, and our model is therefore able to deal with the cold start problem of sessions. This is important, since the original model depends more on observing some events within the current session before it can produce good recommendations.

Chapter	Description
Chapter 1	**Introduction** Gives an introduction, and short overview of the thesis. Introduces and motivates the subject.
Chapter 2	**Background theory** Explains central theory used in this thesis.
Chapter 3	**Related work** In this chapter we investigate related work by others. This include work that has inspired this thesis, and knowledge that this work is based on.
Chapter 4	**Architecture** Explains our proposed model in detail.
Chapter 5	**Experiments and results** Describes the experiments we have performed, and the results from these. This includes information about the datasets used.
Chapter 6	**Evaluation and conclusion** This chapter evaluates and discuss our results. Our contributions are presented here, and further work is outlined.
Appendix	**Appendix** Contains additional information

Table 1.1: Outline of this thesis.

1.5 Thesis Structure

The reminder of this thesis is outlined in Table 1.1

2 Background Theory

In this chapter, we explain terms and concepts used in this thesis.

2.1 Recommender Systems

Recommender systems try to predict a user's evaluation of an item, or what items a user is interested in interacting with. They can be, and are, used in many areas. Some examples are music, movies, news, restaurants, recipes, online shopping, and dating. There are two basic approaches to recommender systems; collaborative filtering and content-based filtering [7]. These two methods can be combined into a hybrid approach.

2.1.1 Collaborative Filtering

Collaborative filtering uses information about user preferences to recommend items highly rated by similar users. I.e. this approach uses the user-item interactions to perform prediction [7]. To illustrate, let us look at a movie recommendation system. A user logs into a website where he can rate movies he has seen, and the site then recommend movies to the user, based on his ratings. With the collaborative filtering approach, the user needs to rate some movies to get good recommendations, and as the user rate more movies, the system can make better recommendations. To make the recommendations, the system groups together similar users, based on their item interactions. A user is then recommended movies that he has not seen and that was rated highly by other similar users. The system decides whether users are similar by looking at how like-minded they are, that is, how similarly they rate movies. This is illustrated in Table 2.1. The system predicts what rating the question mark will be by using the ratings for that item made by similar users. This approach is called *user-based collaborative filtering* [7].

An alternative approach is the *item-based collaborative filtering*. Here, a user-item rating is predicted by identifying similar items instead of identifying similar users. For a given user A, and an item B, where we want to predict A's rating of B, a set of items similar to B is first found. Then the rating of B is predicted based A's ratings of the other items in this set. E.g. if A generally rated other

	Item1	Item2	Item3	Item4
User1	0.8	0.4		0.1
User2	0.9		0.3	0.1
User3	0.4	0.7	0.5	0.9
User4	0.9		?	0.2
User5	0.2		0.6	
User6	1.0		0.5	0.2

Table 2.1: Illustration of collaborative filtering. User4's rating of Item3 is predicted by using the ratings of Item3 (yellow) from similar users (green).

items in the set highly, the system predict that A will give a high rating to B as well.

User-based and *item-based collaborative filtering* are both memory-based methods. They are simple to implement, but they do not work well when the user-item rating matrix is sparse [7].

An alternative to memory-based methods, that is also commonly used, is model-based methods. These methods include machine learning and data mining methods, such as decision trees, nearest neighbor methods, Bayesian methods, NNs and latent factor models (e.g. matrix factorization) [7]. Compared to memory-based methods, these methods are usually more complex, but better able to deal with sparse user-item ratings matrices.

Next, we explain BPR-MF and Item-k nearest neighbors (Item-kNN), which we use in our experiments.

Matrix Factorization

Matrix factorization is a of latent factor model. The idea behind latent factor models, is that users and items can be described by latent factor. E.g. a user might have a preference for a certain actor, or he might like movies with a lot of action and explosions. In fact, this is an important assumption for recommender systems in general. If each user and item is fully unique and have nothing in common with other users or items, then there is no point in having a recommender system. Because each user would not be interested in the items rated by other users [8]. So, we assume that users and items can be described by latent features, and that they might have such features in common. With this assumption, many of the rows and columns in the user-item ratings matrix are highly correlated. Thus, the data contain redundancies, and it should therefore be possible to approximate the data with a low-rank matrix [7].

This brings us to matrix factorization. Given that we have the $m \times n$ user-item

ratings matrix R where all values are observed, the matrix can be expressed as

$$R = UV^T$$

Here U is a $m \times k$ matrix, and V is a $n \times k$ matrix. k is the number of latent factors needed to represent the data, each column in U is the latent vector for a user, and similarly, each column in V is the latent vector for an item. So R can be constructed if we know U and V. The problem is that we do not have access to all values in R, else we would not need a recommender system, and we do not know the value of k. Fortunately, we can choose a value for k and often get a good approximation of R even when the chosen value for k is lower than the actual value [7]. So, a reasonable choice for k is chosen and we have

$$R \approx UV^T$$

The error between the actual R and our approximation can then be expressed as $||R - UV^T||^2$. Each row $\boldsymbol{u}_i$ in U is the latent factor vector for user i, and each entry in this vector describes the user's affinity towards the corresponding latent factor. Similarly, each row $\boldsymbol{v}_j$ of V describes the latent factors of item j. With this, we can predict how user i will rate item j, by

$$r_{ij} \approx \boldsymbol{u}_i \cdot \boldsymbol{v}_j$$

To calculate the approximation UV^T, U and V can be initialized with random values. Then the error can be iteratively reduced until it goes below a specified threshold (or for a maximum number of iterations), using a gradient decent method [7, 8]

Bayesian Personalized Ranking Matrix Factorization

BPR-MF was introduced by [9]. For the matrix factorization method described above, we assumed that some user-item ratings were available. In some systems, a typical example is movie streaming sites, explicit ratings are available. I.e. we have access to ratings telling whether a user liked or disliked an item. However, in many cases only implicit behavior is available. E.g. we know which movies a user has watched, but we do not have user-item ratings for those movies. Therefore we might assume that the user liked the movie because he watched it. But this leaves us with only positive ratings. Missing ratings could mean that the user has not had the chance to consider the item, or that he did not find it interesting and chose to not interact with it. More formally, we have a set S of implicit feedback, where $S \subseteq R$. And the task is to provide each user with a personalized ranking, $>_u$, of items I. Hence the task is to find $>_u \subset I^2$, given the implicit feedback S. Thus, the ranking is a pairwise ranking between items. In [9], they

train the matrix factorization model by fitting it to rank $i_o >_u i_n$, where i_o are the items from S that user u has interacted with and i_n are items u has not interacted with. $i_1 >_u i_2$ means that u ranks item i_1 above i_2. The goal is that the system becomes able to rank unobserved items for each user, after the training is complete.

In [9] they show how this training can be done by optimizing the posterior probability

$$p(\Theta| >_u) \propto p(>_u |\Theta)p(\Theta)$$

Where Θ represents the parameter vector of an arbitrary model class (e.g. matrix factorization), and $>_u$ is the desired but latent preference structure for user u.

Item k Nearest Neighbors

K nearest neighbors methods are some of the most popular collaborative filtering methods used [10]. They are fairly simple to implement, but often very effective. Both user-based and item-based approaches are possible. In this thesis, we focus on the item-based approach, referred to as Item-kNN. The main idea behind the algorithm is to form top-K recommendations based on similarity of items. The motivation behind this idea is the assumption that users are more likely to interact with items that are similar to those he has interacted with in the past. A general description of the kNN method can be found in [10]. Our implementation is based on the implementation from [1], since the traditional implementations does not fit directly in the session-based setting.

The Item-kNN model is usually built based on the $m \times n$ user-item interactions matrix R. Since we deal with sessions, we build it based on the item interactions in the user sessions. Formally, our training data consists of a set of sessions $S = \{S_1, S_2, ..., S_n\}$. Each session consists of a set of items that the user interacted with, $S_s = \{v_1^s, v_2^s, ..., v_p^s\}$. Here V is the set of items, $v \in V$, and $|V| = n$. Traditionally, a $n \times n$ item similarity matrix is calculated based on R. A $n \times n = N$ similarity matrix is also created in our session-based scenario, but the similarity is calculated based on the co-occurrences of items in sessions. So $N_{i,j} = sim(v_i, v_j)$, where

$$sim(v_i, v_j) = \frac{coc(v_i, v_j)}{oc(v_i)oc(v_j)}$$

Here $coc(v_i, v_j)$ is the number of co-occurrences of item v_i and v_j in sessions, for $i \neq j$, and $oc(v_i)$ is the number of sessions where v_i is present. That is, the similarity of two items is the number of co-occurrences of the two items in sessions divided by the square root of the product of the number of sessions in which the individual items occur [1]. This is the cosine similarity between the vectors of the sessions the items appear in.

With this, a top-k recommendation can be made after the user has interacted with item v_i. This is done by recommending the items corresponding to the k highest similarity scores in the ith row of N.

For cases where the set of items is large, N will be huge, which can cause a memory problem. Since N also can end up being sparse, we solve this by not actually creating the whole matrix, and only calculate similarity for items that co-occur.

2.1.2 Content-based Filtering

In content-based recommender systems, descriptive attributes of items are used to make recommendations [7].

Content-based approaches builds a user profile based on the item interactions of the user, and features of items. For example, if a user watches a lot of western movies or movies with a certain actor, the system can infer that the user likes the western genre or is a fan of that actor. Then, the system can recommend movies within the western genre or movies where the actor appears. So, the approach is to build a user profile based on the latent features of the items he interacts with.

To achieve good results with content-based filtering, it is important to create accurate item profiles with representative features. An advantage of the content-based approach is the ability to make recommendations for new items. Even though no ratings for the new item are available, recommendations can be made based on ratings of similar items. However, content-based methods can tend to provide obvious recommendations, and is not able to recommend items with keywords that the user has never interacted with. On the other hand, collaborative-methods are better at recommending such novel items to the user, because they can leverage the knowledge from similar users. Even though content-based methods are effective with new items, they are not effective for new users. The new user needs to perform several interactions or ratings before the system can give him strong recommendations. All in all, content-based and collaborative-based methods have different trade-offs [7].

2.1.3 Cold Start Problem

Recommender systems are dependent on rich data of user interactions in order to provide robust recommendations. Collaborative models are usually more dependent on user data than content-based models [7]. However, item content is not always available to the content-based methods either. In general the cold start problem occurs when the recommender system is based on having information that might not be available in a sufficient amount. For the session-based RNN recommender proposed in [1], the cold start problem is present at the beginning of each session, since the model has no initial information about the user at the

start of a session. Even if that model had initial information about the user, the user's interests might vary from session to session. Thus, the system could still be dependent on observing some user interactions before being capable of providing precise predictions.

Collaborative filtering methods have the cold start problem for both new users and new items, while content-based methods have the cold start problem only for new users [7].

2.1.4 Session-based Recommender Systems

Users often interact with systems in sessions. I.e. they interact heavily with the system for a limited amount of time, and then become inactive for some time, before interacting with the system again. A session refers to the, shorter period of active interaction, where the user performs multiple actions. Examples of user interaction in a session-based manner include listening to music on Spotify, browsing an e-commerce site like Amazon, and participating in an online discussion forum like Reddit.

Many existing recommender systems only considers the last item clicked, such as Item-kNN. Other approaches might take the full user history into account, such as matrix factorization models. In many cases, such as for small e-commerce sites, sessions are treated as independent sessions, even though the sessions are tracked with a user id. The reason for this is often that most users only visit the site a few times. Also, even though a user visits the site more than once, he is often interested in something very different from last time. This lack of user profile makes factor models hard to apply. Thus, neighborhood model are mostly used [1]. However, recent work has shown that RNN-based models can outperform existing session-based recommender system approaches [1, 2, 4, 5].

2.2 Neural Networks

NN are models inspired by the human brain [11]. They consist of layers of neurons, referred to as nodes or units. The nodes are connected by directed links, with the purpose of transferring values from one node to the next. Each link is associated with a numeric weight. Each node can have multiple links coming in, and multiple links going out. The output of each node, a_j, is calculated as

$$a_j = g(\sum_{i=0}^{n} w_{i,j} a_i) + b_j$$

Here, $w_{i,j}$ are the weight associated with the links coming in to node j from nodes i, and a_i is the output, or activation, from nodes i. b_j is an additional dummy

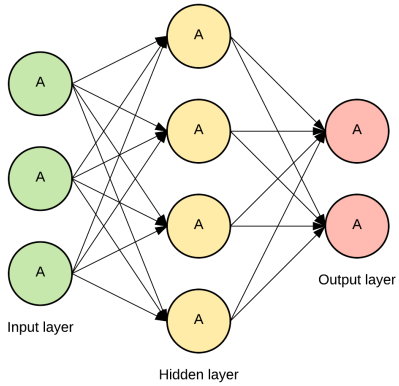

Figure 2.1: A feed-forward network. Examples are inserted into the network by setting the values in the input layer. The values of the nodes in each layer are computed as a function of the values in the nodes in the prior layer.

input associated with each node, called a bias. The bias can be zero, i.e. it is optional. $g()$ is the activation function. By using nonlinear activation functions is important because it makes a neural network able to represent nonlinear function [11]. The nodes in a NN are arranged in layers. A NN has one input layer, one output layer, and any number of hidden layers. Figure 2.1 illustrates a simple NN with one hidden layer. The values of the nodes in the input layer is the input to the model. An example of such input could be an image, where each node gets the value from one pixel. Hidden layers are called hidden because they operate as a black box, where we do not observe their output. There are two distinct ways of connecting the layers in a NN. A Feed-forward Neural Network (FFNN) have connection in one direction only, from the input layer towards the output layer. Here, each node in a layer only receives input from upstream nodes. The output of a Feed-forward Neural Network (FFNN) only depends on the current input. The other approach is called a Recurrent Neural Network (RNN). In a RNN, the output of nodes can be sent back as input into upstream layers, or into its origin layer. Therefore, the output of a RNN will depend on both the current input and previous inputs. This gives RNNs a memory between inputs. We discuss RNNs in more detail in Section 2.3.1.

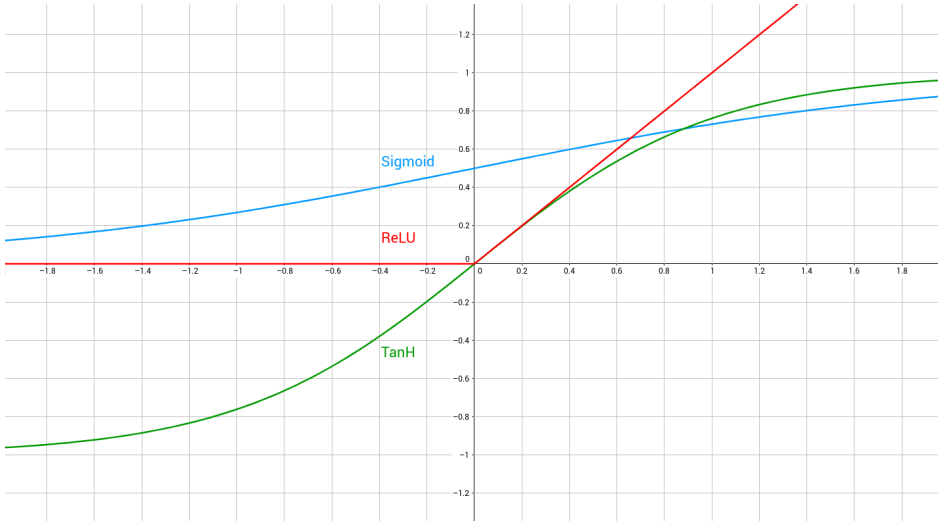

Figure 2.2: Sigmoid, ReLU, and tanh activation functions.

2.2.1 Activation Functions

Multiple activation functions exist. It is seldom obvious which one is the right choice for each case. It is also possible to apply different activation functions to different layers or nodes. Some of the most common activation functions, that often work well, are rectified linear unit (ReLU) [12], sigmoid function, and tanh. These are illustrated in Figure 2.2.

$$\text{Sigmoid: } f(x) = \frac{1}{1+e^{-x}}$$

$$\text{TanH: } f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

$$\text{ReLU: } f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

2.2.2 Training Neural Networks

The most popular training method is backpropagation. Backpropagation uses a chain rule to calculate the derivative of the loss function, with respect to each parameter in the network. The weights are then adjusted by gradient descent [13]. Hence, a loss function is used to compute the error between the actual

output and the desired output of each output node in the network. From this, the weights to upstream nodes can be updated based on the partial derivative of the loss with respect to the weights. In other words, the weights are adjusted by how much they contributed to the loss, in order to minimize the loss.

Other methods of training a NN are also possible, such as evolutionary algorithms. However, the most successful algorithm for training neural networks is the backpropagation algorithm [13], which was introduced by [14].

2.2.3 Dropout

Dropout is a regularization technique that can be used when training neural networks. As the name implies, a subset of the nodes is deactivated. Deactivating subsets of the nodes during training, can help avoid overfitting and it can speed up the training [15]. Speed up is achieved because there are less nodes to train. By randomly deactivating nodes, the network becomes more robust, this is because it cannot rely too much on any one node to produce good results.

One can also view dropout as a way of performing model averaging. By deactivating random nodes, different networks are created and trained. Nodes are dropped with probability $1 - p$ and kept with probability p. During testing, all nodes are kept active.

2.2.4 Minibatches

When training NNs, the error for multiple training examples can be calculated, then the average of the errors can be used to update the weights. These groups of examples are called minibatches. There are several benefits of using this. Using large batches generally results in a more accurate estimate of the actual error of the model. For example, if some of the examples are outliers. With a better estimate of the real error, the training algorithm can use a higher learning rate. Also, some hardware can be better utilized when training with minibatches. [16]

2.3 Deep Learning

The small network shown in Figure 2.1 is not capable of learning very complex tasks, it is too simple. Increasing the size of the layers would probably give it some more capacity. If we wanted to do image labeling, the task of recognizing objects in an image, we could increase the input layer so that it had three nodes for each pixel (RGB). We would also have to increase the output layer to have one node for each possible label. However, even with a huge hidden layer, the model will struggle to make sense of raw pixels with only a few calculations. To help the model, we could give it additional input, we could tell it whether geometric

figures like circles exists in the image. But by doing this, we have started to do some of the image labeling ourselves. Also, the model is dependent on us being able to find good features for it. We want the model to do most of the work for us. It would be nice if the model could extract good features by itself, and it turns out that it can.

By adding more hidden layers to our model it becomes much more capable. In its simplest explanation, deep learning is just that, using more layers. This does not just apply to neural networks, one can also stack layers of other artificial intelligence methods, but we will focus on neural networks. The great benefit of using deep models is that the models can learn to extract useful features themselves [16]. In the case of image labeling, the first layers can learn to extract features like edges and simple shapes. The deeper layers can then use these features to recognize more complex shapes in the image. Or in the case of recommender systems, the model can learn to extract features from the items and user preferences from user actions. Deep learning lets the machine learn hierarchical concepts, giving it more power and flexibility [16].

2.3.1 Recurrent Neural Networks

A RNN is a form of NN that is suited for processing sequences of data. Standard NN have no form of memory between examples, they assume that each example is independent, which often is not true. A RNN solve this by using loops where information from each time step is passed on to the next one. This gives the model memory, and it does not need to assume independence between examples. It also means that the model is more suited for sequences of varying lengths. Figure 2.3 illustrates a basic RNN. It can be illustrated both as a loop and as an unrolled network. Note that, as implied by the looped representation, the RNN is the same across all time steps. In the unrolled version, the RNN boxes are the same network, there are not t different RNNs that are connected.

At each time step, the network takes in external input and input from the last time step, and two outputs are created. One output is passed on to the output layer, or the next hidden layer if there is one. The other output is the state of the RNN, which is passed on to the next time step. Each time step does not have to be separated by a fixed amount of time, each time step is just when input arrives. It is fully possible to apply deep learning to RNNs. One can stack multiple RNN layers, or add other types of layers such as a feedforward layer.

RNNs are not constrained to neither fixed input sizes nor output sizes. The size of input and output *vectors* are fixed, but RNNs are not limited to a fixed number of such vectors. However, RNNs can be applied both to domains where it is natural to treat the data as sequences, as well as problems where the amount of input is fixed. An example of this is to use a RNN as a sliding window over

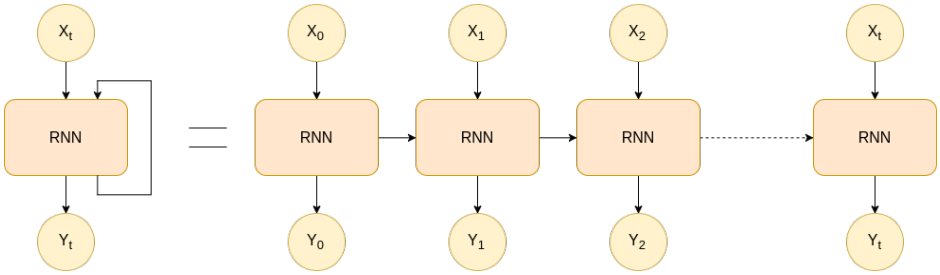

Figure 2.3: A simple RNN. Represented as a loop(left), and unrolled to t timesteps(right).

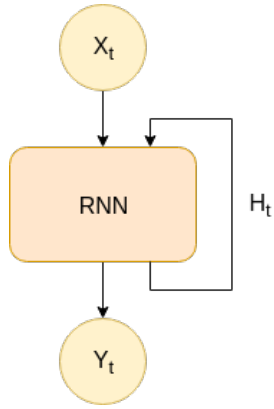

Figure 2.4: Simple illustration of a RNN.

fixed sized images.

We now explain the RNN more formally, using Figure 2.4 to illustrate. The output, Y_t, at each time step in a minimal RNN is calculated as follows.

$$X = X_t | H_{t-1}$$
$$H_t = a(XW_H + \boldsymbol{b_H})$$
$$Y_t = softmax(H_t W + \boldsymbol{b})$$

Where X_t is the input vector, and X is the concatenation of the input vector, X_t, and the hidden state from the last time step, H_{t-1}. W_H and W are weight matrices, while b and b_H are bias vectors. $a()$ is the activation function, and the hyperbolic tangent is a typically choice here. For the first time step, there is no H_{t-1}, so for H_0 an all zero vector is used. The softmax function used when

calculating Y_t is optional. It is useful if you want to interpret the output as probabilities, but if you are only interested in e.g. the index of the highest value, it can be skipped.

LSTM and GRU

Early RNNs had trouble with training because of vanishing and exploding gradients. When using many time steps the gradients often grew too steep, exploded, or they approached zero, vanished. This problem happened because the recurrent edge in a node always had the same weight, which resulted in the derivative of the error either exploding or approaching zero, at an exponential rate, as the number of time steps grew [13]. This was solved by introducing a memory cell. The new model was introduced by [17] and is called LSTM. Improvements to the original model has been made later. In the LSTM model, each node in the recurrent layer is replaced by a memory cell. The internal structure of the memory cell is a bit complex, but simply explained it has an internal state that it can modify, in addition to the old features of RNN nodes. So, the cells can decide how much information in the internal state they want to keep, and how much new information they want to add at each time step.

More recently, in 2014, [18] introduced a new type of hidden units. The cell was based on the LSTM cell, but with a simpler and more computationally efficient architecture. This new recurrent unit is commonly referred to as a GRU. RNNs using either of the two units have been shown to perform well on tasks that require long-term dependencies to be captured [19].

We first explain the LSTM in detail, then the GRU. A detailed illustration of a LSTM is shown in Figure 2.5. The output Y_t, and the hidden state H_t, for each time step is calculated with the following equations.

$$
\begin{aligned}
X &= X_t | H_{t-1} \\
\boldsymbol{f} &= \sigma(XW_f + \boldsymbol{b_f}) \\
\boldsymbol{u} &= \sigma(XW_u + \boldsymbol{b_u}) \\
\boldsymbol{r} &= \sigma(XW_r + \boldsymbol{b_r}) \\
X' &= tanh(XW_c + \boldsymbol{b_c}) \\
C_t &= \boldsymbol{f} \cdot C_{t-1} + \boldsymbol{u} \cdot X' \\
H_t &= \boldsymbol{r} \cdot tanh(C_t) \\
Y_t &= softmax(H_t W + \boldsymbol{b})
\end{aligned}
$$

As in the basic RNN, X is the concatenation of X_t and H_{t-1}. Also, the softmax function in Y_t is optional here as well. σ is the sigmoid activation function, which

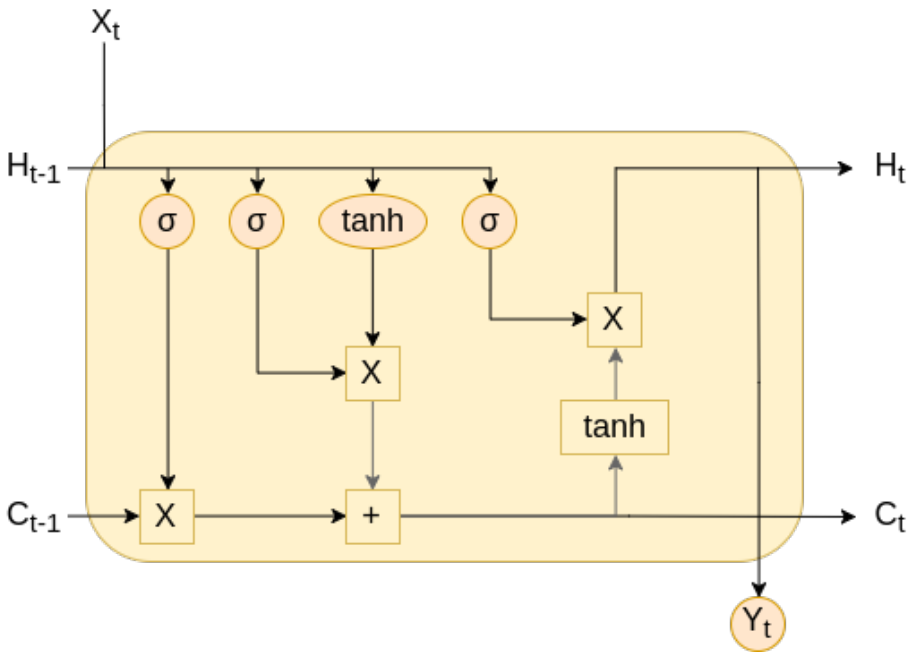

Figure 2.5: Illustration of a LSTM, adapted from [20]. Rectangles represent element-wise operations. Circles and ellipses represent NN layers.

squashes values to the range $[0, 1]$. W and b represent different weight matrices and bias vectors, respectively. All the vectors inside the LSTM cell are the same size. This size is often smaller than the input and output vectors, because a smaller size means less calculations and thus a faster network. However, the size must be large enough for the network to model the data it is trained on, in order to produce accurate results. This size is an adjustable hyperparameter, and it is a trade-off between runtime speed and how complex data the model can represent.

These equations might seem complex at first, so we will explain the core idea behind them. A RNN has a hidden state that makes it able to remember information from earlier inputs in a sequence. The LSTM introduces a second memory state, C. f is intended as a forget gate, which decides what information in the memory that should be forgotten. For example, if we are using a LSTM to give recommendations on an e-commerce site. After a user have looked at some items, the LSTM has learned what items the user is interested in. If the user then buys one of these items, then he will probably not be interested in getting that item as a recommendation. Hence, the forget gate will decide that the LSTM should forget that the user was interested in that item.

u is the update gate in the LSTM cell. This decides which information from the current input should be added to the memory. X' is the current input after it has been through the activation function. Alternatives to tanh, such as the ReLU function, can be used here. The new memory state, C_t, is calculated by applying the forget vector to the last memory state, C_{t-1}, and adding it to the update vector applied to the input. In other words, it is the sum of what should be remembered from the past and what should be remembered from the new input. H_t is calculated by applying the result gate to C_t. Due to the σ function, the f and u vectors only have values between zero and one. This means that C_t will have values between zero and two. To avoid increasing values in H_t for each time step, the values in C_t are squashed to the $[-1, 1]$ range with the tanh function. So C works as the long-term memory, and the result gate is applied to it in order to retrieve information relevant to the current time step. This information is stored in the working memory H.

Note that there are many variations of the LSTM. The one described here is often used. We now look at GRU, a variation of LSTM, that has become popular. GRU is a simpler version of , and therefore less computationally expensive, but often without performing worse than LSTM. The GRU cell is illustrated in Figure 2.6, and the outputs are calculated by the following equations.

$$X = X_t | H_t - 1$$

$$\boldsymbol{z} = \sigma(XW_z + \boldsymbol{b_z})$$

$$\boldsymbol{r} = \sigma(XW_r + \boldsymbol{b_r})$$

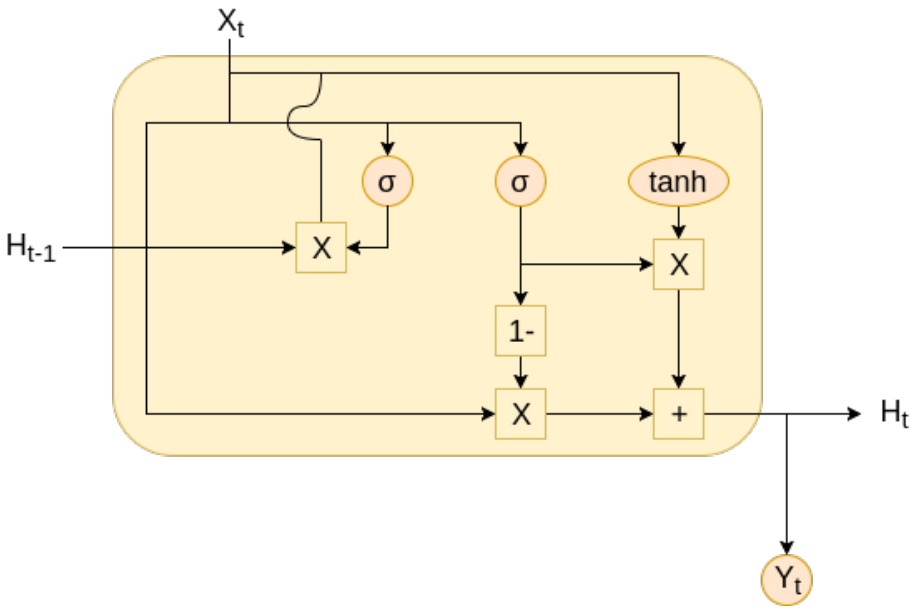

Figure 2.6: Illustration of a GRU, adapted from [21]. Rectangles represent element-wise operations. Circles and ellipses represent NN layers.

$$X' = X_t|\boldsymbol{r} \cdot H_{t-1}$$
$$X'' = tanh(X'W_c + \boldsymbol{b_c})$$
$$H_t = (1 - \boldsymbol{z}) \cdot H_{t-1} + \boldsymbol{z} \cdot X''$$
$$Y_t = softmax(H_tW + \boldsymbol{b})$$

The difference from the LSTM architecture is that in GRU the forget and update functionality is combined into one function, z. Also, only one memory state is used. As before, softmax is optional and other choices of activation functions than tanh are possible.

2.3.2 Item Representations

Here we explain two different ways of representing items as input into a NN.

One-hot Vectors

A one-hot vector is simply a vector where all the values except for one, are the same default value. Usually the default value is zero, and the non-default value is 1.

This can be useful when we want to feed a neural network with classes that can be enumerated, but where the order of the numbers has no more meaning than as an identification number. That is, the numbers do not describe the classes, and the assignment of numbers is indifferent. Furthermore, if we wanted the network to output one of the items, then we could let the network output a vector with the same size as the number of items. The value at each index can then be interpreted as how certain the network is that the corresponding item, is the correct output.

Embeddings

A one-hot vector is very sparse, which means that there are a lot of "empty" values. Thus, the vector takes up much space without containing much information. An embedding is a mapping to a lower dimensional space, a smaller vector. A simple example of this is numbers in binary format. With one-hot encoding, we can represent 10 numbers with 10 bits, but by using all possible combinations of ones and zeros, it is possible to represent 2^{10} numbers with 10 bits. Similarly, sparse vectors can be mapped to denser, lower dimensional vectors. Note that in principle, this can be done as a one-to-one mapping, where no information is lost. When used in practice with neural networks, the embedding matrix, which maps a sparse vector to a dense embedded vector through matrix multiplication, is initialized with random values. Part of the training of the network is then to

also train the embedding matrix. The embedding matrix is just another layer in the network, and can be trained like any other layer.

Going from a one-hot vector to an embedded vector can help speed up calculations, because multiplying a matrix with a one-hot vector is equivalent to extracting the row in the matrix that corresponds to the hot value. Furthermore, by training the network to learn to embed, it can learn to map vectors that represent similar things to similar vectors. This is used in Word2Vec [22] to map words with similar meanings to similar vectors in vector space.

The advantage of using embeddings to represent input items, instead of a one-hot representation, is that the embeddings can be trained together with the rest of the network. This means that the different indexes in the embedding vectors can represent latent features in the item that the embedding represent. Since the embeddings can be trained, they can be initialized with random values and the network can discover the latent features through training.

2.4 Evaluating Recommender Systems

Here we describe the two evaluation metrics used by [1], and many of the papers using RNN-based session-based recommendation models discussed in Chapter 3.

2.4.1 Recall

We assume that the recommender system can make a recommendation in form of a ranked list of N items. Thus, the item at the top of the list is the one that the recommender system believes is the most relevant for the user, and so on.

In some cases, the user is presented with the full list of recommendations, thus the ordering does not matter. What matters then is that the relevant item is present in the list. Recall is a measure of how often the list of recommendations contains the relevant item. In our case the relevant item is the next item that the user will click on, since we are focused on predictions.

Depending on the system, different number of recommendation can be made. But in most cases only a small number of recommendations, about 5-10, can be made, because the users don't have patience to look through a long list of recommendation. Furthermore, a good recommender should not need many attempts to predict the relevant item.

Recall can be evaluated for different values of N. The recall score is calculated as

$$\text{Recall@N} = \frac{|\{\text{relevant recommendations}\}|}{|\{\text{relevant items}\}|}$$

And in our case, we define a 'relevant recommendation' to be a list of item recommendations that contains the relevant item.

2.4.2 MRR

A recommender that has the relevant item in the end of its list of recommendations will get the same recall score as a recommender that has the relevant item at the start of its list. It is preferable that a recommender ranks the relevant item as high as possible compared to other items. E.g. impatient users will only look at the first of many recommendations. mean reciprocal rank (MRR) is a score that tells us something about how high in the list of recommendations the relevant item is. MRR is the average of the reciprocal ranks of the relevant items in the recommendation lists given as response to the recommendation queries Q:[23]

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i}.$$

So, it is the average of the inverse rank of the relevant item in each list of recommended items from the recommender system. The rank is 1 if the relevant item is first in the list, 2 if it is second etc. If the relevant item is not in the list the rank is set to infinite. I.e. 0 is used instead of $\frac{1}{\text{rank}_i}$.

If one recommender system rarely predicts the relevant item, but is able to rank it highly when it does, it can get a similar score as a recommender that often predicts the relevant item but always gives it a low rank. But in this case the first recommender would get a low recall score, while the second would get a higher one. Therefore, it is useful to use both recall and MRR to evaluate recommender systems. Also, the two scores are intuitive and can easily be interpreted.

2.5 Next Basket Recommendation

The next basket recommendation problem (NBRP) is different but related to the session-based recommendations that we deal with in this paper. We mention NBRP because it faces one of the same challenges that our proposed model, and therefore we discuss some papers on NBRP in Section 2.5.1.

Given a user's purchase history, the NBRP is to recommend the next basket of items that the user would be interested in buying. The purchase history contains previous bought baskets, and each basket contain a set of items. A typical example is shopping on e-commerce sites like Amazon. This is similar to session-based recommendations, since baskets are similar to sessions. Each basket could be associated with a session, where the session can consist of all user actions and items interacted with, while the basket only consists of the bought items. The items in the baskets could be temporally ordered. Items in a user's basket are often related to other items in the same basket or in previous baskets. E.g.

a user may buy a laptop and some accessories in the same basket. It may also buy only the laptop accessories in one basket, because it bought a laptop in a previous basket.

2.5.1 Representing Past User Sessions in Next Basket Recommendation

The idea behind the model we propose in this thesis is to improve upon the session-based RNN models proposed by [1] and others mentioned in Section 3.1, by utilizing past user sessions. Therefore, we need a way of representing these past sessions.

In [24] and [25] the authors deal with this problem when they look at the NBRP. [25] use a FFNN in their approach, while [24] use a RNN. Both papers use embeddings to represent the items. Also, both papers create vector representations of the previous user baskets, which are used as input into their NN-model. [25] concatenate the vector representations of the previous k user baskets into one vector which becomes the input vector, while [24] inputs the vector representation of one basket at each time step in the RNN. Creating the basket representations from their set of items are solved with pooling.

Two pooling methods are used. The first is average pooling, where each dimension of the basket representation vector is calculated as

$$\boldsymbol{b}_t^u = \frac{1}{|B_t^u|} \sum_{j=1}^{|B_t^u|} \boldsymbol{n}_{t,j}^u. \tag{2.1}$$

Here $\boldsymbol{n}_{t,j}^u$ is the embedded item representation of the j-th item in the basket of user u at time t, B_t^u is the basket of embedded item representations, and $\boldsymbol{b}_t^u$ is the embedded basket representation for user u at time t.

In other words, average pooling is the aggregation of vectors by taking the average value of every dimension of those vectors.

The second pooling method suggested is maximum pooling, which is calculated as

$$b_{t,k}^u = max(n_{t,1,k}^u, n_{t,2,k}^u, ...), \tag{2.2}$$

where $b_{t,k}^u$ is the k-th dimension of a basket representation vector $\boldsymbol{b}_t^u$, and $n_{t,j,k}^u$ is the value of the k-th dimension of the vector representation of the j-th item, $\boldsymbol{n}_{t,j}^u$, in basket B_t^u. Both pooling methods are illustrated with an simple example in Table 2.2

$\boldsymbol{n}^u_{t,1}$	[	0.4	0.3	0.9]
$\boldsymbol{n}^u_{t,2}$	[	0.5	0.7	0.1]
$\boldsymbol{n}^u_{t,3}$	[	0.6	0.2	0.5]
avg-pooling	[	0.5	0.4	0.5]
max-pooling	[	0.6	0.7	0.9]

Table 2.2: Simple example of average pooling and maximum pooling. Values for max-pooling are indicated with blue color.

3 Related work

In this chapter, we first look at work on using RNN as (part of) a recommender system. Then we look at different approaches to improve a RNN recommender, and how user history can be incorporated.

3.1 RNN as a Recommender System

In [1] a RNN with GRU cells is tested as a session based recommender where no user history is available. Their proposed architecture is illustrated in Figure 3.1. In short, they achieved very promising results. This was the paper that first introduced a straightforward implementation of a session-based RNN recommender system. The model was tested on two datasets, both containing sequences of user clicks with timestamps. The first dataset contained clicks on items from an e-commerce site from the RecSys Challenge 2015 [1], and the other contained clicks on videos from a YouTube-like platform. In addition to some trivial baselines, Item-kNN and BPR-MF [9] were used. Item-kNN and BPR-MF are usually strong baselines for recommendation problems and are often used in practice. BPR-MF performed poorly on both datasets, while Item-kNN was the best performing baseline in both cases. The proposed RNN model significantly outperformed all baselines.

An interesting approach that was tested, was to use a weighted sum of the item representations where earlier items were discounted. That is, the input vector at each step was

$$\boldsymbol{i_t} = \alpha \boldsymbol{i_{t-1}} + \boldsymbol{o_t}$$

where $0 < \alpha < 1$, and $\boldsymbol{o_t}$ is the one-hot representation of the item clicked at time step t. The motivation was that encoding information about earlier events into each input could help to reinforce the memory effect in the RNN. However, it turned out that the model always performed better when only receiving the one-hot representation for the current item at each time step.

Another possibility that was experimented with was using multiple layers of GRU cells. This means that multiple GRU cells are stacked and where the hidden state from each cell becomes the input of the next one. This is illustrated

[1] RecSys Challenge 2015: http://2015.recsyschallenge.com/challenge.html

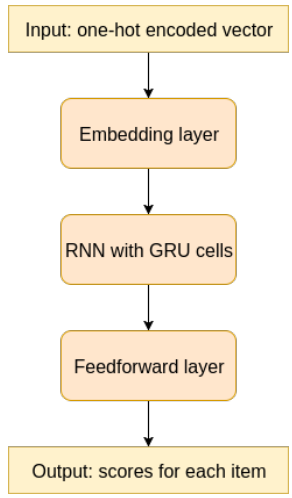

Figure 3.1: Illustration of the proposed model in [1].

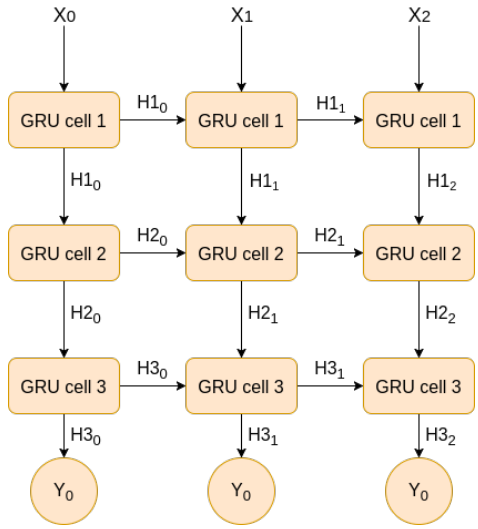

Figure 3.2: RNN using multiple GRU cells, unrolled to three time steps. Any number of sequence lengths is of course possible as with a RNN with one layer of cells.

in Figure 3.2. Also, here the simpler method of using only a single layer gave the best performance. The authors stated that the sequences in their dataset might have been too short for multiple cell layers to contribute. With longer sequences multiple layers could be useful because each layer could learn information at different time scales.

The successful application of RNNs for the session-based recommendation problem, and closely related problems, has been further supported by many other papers [2–6, 24, 26, 27]. In the next sections, we look at some interesting approaches from some of these and other papers.

3.2 Dealing with Large and Dynamic Item Sets

Two problems that real world recommender systems often must deal with are when the set of items are large, and when the item set is dynamic. Sites like Amazon and Spotify deal with millions of items and songs. For both, new items/songs are continually added to the site. Furthermore, for news sites, the articles quickly become old and uninteresting as well. In [5] and [28], the authors investigate approaches that can deal with these problems. This is done by having the recommender system only operate in the latent feature space. This requires methods to encode item- and other input to latent features, and to find corresponding items from embeddings. [28] suggest using nearest neighbors methods to find a set of items close to the recommended embedding in embedded space. Since the recommender system only deals with item embeddings, i.e. latent features, it does not need to know about the actual item set. Thus, it is able to operate even though new items arrive and old items are removed. Furthermore, the embedded space is smaller than full item space, which makes the recommender system able to scale with large item sets. Also, the reduced number of parameters means that the recommender can calculate recommendations faster. However, the need for additional methods to map back to item space might restrict the speed gain somewhat.

3.3 Temporal Dynamics

Temporal changes in both user preferences and item relevance is a complicating factor for recommender systems. Items that were popular half a year ago, might not be as popular today. This implies that a recommender system should be trained on recent data to keep it up to date with the latest consumer trends. However, the amount of recent data can be limited, and old data can contain useful information as well. In [5] shows that a useful approach is to pre-train the recommender system on the full dataset, and then use recent data to fine-tune

the model afterwards.

The authors of [29] argue that both item-features and user-features can evolve over time. As an example, consider an online discussion forum like Reddit. Here, users can create sub-forums, called subreddits, for discussing specific topics. Users with different interests can join the subreddit, participate in discussions, and start their own discussions. The participants obviously affect the discussions. Therefore, the topic of the subreddit can change over time as new users join and adds their personal touch to the discussions. Likewise, a long-time follower of a subreddit may gain new interests based on what is discussed in the subreddit. The authors therefore suggest modelling this co-evolutionary nature. This is done by combining a RNN with multi-variant point process models. The posed problem is interesting, and underlines the need for recommender systems to be able to keep up with the latest trends.

The idea that latent item- and user-features are not static, has also been investigated in [30]. They work on a movie dataset and the task of predicting user-movie ratings. Movies can achieve cult status long after their release, or get a surge of interest after receiving an award such as an Oscar, while some movies are popular during certain seasons such as Christmas. Also, user's preferences change over time. The authors experiment with modeling static user- and item features, and they use two RNNs to model the temporal evolution of users and movie state. That is, a individual RNN for the users and one for the items. The user-state RNN receive a user's movie rating vector and time as input. The movie-state RNN is defined in the same manner.

In [31], the problem of Just-In-Time recommendation is handled with an LSTM. The task is to predict when the user will return to a service, and what he will be interested in a that time. E.g. the user's music preference will probably be different on a Friday night compared to Monday morning. The approach is based on survival analysis, which is the study of the probability that a patient will survive at least until a time T. The approach suggested in this paper operates on an inter-session level, similar to what we use in our own proposed model, discussed in Section 4.3.2.

In [26], the authors look at modeling both the short- and long-term interests of users. They suggest two models. The first one combines two NNs and a RNN. A NN is used to model item static features, one is used to model user static features, and the RNN is used to model user temporal features. The second model, deals with the temporal user features at two levels of granularity. A fast-rate RNN and a slow-rate RNN is used to model the most recent user interests and seasonal user interests, respectively. This second model is referred to as a multi-rate model. Both models outperformed state-of-the-art baselines. Best performance was achieved with the multi-rate model, but this one also has the drawback of having many parameters to train.

3.4 Context Aware Systems

As already mentioned, the time of a user's actions can play an important role in what the user's preferences are. In addition to this, if user actions happen close to each other in temporal space, then the actions are probably more related to each other than if the time between them had been longer. In [4], the authors refers to these two conditions as input context and transition context. Transition context is the temporal distance between user actions. Input context refers to the surrounding situation under which the user action takes place. The input context is not restricted to time, other contexts such as weather pr physical location can also be used. The authors suggest using input context and transition context as additional input to a RNN-based recommender system. They found that utilizing either of these can improve the recommendations, and that the best results can be achieved by combining both types of context.

Many recommender systems only use the items interacted with as input. We have seen suggestions to use temporal information as well. In [2], it is suggested that all user actions can be useful in order to provide stronger recommendations. Search functionality is typically available on modern e-commerce sites. The user's search queries are a rich source of their preferences. In general, all types of observable user behavior have potential of providing the recommender system with information about the user. The paper investigates using item- and event embeddings as input to a RNN-based recommender, where both are latent feature vectors describing the item or the event interacted with. In case of the event, methods such as bag-of words can be used on search queries to create the event encoding.

3.5 Other Relevant Techniques and Approaches

3.5.1 Dealing with Long Sessions

To be able to train a RNN, it must be unrolled. In practice, this means that constraints on computational capacity or time, can result in an upper limit of time steps the RNN can be unrolled to. User sessions can of course be of any length. So how should a session of length s, be handled when the RNN implementation allows a maximum of t time steps, and $s > t$. Note that we here assume that we have access to the full session and that we want to make a recommendation based on it. A simple solution is to cut off the first $s - t$ time steps from the session, and make the recommendation on the last t time steps. This can result in reduced prediction accuracy. Note that, this problem only exist during training since the unrolling is needed for backpropagation of the loss. When testing, only the hidden state from the previous time step is needed so the RNN does not need

to be unrolled. In [27] they use history states to deal with this problem. This is done as follows. Let $i_0, i_1, ..., i_s$ represent the items that a user has interacted with in its session. If $s \geq t$, the history state is computed as:

$$\boldsymbol{n} = \sum_{j=0}^{s-t} \epsilon_j \boldsymbol{n}_j,$$

where $\boldsymbol{n}_j$ is the embedded vector representation of item i_j, and ϵ_j is the aging factor for old states, which give more weight to the newest items. The history state is then given as the first input into the RNN, and the corresponding output is ignored. So, the idea is to balance the trade-off between computation and overhead by supplying the RNN with a summary of the first part of the session. In the paper, usage of a history state improved the model's performance. However, the difference was most notable when the hyperparameters of the model were not fine-tuned. So even though using a history state can give an improvement, [27] found that tuning the model is more important and can have a bigger impact on the performance.

3.5.2 Feeding Rich Input to the Recommender System

Strong results have been achieved by using recommender models that only receives item IDs as input. Intuitively, a recommender could improve on this if it received more information. Several papers have looked at different approaches to this. In [6], the authors look at recommender models that take item IDs and/or item features as input. They compared several architectures, and found that a model that use both types of input can significantly outperform models that use only item IDs or only item features. The strongest model they found, used a GRU layer to process item IDs, and another GRU layer to process item features, separately. The output of these two layers were then combined through another final NN layer to produce output recommendations. This approach requires a method to extract item features. E.g. to extract features from textual description of items, encoding methods such as word2vec [32] can be used. Their parallel GRU model is illustrated in Figure 3.3

In addition to information about the items, information about the user interactions can also be used.

3.5.3 Training Recommender Systems Consisting of Multiple RNN Layers

In [6], the authors also look at how to best train their model that consists of two parallel GRU layers. The straightforward method is to train the whole model in each backpropagation pass, but their results showed that better results could

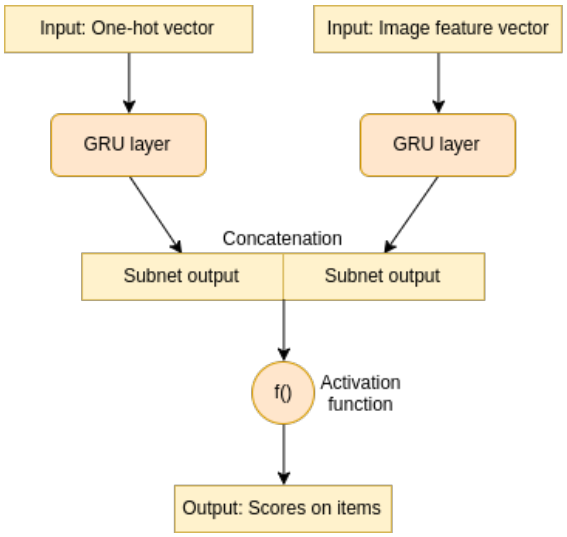

Figure 3.3: Adapted illustration of the parallel GRU model used in [6].

be achieved when only training parts of the network at a time. The reasoning is that alternating training approaches prevents different components in the model from learning the same relations. Two of the approaches that worked well, was to alternate between training sub-networks from mini-batch to mini-batch, and from epoch to epoch. E.g. training the item ID GRU layer for 10 epochs, while the other GRU layer was frozen, and then switching.

3.5.4 Data Augmentation and Privileged Information

The authors of [5] looked at different ways of improving the results achieved with the RNN-based session-based recommender system proposed in [1]. In addition to pre-training the model and outputting embedded recommendations, as discussed earlier, they propose using data augmentation and privileged information. The data augmentation consists of using all prefixes of the original input sessions as new training sequences, and applying dropout to session events. This is illustrated in Figure 3.4.

Privileged information is used by training a teacher model on the remaining parts of sequences, and training the recommender model on a trade-off between the real labels and the labels predicted by the teacher model. Formally, given a sequence $[x_1, x_2, ..., x_r]$ with label x_{r+1} from a session, the privileged sequence is $\boldsymbol{x}^* = [x_n, x_{n-1}, ..., x_{r+2}]$, where n is the length of the original session before

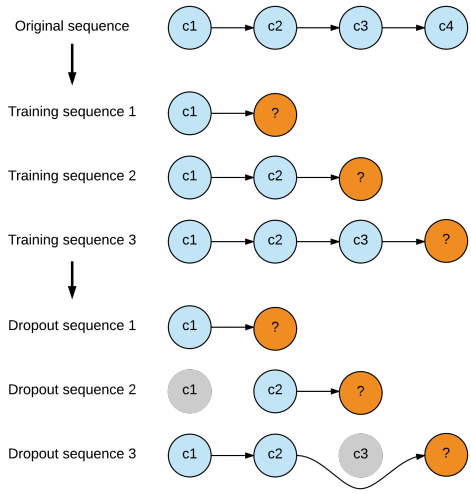

Figure 3.4: Adapted illustration of the data augmentation used in [5].

preprocessing. Thus, the privileged sequence is simply the reversed reminder of the full sequence. The teacher model is trained on the privileged sequences $\boldsymbol{x}^*$, with the same label, x_{r+1}. Then, the student model $M(\boldsymbol{x})$ is trained by minimizing a loss function of the form $(1-\lambda) \cdot L(M(\boldsymbol{x}), V(x_n)) + \lambda \cdot L(M(\boldsymbol{x}), M^*(\boldsymbol{x}^*))$, where $\lambda \in [0, 1]$ is a trade-off parameter between the two sets of labels. The authors suggest that this approach can be useful when the amount of training data available is small.

4 Proposed architecture

In this chapter, we present our proposed model in detail. We first present the motivation and intuitive idea behind our model, then we describe the model itself.

4.1 Main Idea: the II-RNN

In the session-based setting, the user's actions might depend on all earlier actions in the session, not just the previous one. How the dependencies between the actions work, will vary with domains. E.g. on a news site, if a user reads articles about German news and international sports, that user will probably be interested in reading news articles about German sport. While for a online grocery shopping site, past actions might indicate that the user will not be interested in similar items. If the user has added bread and milk to his basket, he will probably not add anymore bread or milk to that basket. But if the user has only added milk to the basket, it might be interested in adding bread as well. RNNs work well in the session-based recommendation scenario because it can process sequences of user actions, and create an internal representation of the user's interests. Also, it does not assume that all actions indicate interest in something, it can learn to interpret actions as sign of disinterest. And as discussed in Section 3.1, the RNN model achieves state-of-the-art performance on session-based recommendation problems.

In addition to the short-term dependencies between actions within a session, there are usually long-term dependencies between actions from different sessions. E.g. a user that was interested in news articles about golf in his previous session(s), will probably also have that interest in his current session. Or a user that bought a new laptop in a recent previous session, will probably not be interested in buying another one in the current session, but he might be interested in accessories to the laptop he bought. This means that it should be possible to improve the recommendations for a session-based recommender system, by giving it information about the user's interaction history. Furthermore, one of the reasons that a RNN works well for recommendations within a session, is that it can process the sequence of the session events. Similarly, we believe that the order of the sequence of earlier sessions can be important. An example could be a person that regularly does his grocery shopping online. If he buys bread in one

session, then he will probably not be interested in buying another one within the next few sessions. And when he has not bought bread in the last sessions, he is probably going to buy one soon.

Since RNNs work well for recommendations on sequences of events within a session, and because the sessions themselves form a sequence, we think that a RNN could work well to process the sequence of sessions as well.

So, our idea is to use one RNN to process the events within a session, as has been done before, and to enhance the recommendations from this by using a second RNN to process a user's recent sessions and help the first RNN with a initial prediction about the current session. In other words, a RNN that works on an *inter*-session level, provides the initial hidden state for a RNN that works on a *intra*-session level. We will refer to this model as II-RNN.

4.2 Problem Formulation

In the session-based recommendation scenario, there is a system with a set of items that a mass of users can interact with. The word item is used in a very broad sense here. We work with three different datasets, where the possible recommendations are sub-forums of a discussion site, artists on a music website, or groceries on a Internet-based grocery service. The datasets are explained in Section 5.1.1. Each user interacts with the system in sessions, as described in Section 2.1.4.

Let N be the set of items in the system, and $\boldsymbol{n}_v \in \mathbb{R}^d$ is the embedded representation of item v. Each user u has an interaction history $S^u = \{S^u_{t_1}, S^u_{t_2}, ...\}$, where $S^u_{t_i}$ is a session of interaction by user u at time t_i. The session history is ordered temporally by t_i. The session length is $|S^u|$. Each session $S^u_{t_i}$ consists of a collection of events $\{\boldsymbol{e}^u_{t_i,j} \in \mathbb{R}^m | j = 1, 2, ..., |S^u_{t_i}|\}$. Where $\boldsymbol{e}^u_{t_i,j}$ is the representation of event j in the session. Events can be any type of interaction, as discussed in Section 3.5.2. However, in this thesis the events will simply be items the user interacts with, so here each event will be an item v. All recommendation models we experiment with use an item id, $i_v \in \{1, 2, ..., |N|\}$, as input for each item. However, the RNN models retrieves the corresponding embedded representation $\boldsymbol{n}_v$ for each i_v, and feed those into the RNN layer of the model.

The common task for all the recommendation models we experiment with, is to predict each consecutive item v in a session $S^u_{t_i}$. That is, for each item $v_{S,j}$ in a session S, predict $v_{S,j+1}$. Thus, items $v_{S,j}$ for $j = 1, 2, ..., |S^u_{t_i}| - 1$ are given as input to the recommender system, and $v_{S,j+1}$ are the relevant items. A recommendation $R_j = \{v_{r_1}, v_{r_2}, ..., v_{r_k}\}$ is made for each item $v_{S,j}$. A perfect recommender would always have $v_{r_1} = v_{S,j+1}$. So R_j is an ordered list of recommended items, where we want the relevant item $v_{S,j+1}$ to appear as early as

possible in the list. The recommender can have k recommended items in each recommendation R_j.

We evaluate each recommender system by their Recall@K and MRR@K scores, for $K = 5, 10, 20$. Therefore, the recommender systems should provide $k = 20$ recommended items in each recommendation R_j. Recall and MRR is explained in Section 2.4.1 and 2.4.2.

4.3 Model Description

II-RNN is an extension to a RNN. Therefore, we start by describing our RNN model, which is similar to the one used in [1], described in Section 3.1. This will be the intra-session RNN in the II-RNN, and will also be used as a baseline to compare the II-RNN to. Afterward we describe the full II-RNN.

4.3.1 Intra-session RNN

The intra-session RNN produce recommendations by processing the sequence of items in a session. Figure 4.1 illustrates the model. This model is very similar to the one in [1] and other papers. We do not use one-hot encodings as input, but use item embeddings directly. Mathematically these two methods are equivalent, but in practice this saves us the computation required to create the one-hot vectors. When the set of items is huge, creating a mini-batch of one-hot vectors will require a large amount of memory, which can be a problem.

The embedded item representation is sent through one or multiple layers of GRU, and dropout is applied to these layers. Afterwards a feedforward layer is used to scale up the vector to $\mathbb{R}^{|N|}$. The output vector is then $\begin{bmatrix} o_{v_1} & o_{v_2} & ... & o_{v_{|N|}} \end{bmatrix}$, where o_{v_i} is a score for item $v_i \in N$. The list of recommendations, R_j is then created by taking items corresponding to the k highest scores, sorted by their score.

Training is done with the Adam algorithm for stochastic gradient descent [33]. And the loss is calculated with cross entropy. The target output is a score of 0 for all items, except for the relevant item which should get a score of 1. This means that we treat the recommendation problem as a classification problem. That is, given the users recent activity, predict the next item he will interact with. This works because the model predicts scores for how likely it believes that each item is the correct class, and these scores then form a natural way of ranking the recommendations.

Our model is very similar to the RNN model introduced in [1]. We skip the one-hot input vectors, and use the embeddings directly. Also, we do not sample on the output. In [1] they have a correct item output for each input, and they sample some items that serve as negative samples, i.e. wrong outputs. When

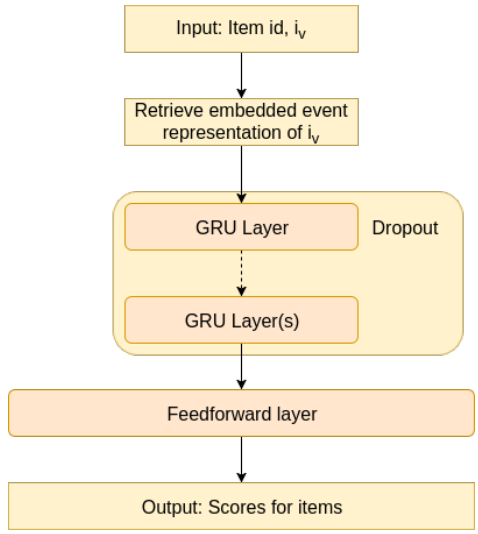

Figure 4.1: Illustration of the intra-session RNN.

we calculate the loss, we set all other items than the target item as negative samples. In other words, we do not use sampling on the output. Furthermore, we only use cross-entropy as our loss function. This means that we train our model as a classifier. However, since the model is asked to produce a ranked list of recommendations, the problem is actually a ranking problem. In [1], the authors found that Bayesian Personalized Ranking (BPR) [9], worked better than cross-entropy for large GRU-layers (~1000 units). For smaller GRU-layers (~100 units) however, the difference between Bayesian Personalized Ranking (BPR) and cross-entropy was minimal on one of two datasets. Since we achieved strong results in our experiments using cross-entropy, we did not experiment further with other loss functions. In addition to this, [1] did many small tweaks and optimizations in order to minimize training and prediction time [34]. This has not been a focus in our work.

4.3.2 II-RNN

Although the intra-session RNN can achieve a strong performance, it starts out in each session without any knowledge about the user. It learns about the user's interests throughout the session, but all that information is discarded again at the end of the session. The II-RNN can therefore improve upon the intra-session RNN because it considers the user's most recent sessions and supplies the intra-

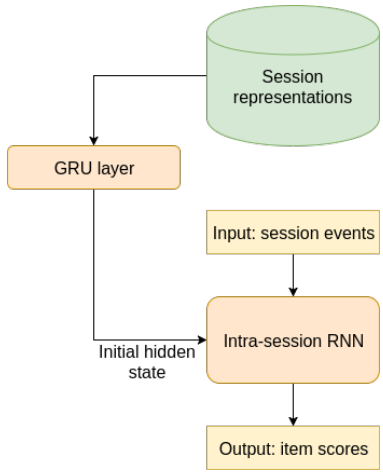

Figure 4.2: Illustration of the II-RNN. This illustration does not show how the session representations are obtained.

session part with information at the start of each session. Figure 4.2 illustrates the II-RNN.

For each session $S^u_{t_i}$ in a user's interaction history S^u, let $\boldsymbol{s}^u_{t_i}$ be an embedded vector representation of that session. The input to the inter-session RNN layer (the GRU layer in Figure 4.2) is then the sequence $\{\boldsymbol{s}^u_{t_{z-g}}, \boldsymbol{s}^u_{t_{z-g+1}}, ..., \boldsymbol{s}^u_{t_z}\}$, where $\boldsymbol{s}^u_{t_z}$ is the representation of the most recent session, and g is the number of recent sessions that should be processed. The initial hidden state, H_0, of the intra-session RNN is then set to final output of the inter-session RNN. In other words, the inter-session RNN produce the initial hidden state of the intra-session RNN, based on a series of vector representations of the most recent sessions for the given user. And the output of the inter-session RNN is calculated before the intra-session RNN starts producing predictions.

We apply two different methods of producing the session representations $\boldsymbol{s}^u_{t_i}$. One is the average of the embedded vector representations of the items in the session, as described in Section 2.5.1, and illustrated in Figure 4.3. The other is to simply use the the last hidden state of the intra-session RNN as the session representation, illustrated in Figure 4.4. Even though the final hidden state can contain more useful information learned by the intra-session RNN, it is more a representation of the end of the session, rather than the whole session. Since the hidden state is produced by a RNN, it will depend on the order of the sequence of items in a session, while the average of the embeddings is unaffected by the order of the items.

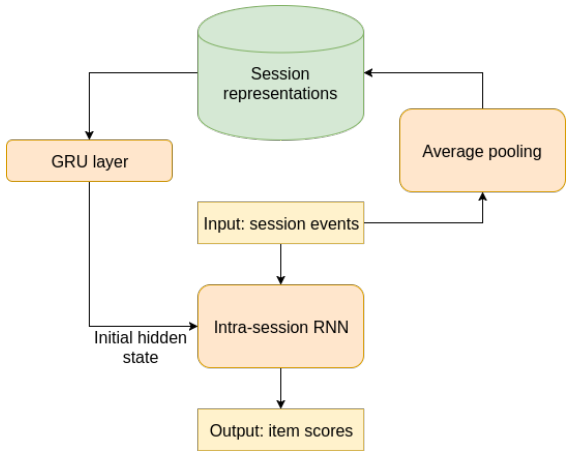

Figure 4.3: Illustration of the II-RNN with average pooling to create session representations from items.

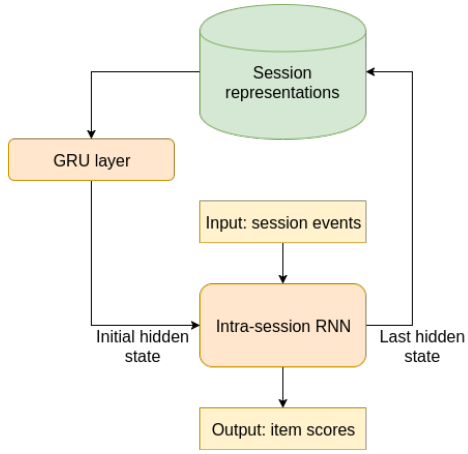

Figure 4.4: Illustration of the II-RNN where the last hidden state of the intra-session RNN is stored as the session representation.

5 Experimental Setup

In this chapter, we explain which experiments we did, how we did them, and we discuss the datasets we used.

5.1 Experimental Setup

5.1.1 Datasets

We experimented with three different datasets. One is a dataset on user activity on the social news aggregation and discussion website Reddit [1] [2]. This dataset contains tuples of usernames, a subreddit where the user made a comment to a thread, and a timestamp for the interaction. The second dataset contains listening habits of users on the music website Last.fm [3] [35]. This dataset contains tuples of user, timestamp, artist, and song listened to. The last dataset is from the Internet-based grocery delivery service Instacart [4] [36]. The Instacart dataset contains logs of users' shopping carts, and the order in which items were added to the cart.

Reddit Dataset

The Reddit dataset contains a log of user interaction on different subreddits (subforums), with timestamps. Here, an interaction is when a user adds a comment to a thread. Since the dataset itself, does not split the interactions into sessions, we did this manually when preprocessing the dataset. To do this we analyzed the dataset and specified a time limit for inactivity. Using the timestamps, we let consecutive actions that happened within the time limit belong to the same session. That is, for a specified time limit l, and a list of a user's interactions $\{a_{t_0}, a_{t_1}, ..., a_{t_n}\}$, ordered by their timestamps t_i, two consecutive interactions a_{t_i} and $a_{t_{i+1}}$ belong to the same session if and only if $t_{i+1} - t_i \leq l$. We set the time limit to 1 hour (3600 seconds).

[1] Reddit: https://www.reddit.com/
[2] Subreddit interactions dataset: https://www.kaggle.com/colemaclean/subreddit-interactions
[3] Last.fm: https://www.last.fm/
[4] Instacart: https://www.instacart.com/

Note that users, in addition to commenting on threads, also do a lot of browsing and reading. Therefore, it makes sense to set a time limit that allows for some time between the interactions captured in the dataset. Also, some users are more active than others, some users are mostly passive consumers who rarely comments. So, it is impossible to set a time limit that fits all users. However, it is important that the time limit is large enough that the average session contains a fair amount of interactions, but small enough so that it is reasonable to assume that the interactions are dependent on each other.

Last.fm Dataset

We also had to split each user's history into sessions manually for the Last.fm dataset. We used the same approach as for the Reddit dataset, but here we used 30 minutes (1800 seconds) as the time limit. Also, we faced the problem that the dataset contains an overwhelming amount of songs. Since our recommendation models produce a score for each possible item, the huge amount of songs caused a memory requirement problem. To solve this, we simplified the dataset by ignoring the specific song of each user interaction and only use the artists. This reduce the item set to a manageable size.

Instacart Dataset

The user actions in the Instacart dataset are naturally sorted into sessions as the dataset is provided. Each user is associated with several previously bought carts, and each cart contain items bought, and the order in which they were placed in the cart.

Preprocessing

After the initial manual splitting into sessions, we used the same preprocessing for all three datasets. In the Reddit and Last.fm datasets, there were many items that repeated consecutively. We are not interested in a recommender system that learns to predict the last seen item, therefore we removed all consecutively repeating items, and only kept one instance. Furthermore, the RNN models need to have a specified maximum length of the sessions, because they must be unrolled in order to be trained. To deal with this, we set the maximum length, L, of a session to $L = 20$. Sessions that had a length l of $L < l < 2L$ were split into two sessions. This was done because we did not want to throw away all sessions that were too long, but splitting very long sessions create many sessions that should not be separate sessions, since the events in them depend on each other. However, there were some unreasonable long sessions that probably originate from bots or some other error source. These were removed with the $2L$ limit for

	Reddit	Last.fm	Instacart
Number of users	18271	977	19420
Number of sessions	1135488	630774	319688
Sessions per user	62.1	645.6	16.5
Average session length	3.0	8.1	9.7
Number of items	27452	94284	41095

Table 5.1: Statistics for our three datasets, after preprocessing.

session lengths. With this scheme, most of the sessions from all datasets were kept.

Sessions of length $l < 2$ were removed, and users with less than 3 sessions were also removed. Finally, the datasets were split into a training set and a test set on a per user basis. For each user, 80% of his sessions were placed in the training set, and the remaining in the test set. Each user's sessions were sorted by the timestamp of the earliest event in the session, and the test set contains the most recent sessions of each user.

Table 5.1 shows statistics for the three datasets after preprocessing (before splitting into training and test sets).

5.1.2 Baselines

In addition to the following baselines, the intra-session RNN itself forms a baseline for the II-RNN.

Most Popular

The most popular baseline is a very simple baseline, but it can perform decently in some cases. All items are sorted by their number of occurrences in the training set, and the top k items are recommended at each time step. Although a very basic baseline, it provides a nice sanity check. Any serious model should be able to beat this model.

Most Recent

Even though we removed consecutive repetitions of items in all sessions, there could still be a high repetitiveness of items within sessions. I.e. some items can occur multiple times in a session. Especially in the Reddit and Last.fm datasets, where users can tend to interact with some subreddits or artists multiple times in their sessions. We believe that it is less likely to see such repetitiveness in the Instacart dataset, because users probably only add each item to their cart once.

The most recent baseline behaves as a stack. It is initially filled with k random items. For each time step, the item interacted with is added to the top of the stack, and the item at the bottom is pushed out of it. However, if the new item is already in the stack, it is just moved to the top. The recommendation at each time step is then the stack of recently seen items, where the top recommendation is the item just interacted with. Our model should be able to beat this baseline significantly. But the most recent baseline gives us information about the diversity of items within sessions.

Item-kNN

Item-kNN is a simple, but usually strong baseline. It is commonly used in practice as a item-to-item recommender [37]. Different implementations are possible. We implemented it as follows. For each item in the dataset, we count the number of co-occurrences with the other items in the dataset. A co-occurrence is when two items appear in the same session. When testing, the algorithm recommends the top k items with highest co-occurrences with the last seen item.

BPR-MF

BPR-MF [9] is a commonly used matrix factorization method. It tries to predict personal pairwise rankings of unseen items. I.e. given a user and two items, BPR-MF tries to predict which of the two items the user would rate higher. We use an existing implementation [5], that we tweak slightly to fit our use case. The original implementation does not recommend already seen items, but in our case, users often interact with items they have already seen. Therefore, we changed it to also recommend seen items.

BPR-MF computes feature vectors for users and items based on the user's earlier interactions, and is then able to make a recommendation based on this. This means that the recommendations will be the same throughout future sessions, unless the model is re-trained. In other words, BPR-MF cannot be applied directly to session-based recommendations. To make a fairer comparison, we created a hole-one out split of the dataset. Only the last session of each user was put in the test set. BPR-MF still produce the same recommendations for all time steps in the test session for a given user.

5.1.3 Implementation

All our code is available on GitHub here [6]. The implementation is done in Python 3.5.2, with the Tensorflow machine learning software library. Instructions to

[5] theano-bpr: https://github.com/bbc/theano-bpr
[6] Implementation: https://github.com/olesls/master_thesis

	Reddit	**Last.fm**	**Instacart**
Embedding size	50	100	80
Learning rate	0.001	0.001	0.001
Dropout rate	0	0.2	0.2
Max. recent session representations	15	15	15
Mini-batch size	100	100	100
Number of GRU layers, intra-session level	1	1	1
Number of GRU layers, inter-session level	1	1	1

Table 5.2: Best found configurations for the RNN models. We found that the configurations that worked well for the II-RNN, worked well for the standalone intra-session RNN as well. Not all configurations are applicable to the standalone intra-session RNN.

recreate our results are available together with the code.

We run our experiments on three different computers, all with the Ubuntu 16.04 operating system. All computers have at least 16 GB of RAM, and a Nvidia GeForce GTX 1060 6 GB or better.

5.1.4 Experiments

We used Recall@K and MRR@K for $K = 5, 10, 20$ to evaluate all models. In addition to the baselines described, we compared the intra-session RNN and the II-RNN, on the described datasets. We experimented with mini-batch sizes, embedding sizes, learning rate, dropout rate, using multiple GRU layers, and number of session representations, to find the best configurations for each dataset. The best configurations we found are summarized in Table 5.2. For the II-RNN we compared results using average-pooling and the last hidden state as session representations for past sessions. We used the same size for the item embeddings and internal vectors in the GRU layers. We found tanh to work well as activation function in the GRU layers, but we did not test other alternatives.

First-n Recommendations

The intra-session RNN learns about the user as it observes item interactions throughout a session. It is therefore reasonable to believe that the model's prediction accuracy increases throughout the session. As discussed in Section 2.1.4, we hope that the II-RNN can improve both the overall recommendations, and especially the first few recommendations in each session. To evaluate this, we test the RNN models both on the overall recommendations and on the first n recommendations, for $n = 1, ..., L$, where L is the maximum session length. That is, we

evaluate the Recall and MRR scores as explained earlier, on recommendations for the first time step, for the first two time steps, and so on. The evaluation score for the first $n = L$ recommendations, is the same as the overall score. To clarify, the first-n score is the score over the recommendations for all time steps up until the nth. As an example, the first-n score for $n = 4$ is the total score for all the four first time steps across all sessions. Thus, for $n = 1$ we only evaluate recommendations at the first time step, while for $n = L$ we evaluate over all time steps.

Creating Mini-batches

As discussed in Section 3.5.4, we want our model to be biased towards recent user trends. This is often desirable in practice, and we find it reasonable to assume that it applies for our datasets. Furthermore, the way we split our dataset into training- and test sets reflect this. I.e. the test set contains the most recent samples for each user. This leaves us with two desirable properties for how the training samples should be processed. First, more recent samples should be processed last. Second, each mini-batch should contain a variety of users. I.e. no user should be over represented with samples in any mini-batch.

To achieve these properties, we constructed the following scheme for creating mini-batches. Each training sample, a session, is associated with a user. All sessions belonging to the same user, are grouped together, and sorted oldest to newest.

6 Results and Discussion

Here, we present our results, then we evaluate and discuss our findings, highlighting interesting results.

6.1 Results

We found that using multiple GRU layers did not improve performance neither when applied at the inter-session level, nor at the intra-session level. Dropout was crucial to get good results on the Last.fm and Instacart datasets, while on the Reddit dataset the models performed best without dropout. To achieve the best results, dropout had to be used on all GRU layers.

Table 6.1 shows an overview of how the models and baselines scored. Relative scores are given compared to the standalone intra-session RNN. We ran the RNN model three times and the results presented in the table are averages of three runs. However, the results were usually consistent between runs. Table 6.2 shows how the BPR-MF baseline performed on the hold-one-out version of the dataset. Item-kNN and most recent baselines were the strongest baselines on the Reddit and Last.fm dataset, but were both clearly outperformed by the intra-session RNN. While on the Instacart dataset, the most recent baseline performed worse than just giving random recommendations. However, the Item-kNN baseline performed closer to the intra-session RNN here. In all cases, the II-RNN significantly outperformed the standalone intra-session RNN and all baselines.

The results from the first-n-recommendation testing is shown in Tables 6.3, 6.4, and 6.5 for the Reddit, Last.fm, and Instacart datasets, respectively. On the Reddit dataset, the II-RNN scores significantly higher on all scores after the first recommendation, compared to the overall score ($n = 19$) of the intra-session RNN. Figures 6.1 and 6.1 shows graphs for the Recall@5 and MRR@5 metrics. Graphs for the other metrics are similar. Figures 6.3 and 6.3, and 6.5 and 6.5, shows equivalent evaluation graphs for the Last.fm and Instacart datasets, respectively. Only the best performing variant of the II-RNN for each dataset is shown.

Dataset: Reddit

Model	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
RNN	0.3372	0.4173	0.5004	0.2436	0.2542	0.2600
II-RNN (last hidden state)	**0.4476 (+32.7%)**	**0.5344 (+28.1%)**	**0.6180 (+23.5%)**	**0.3213 (+31.9%)**	**0.3329 (+31.0%)**	**0.3388 (+30.3%)**
II-RNN (avg.-pooling)	0.4361 (+29.3%)	0.5168 (+23.8%)	0.5963 (+19.2%)	0.3202 (+31.4%)	0.3309 (+30.1%)	0.3364 (+29.4%)
Most popular	0.1322	0.1946	0.2647	0.0850	0.0932	0.0982
Most recent	0.2152	0.2205	0.2209	0.0969	0.0977	0.0977
Item-kNN	0.2171	0.3032	0.3885	0.1174	0.1288	0.1349

Dataset: Last.fm

Model	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
RNN	0.1350	0.1843	0.2478	0.0867	0.0932	0.0976
II-RNN (last hidden state)	0.1439 (+6.6%)	0.2018 (+9.5%)	0.2776 (+12.0%)	0.0891 (+2.8%)	0.0968 (+3.9%)	0.1020 (+4.5%)
II-RNN (avg.-pooling)	**0.1478 (+9.5%)**	**0.2048 (+11.1%)**	**0.2788 (+12.5%)**	**0.0930 (+7.3%)**	**0.1005 (+7.8%)**	**0.1056 (+8.2%)**
Most popular	0.0528	0.0650	0.0829	0.0433	0.0449	0.0462
Most recent	0.1061	0.1305	0.1379	0.0422	0.0456	0.0462
Item-kNN	0.0851	0.1191	0.1590	0.0504	0.0548	0.0576

Dataset: Instacart

Model	Recall@5	Recall@10	Recall@20	MRR@	MRR@	MRR@
RNN	0.0848	0.1248	0.1773	0.0480	0.0533	0.0569
II-RNN (last hidden state)	**0.1050 (+23.8%)**	**0.1541 (+23.5%)**	**0.2179 (+22.9%)**	**0.0603 (+25.6%)**	**0.0668 (+25.3%)**	**0.0712 (+25.1%)**
II-RNN (avg.-pooling)	0.1026 (+21.0%)	0.1515 (+21.4%)	0.2145 (+21.0%)	0.0586 (+22.1%)	0.0651 (+22.1%)	0.0694 (+22.0%)
Most popular	0.0446	0.0692	0.0976	0.0230	0.0262	0.0282
Most recent	0.0000	0.0001	0.0003	0.0000	0.0000	0.0000
Item-kNN	0.0754	0.1114	0.1546	0.0413	0.0460	0.0491

Table 6.1: Recall and MRR scores for the RNN models and the baselines. Relative scores are given compared to the standalone intra-session RNN. The best results per dataset are highlighted.

Dataset: Reddit

Model	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
RNN	0.3660	0.4388	0.5118	0.2781	0.2878	0.2928
II-RNN (last hidden state)	**0.5022 (+37.2%)**	**0.5803 (+32.2%)**	**0.6537 (+27.7%)**	**0.3807 (+36.9%)**	**0.3912 (+35.9%)**	**0.3963 (+35.3%)**
Most popular	0.1296	0.1900	0.2569	0.0883	0.0962	0.1010
Most recent	0.2389	0.2421	0.2425	0.1100	0.1105	0.1105
Item-kNN	0.2463	0.3331	0.4169	0.1403	0.1517	0.1577
BPR-MF	0.1271	0.1900	0.2621	0.0878	0.0961	0.1011

Dataset: Last.fm

Model	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
RNN	0.1568	0.2088	0.2761	0.0972	0.1041	0.1088
II-RNN (avg.-pooling)	**0.1775 (+13.2%)**	**0.2390 (+14.5%)**	**0.3133 (+13.5%)**	**0.1085 (+11.6%)**	**0.1165 (+11.9%)**	**0.1216 (+11.8%)**
Most popular	0.0511	0.0646	0.0801	0.0432	0.0450	0.0460
Most recent	0.0986	0.1170	0.1217	0.0398	0.0423	0.0427
Item-kNN	0.0988	0.1348	0.1746	0.0566	0.0614	0.0642
BPR-MF	0.0619	0.0833	0.1207	0.0467	0.0494	0.0520

Dataset: Instacart

Model	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
RNN	0.0812	0.1189	0.1688	0.0465	0.0514	0.0548
II-RNN (last hidden state)	**0.0974 (+20.0%)**	**0.1423 (+19.7%)**	**0.2010 (+19.1%)**	**0.0562 (+20.9%)**	**0.0621 (+20.8%)**	**0.0661 (+20.6%)**
Most popular	0.0422	0.0661	0.0912	0.0222	0.0253	0.0271
Most recent	0	0.0001	0.0003	0	0	0
Item-kNN	0.0718	0.1046	0.1442	0.0396	0.0439	0.0467
BPR-MF	0.0409	0.0653	0.0923	0.0218	0.0251	0.0269

Table 6.2: Recall and MRR scores for the BPR-MF baseline and the RNN models on the hold-one-out version of the dataset. Only the best performing II-RNN model is included for each dataset. Relative scores are given compared to the standalone intra-session RNN. The best results per dataset are highlighted.

RNN

n	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
1	0.2184	0.3018	0.3934	0.1420	0.1530	0.1594
2	0.2864	0.3639	0.4487	0.2078	0.2180	0.2239
3	0.3042	0.3821	0.4663	0.2199	0.2302	0.236
4	0.3156	0.3935	0.4771	0.2289	0.2392	0.2451
5	0.3215	0.3998	0.4833	0.2331	0.2435	0.2493
6	0.3257	0.4041	0.4876	0.2363	0.2467	0.2525
7	0.3285	0.4070	0.4905	0.2382	0.2486	0.2544
8	0.3305	0.4092	0.4926	0.2397	0.2501	0.2559
9	0.3321	0.4109	0.4842	0.2408	0.2512	0.2570
10	0.3333	0.4122	0.4956	0.2417	0.2521	0.2580
11	0.3343	0.4132	0.4967	0.2423	0.2528	0.2586
12	0.3350	0.4140	0.4974	0.2429	0.2533	0.2592
13	0.3357	0.4147	0.4981	0.2433	0.2538	0.2596
14	0.3362	0.4153	0.4987	0.2438	0.2542	0.2601
15	0.3366	0.4157	0.4991	0.2440	0.2545	0.2603
16	0.3370	0.4161	0.4995	0.2443	0.2548	0.2606
17	0.3373	0.4164	0.4998	0.2445	0.2550	0.2608
18	0.3375	0.4166	0.5001	0.2447	0.2552	0.261
19	0.3377	0.4169	0.5003	0.2448	0.2553	0.2611

II-RNN

n	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
1	0.4130	0.4998	0.5842	0.2893	0.3009	0.3067
2	0.4332	0.5189	0.6018	0.3103	0.3217	0.3275
3	0.4362	0.5225	0.6056	0.3122	0.3237	0.3295
4	0.4396	0.5258	0.6088	0.3152	0.3267	0.3325
5	0.4414	0.5278	0.6108	0.3165	0.3280	0.3338
6	0.4429	0.5294	0.6124	0.3178	0.3294	0.3352
7	0.4439	0.5305	0.6135	0.3186	0.3301	0.3359
8	0.4447	0.5313	0.6144	0.3191	0.3307	0.3365
9	0.4452	0.5319	0.6150	0.3196	0.3312	0.3370
10	0.4457	0.5325	0.6155	0.3200	0.3316	0.3374
11	0.4462	0.5330	0.6161	0.3203	0.3320	0.3378
12	0.4466	0.5335	0.6165	0.3207	0.3323	0.3381
13	0.4469	0.5338	0.6168	0.321	0.3326	0.3384
14	0.4472	0.5341	0.6171	0.3211	0.3328	0.3386
15	0.4474	0.5343	0.6173	0.3213	0.3329	0.3387
16	0.4475	0.5345	0.6174	0.3214	0.3330	0.3388
17	0.4477	0.5346	0.6176	0.3215	0.3332	0.339
18	0.4479	0.5348	0.6178	0.3217	0.3333	0.3391
19	0.4480	0.5350	0.6179	0.3218	0.3334	0.3392

Table 6.3: First-n-recommendations results on the Reddit dataset. The values for II-RNN are for the last hidden state implementation.

RNN

n	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
1	0.0951	0.1273	0.1715	0.0592	0.0634	0.0665
2	0.1169	0.1538	0.2021	0.0766	0.0815	0.0848
3	0.1231	0.1626	0.2141	0.0801	0.0853	0.0889
4	0.1271	0.1684	0.2223	0.0826	0.0881	0.0918
5	0.1289	0.1717	0.2273	0.0836	0.0892	0.0931
6	0.1308	0.1748	0.2314	0.0846	0.0904	0.0943
7	0.1318	0.1766	0.2344	0.0851	0.091	0.0949
8	0.1327	0.1781	0.2368	0.0855	0.0915	0.0955
9	0.1335	0.1795	0.2388	0.0858	0.0919	0.0959
10	0.1339	0.1803	0.2403	0.086	0.0921	0.0962
11	0.1343	0.1811	0.2417	0.0861	0.0923	0.0964
12	0.1347	0.1819	0.243	0.0863	0.0925	0.0967
13	0.135	0.1825	0.2439	0.0863	0.0926	0.0968
14	0.1351	0.183	0.2447	0.0863	0.0927	0.0969
15	0.1353	0.1834	0.2454	0.0863	0.0927	0.0969
16	0.1354	0.1837	0.2461	0.0864	0.0928	0.097
17	0.1355	0.184	0.2466	0.0863	0.0927	0.097
18	0.1356	0.1841	0.2471	0.0864	0.0928	0.0971
19	0.1357	0.1844	0.2475	0.0863	0.0928	0.0971

II-RNN

n	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
1	0.13	0.1812	0.2476	0.0815	0.0882	0.0927
2	0.1421	0.1949	0.2631	0.0908	0.0978	0.1025
3	0.1447	0.1986	0.268	0.0924	0.0995	0.1042
4	0.1465	0.2011	0.2717	0.0934	0.1007	0.1055
5	0.1471	0.2023	0.2734	0.0936	0.1009	0.1058
6	0.1477	0.2032	0.2748	0.094	0.1013	0.1062
7	0.1479	0.2037	0.2757	0.0939	0.1013	0.1063
8	0.1482	0.2042	0.2764	0.094	0.1014	0.1063
9	0.1481	0.2044	0.277	0.0939	0.1013	0.1063
10	0.1482	0.2045	0.2774	0.0938	0.1013	0.1063
11	0.1483	0.2046	0.2777	0.0937	0.1012	0.1062
12	0.1483	0.2048	0.278	0.0937	0.1012	0.1062
13	0.1483	0.2049	0.2783	0.0936	0.1011	0.1062
14	0.1482	0.2049	0.2784	0.0936	0.101	0.1061
15	0.1481	0.2049	0.2785	0.0934	0.1009	0.106
16	0.1481	0.2049	0.2786	0.0933	0.1008	0.1059
17	0.148	0.2048	0.2787	0.0932	0.1007	0.1058
18	0.1479	0.2048	0.2788	0.0931	0.1006	0.1057
19	0.1478	0.2048	0.2788	0.093	0.1005	0.1056

Table 6.4: First-n-recommendations results on the Last.fm dataset. The values for II-RNN are for the average-pooling implementation.

RNN

n	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
1	0.1168	0.1626	0.2139	0.0699	0.0759	0.0795
2	0.1148	0.1607	0.2144	0.068	0.0741	0.0778
3	0.1103	0.1564	0.2113	0.0648	0.0709	0.0747
4	0.1062	0.1519	0.2073	0.0619	0.0679	0.0717
5	0.1023	0.1471	0.2027	0.0592	0.0651	0.0689
6	0.0992	0.1435	0.1989	0.0572	0.0630	0.0668
7	0.0966	0.1403	0.1952	0.0554	0.0612	0.0650
8	0.0944	0.1375	0.1922	0.0541	0.0597	0.0635
9	0.0928	0.1355	0.1898	0.0530	0.0586	0.0624
10	0.0914	0.1337	0.1876	0.0521	0.0577	0.0614
11	0.0899	0.1319	0.1855	0.0512	0.0567	0.0604
12	0.0889	0.1306	0.1840	0.0505	0.056	0.0597
13	0.0880	0.1295	0.1827	0.0500	0.0554	0.0591
14	0.0873	0.1285	0.1815	0.0495	0.0549	0.0586
15	0.0866	0.1276	0.1805	0.0491	0.0545	0.0581
16	0.0861	0.1269	0.1795	0.0488	0.0541	0.0578
17	0.0856	0.1262	0.1787	0.0485	0.0538	0.0574
18	0.0851	0.1256	0.1780	0.0482	0.0535	0.0571
19	0.0848	0.1251	0.1774	0.0480	0.0533	0.0569

II-RNN

n	Recall@5	Recall@10	Recall@20	MRR@5	MRR@10	MRR@20
1	0.1718	0.2335	0.3077	0.1055	0.1137	0.1188
2	0.1615	0.2229	0.2969	0.0976	0.1057	0.1108
3	0.1512	0.2116	0.2849	0.0902	0.0983	0.1033
4	0.1428	0.2017	0.2742	0.0845	0.0923	0.0973
5	0.1359	0.1934	0.2650	0.0798	0.0874	0.0923
6	0.1301	0.1863	0.2569	0.0760	0.0835	0.0883
7	0.1254	0.1806	0.2501	0.073	0.0803	0.0851
8	0.1217	0.1759	0.2445	0.0707	0.0778	0.0826
9	0.1187	0.1720	0.2400	0.0687	0.0758	0.0805
10	0.1162	0.1687	0.2360	0.0672	0.0741	0.0787
11	0.1139	0.1658	0.2326	0.0657	0.0726	0.0771
12	0.1120	0.1633	0.2295	0.0646	0.0713	0.0758
13	0.1105	0.1613	0.2270	0.0636	0.0704	0.0748
14	0.1092	0.1596	0.2250	0.0628	0.0695	0.0740
15	0.1080	0.1581	0.2231	0.0622	0.0688	0.0732
16	0.1071	0.1569	0.2215	0.0616	0.0681	0.0726
17	0.1063	0.1558	0.2202	0.0611	0.0676	0.0720
18	0.1056	0.1549	0.2190	0.0607	0.0672	0.0716
19	0.1050	0.1541	0.2179	0.0603	0.0668	0.0712

Table 6.5: First-n-recommendations results on the Instacart dataset. The values for II-RNN are for the last hidden state implementation.

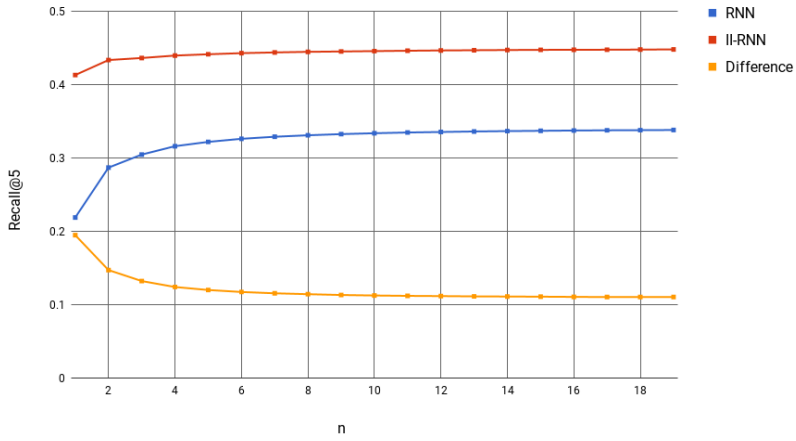

Figure 6.1: First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Reddit dataset, with Recall@5 metric.

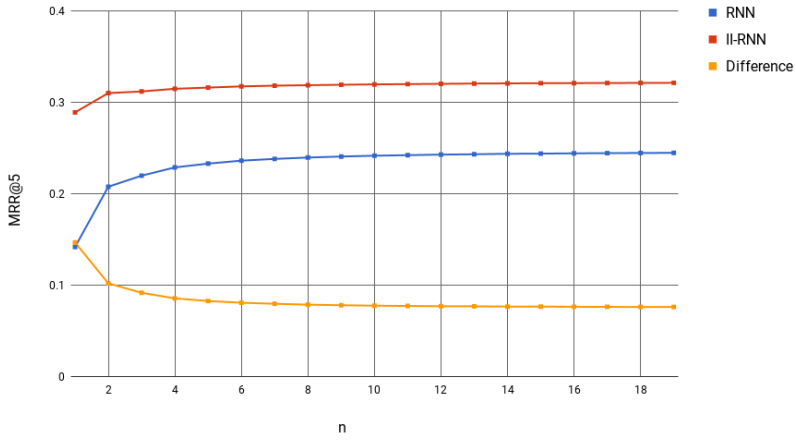

Figure 6.2: First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Reddit dataset, with MRR@5 metric.

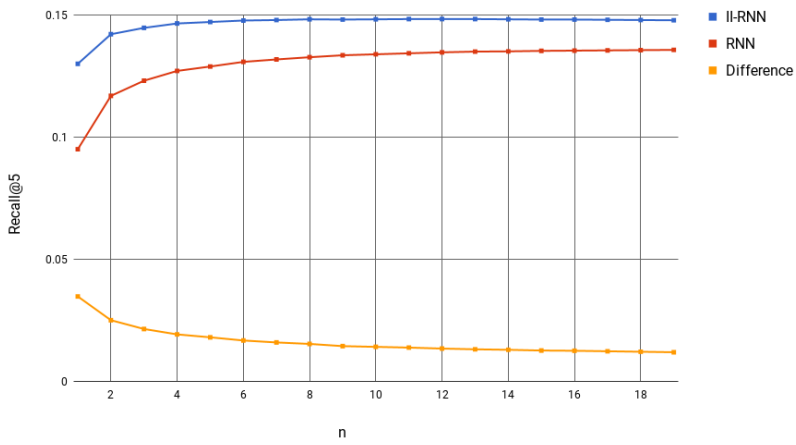

Figure 6.3: First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Last.fm dataset, with Recall@5 metric.

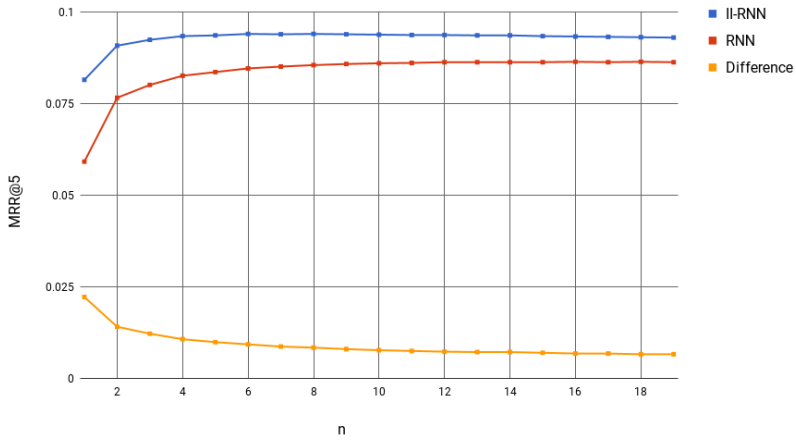

Figure 6.4: First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Last.fm dataset, with MRR@5 metric.

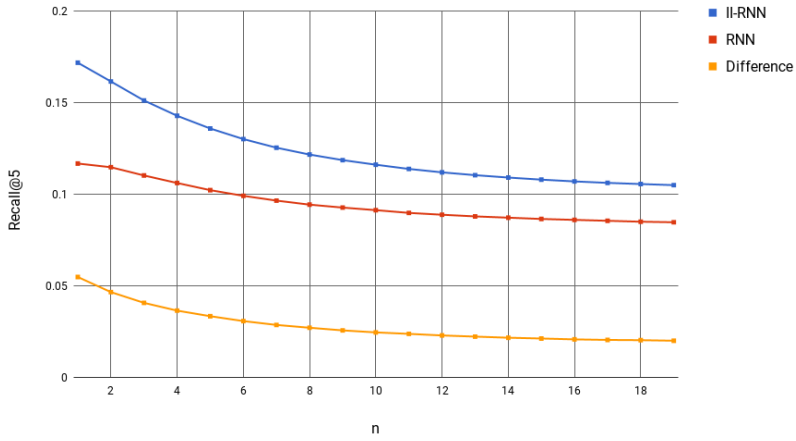

Figure 6.5: First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Instacart dataset, with Recall@5 metric.

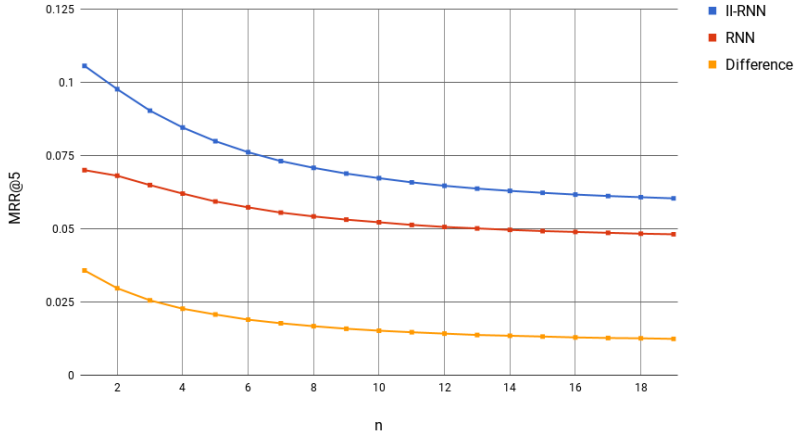

Figure 6.6: First-n-recommendations comparison of standalone intra-session RNN and II-RNN on the Instacart dataset, with MRR@5 metric.

6.2 Evaluation

Here we point out and explain interesting results found.

6.2.1 Baselines

The standalone intra-session RNN outperformed all baselines on all datasets. Based on what we have seen in other papers, this was as expected. The Item-kNN was the overall strongest baseline on the three datasets. On the Reddit and Last.fm dataset, the "most recent" baseline outperformed Item-kNN on some metrics. The reason for this is the high repetitiveness in these datasets. On the other hand, we expected the Instacart to have a very low repetitiveness, which is confirmed by the extremely low score of the "most recent" baseline. Based on the number of items in the dataset, the expected Recall@20 score for random recommendations on Instacart would be approximately 0.0005, which is higher than what the "most recent" baseline achieved. We hypothesize that the baseline scored some points on the first recommendations in each session, since those are random. But that it scored worse in the end, when trying to recommend items already added to the basket.

6.2.2 RNN and II-RNN

For the II-RNN, the intra-session RNN is the most important comparison. II-RNN is an extension to the RNN, that also considers recent past user behavior. Therefore, the II-RNN should in theory be able to perform better than the simpler intra-session RNN. Our results show that II-RNN can significantly outperform the RNN.

In addition to achieving a higher overall score on the whole sessions, the II-RNN scores significantly higher at the start of sessions. In other words, II-RNN can make strong recommendations at the start of sessions, before learning much about the current session. Furthermore, the II-RNN scores closer to its overall score at the start of sessions than the intra-session RNN does.

Note that on the Reddit dataset, the strong improvement the II-RNN has over the intra-session RNN, may bartially be explained by the short average session length in this dataset. Given that II-RNN performs much better at the start of sessions, the intra-session RNN is hampered by the lack of long sessions. However, with longer sessions the II-RNN would also have been able to achieve higher scores.

6.2.3 BPR-MF

Even though BPR-MF usually is a strong recommender, and therefore is popular to use in real applications, it was outperformed by the RNN models in our experiments. However, this might not be too surprising. BPR-MF tries to predict rankings between items in order to recommend unseen items that users would rank highly. In other words, it might perform better when used to produce novel recommendations instead of being used as a prediction recommender. Furthermore, BPR-MF does not directly apply to the session-based setting, since it is not able to dynamically change its recommendations throughout a session.

6.2.4 Dropout

We were only able to get the II-RNN to perform better than RNN on the Last.fm and Instacart datasets with dropout. Without it, II-RNN performed worse than the intra-session RNN. This shows, that even though it is intuitive that the II-RNN model should perform better, it is still necessary to tune and regularize the model to achieve this.

6.2.5 Average-Pooling and Last Hidden State

It is hard to point out why average-pooling would work better than using the last hidden state, or vice versa, for session representation. But we observe that using the last hidden state worked better on the Reddit and Instacart dataset, while average-pooling worked better on the Last.fm dataset.

The last hidden state can contain traces of information from earlier sessions as well as the one it is used to represent. Due to how GRU cells work, the intra-session RNN layer can decide to forget some features and focus on others. On the other hand, average-pooling represent all items in a session. Furthermore, average-pooling cannot give any information about the order of items in a session, and it weighs all items evenly. It is possible that these properties are advantages or disadvantages depending on the type of data worked on.

6.3 Discussion

In this section, we discuss the merits of our work, and it's limitations.

6.3.1 Usefulness of the Inter-Session Level RNN

Our work has shown that a RNN recommender system in a session-based setting, can improve recommendations by utilizing information about users' earlier sessions. We have also shown that a second RNN layer can be used to process this

information to help the first one. However, we cannot draw any conclusions on whether a second RNN layer is needed, or if a simpler solution can be equally good. Intuitively, it makes sense that the order of past sessions could be important, and in that case a RNN should be a good solution. But simpler solutions could probably also work in many cases. Therefore, it would be interesting to compare our model to other solutions for using information about past sessions to help the intra-session RNN. For example, one could feed the intra-session RNN with the session representation of the last session, directly. Or pooling could be used on the r most recent sessions' session representations, to create the initial hidden state for the RNN.

6.3.2 Importance of Retraining the Model

In real world applications, it is important that a recommender system is up to date with the latest user trends. The RNN models scored higher on the hold-one-out version of the dataset used for testing BPR-MF. This could suggest that those models perform better on current sessions than sessions further into the future. I.e. the RNN models are less effective on test sessions temporally far away from the training sessions. This makes intuitive sense, but in our case other factors also influence this difference in scores. One of those factors is that the models have more training data in the hold-one-out version of the dataset.

6.3.3 Artificial and Natural Sessions

The sessions in the Reddit and Last.fm datasets were manually extracted from the full user histories. How we divided the user histories into sessions in these datasets could affect our results. Therefore, it is very promising that the II-RNN achieved strong results on the Instacart datasets, where real users naturally created the sessions. Furthermore, the lack of repetitiveness in this dataset implies that the RNN models were able to learn more complex dependencies than just repeating already seen items. The fact that the RNN models significantly outperformed the "most recent" baseline on the other datasets, confirms this.

6.3.4 Declining Performance on the Instacart Dataset

As shown in Figure 6.5 and 6.6, both RNN models achieved weaker results towards the end of sessions. This is somewhat unintuitive, since one would think that the models would perform better towards the end, having learned more about the user's current interests. A possible explanation is that customers buy their typical items first, and that there is simply less structure to their habits further into the sessions. Hence, the results are not necessarily erroneous, although they look a bit weird.

7 Conclusion

This chapter concludes our work. We look at limitations in the proposed II-RNN model, and how these can be solved. Then we look at the contributions of our work, and we suggest several ideas for further work and improvements.

7.1 Limitations

Even though our proposed II-RNN achieves strong results on the evaluation metrics we have used, there are also other important metrics that we have not looked at.

7.1.1 Response time

In real world recommender system applications, speed is important, as recommendations should become available in real time. Training time is less important, because it can, for example, be performed regularly on a separate machine, and the trained model can be transferred into the online system when done. But the demand for a low response time from the recommendation system still needs to be addressed.

7.1.2 Scalability

In addition to response time, scalability is another important requirement in real world applications. Our largest dataset in terms of items, the Last.fm dataset, has about 100 000 items. Amazon are currently offering about 400 000 000 items [38]. Scaling up our model to handle such numbers is not trivial. A simple idea could be to have separate recommender systems for different categories. However, this still leaves us with tens of millions of items on Amazon, about 100 times more than what we encountered in the Last.fm dataset.

7.1.3 Dynamic set of items and item with short lifespans

In the datasets we have worked with, we have assumed a static set of items. I.e. the set of items does not change over time. New items are not added, and no items are removed. In many cases this is a valid assumption, because the model

can be retrained frequently (e.g. weekly) to account for changes in the item set. We have also assumed that the items have a long lifespan. I.e. that items will be relevant in the foreseeable future. This is not the case on news sites. New articles are created by the hour, and they have a short lifespan. A system that recommends last week's news, might not be very useful. In such cases, it might be necessary to create a recommender that is able to recommend items not seen during training.

7.1.4 Suggestion to Overcome the Limitations

The limitations mentioned above means that our model might need further work and modifications to adopt to those cases. Note that most of our model is operating in the embedded vector space, which means that it works on the latent features of items rather than the actual items. Therefore, we hypothesize that an approach similar to the one used in [28], and also suggested in [5], could be useful. By using a method to map items to the embedded space, and a method to find close neighbors in embedded space, the recommender need only produce recommended embeddings. A recommended embedding can then be compared to the embeddings of fresh items, and items close to the recommendation in embedded space are then recommended. In other words, the core of the recommender system would not need to know about which items are outdated and which are fresh, because it would only work with the latent features.

7.2 Contributions

Through our experiments, we have shown how a straightforward session-based RNN recommender can be expanded, making it able to consider a user's past sessions when making recommendations in the current session. We have shown that this approach gives clear improvements on all three datasets that we have experimented with. Furthermore, the proposed II-RNN model was shown to perform particularly well early in sessions. This means that the II-RNN can alleviate the cold start problem of session-based recommender systems.

We have also shown two methods for creating session-representations, which both generally work well. But which one works best, will depend on the domain and data the model is applied on.

7.3 Answering Research Questions

We now revisit our research questions from Section 1.2.

Research question 1: Can a RNN be used to learn from previous sessions, and thus produce initial information about the next session, which can help improve recommendations from the straightforward session-based RNN recommendation system implementation?

We have implemented a RNN-based session-based recommender, very similar to the one suggested in [1]. This model, the intra-session RNN, achieved strong results compared to the baselines, in line with the results from [1]. By extending this model with a second RNN-layer, that processed past sessions, we were able to improve upon the results from the standalone intra-session RNN. Our proposed model fulfills the specifications required in research question 1, and the result is improved recommendations across all three datasets that we have experimented with.

Research question 2: Given that our proposed model performs close to the straightforward RNN model or better, is it able produce improved recommendations at the beginning of sessions, and thereby deal with the cold start problem of session-based recommendations?

Our proposed model, II-RNN, significantly outperformed our standalone intra-session RNN implementation. By comparing the two models on first-n recommendations, we were able to compare the two models at different time steps throughout the sessions. On the Reddit and Last.fm datasets, both models achieved increasing scores towards the end of sessions. In our experiments, the II-RNN model showed marked improvements over the intra-session RNN at the start of sessions. Specifically, we observed the biggest performance difference on the very first recommendation. The standalone intra-session RNN was able to lessen this difference throughout the sessions. However, we observed that the difference stabilized towards the end of sessions. On the Reddit dataset, the II-RNN scored significantly higher on the first time step than the highest score the intra-session RNN was able to achieve throughout the whole session.

7.4 Further work

Here we present some ideas and research questions that could be interesting to investigate in further work on the II-RNN model.

7.4.1 Producing recommendations before the first user interaction

Both the intra-session RNN and the II-RNN models presented here, produce recommendations only after observing the first user interaction in a session. It is of course, possible to present recommendations to the user as soon as he has entered a session, before any interaction has taken place. A possible way to do this, is to let the II-RNN produce an initial dummy interaction. The intra-session RNN layer could then use this dummy interaction to produce initial recommendations. If work on this approach is done, one should also consider work done in [31], where they investigate how to predict when the user will start his next session, and what his interests will be, using a LSTM to model inter-session behavior.

7.4.2 Session representations

Although the II-RNN worked with our two methods of creating session representations, there might be other approaches that are better. In [24], both maximum-pooling and average-pooling was used. They achieved slightly better results when using maximum-pooling, so that is a possible candidate. Other methods such as bag-of-word, or using a NN to create session representations, should also be considered.

7.4.3 Learning Item Embeddings

The II-RNN learned item embeddings during the recommendation training process. It would be interesting to experiment with other methods of learning the item embeddings. Pre-training the embeddings with another method, could improve results, because it could help the II-RNN discover latent features. Also, existing encoding methods should be considered. Especially if additional information about items, such as text or images, are available. In that case, we refer to [6], where relevant work has been done.

7.4.4 Recommending Items with Short Lifespans

This is also discussed in Section 3.2 and 7.1. A solution where the core of the recommender system only works with embeddings, and another method is used to extract the actual items from the embeddings, could make the recommender more applicable to news sites and other cases where the items have short lifespans. Such an approach could enable the recommender to scale to larger item sets as well. Exploring these possibilities would be very interesting.

7.4.5 Alternating Training

In [6], discussed in Section 3.5.2 and 3.5.3, the authors found that when training a model with two parallel GRU layers, it is better to alternate training on sub-networks of the model. I.e. tuning all parts of the model in each backpropagation pass was not optimal. Alternating on freezing one of the GRU layers, while tuning the other gave better results.

Such an approach would be interesting to experiment with on the II-RNN model as well. A difference from [6] is that their GRU layers worked in parallel to produce recommendations, while in our model the GRU layers are working sequentially.

7.4.6 Novel Recommendations

Producing novel recommendations are often more interesting and useful than predictive recommendations. However, they usually require feedback from real users, making such systems harder to experiment with. Nevertheless, it would be interesting to approach novel recommendations with II-RNN.

7.4.7 Utilizing Contextual Information

As discussed in Section 3.4, additional information can be given to the recommender system in order to improve recommendations. Timestamps are often available in training data, and can be used both to find temporal distance between user interactions, and to say something about the time of the interaction. E.g. day of the week, and whether the interaction happened during holidays. Some of the papers mentioned in Section 3.4 were able to improve recommendations by utilizing such information, and it is possible that adding such information to the II-RNN could improve it as well.

Bibliography

[1] Balázs Hidasi et al. “Session-based Recommendations with Recurrent Neural Networks”. In: *CoRR* abs/1511.06939 (2015). URL: http://arxiv.org/abs/1511.06939.

[2] Bartłomiej Twardowski. “Modelling Contextual Information in Session-Aware Recommender Systems with Neural Networks”. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. RecSys ’16. Boston, Massachusetts, USA: ACM, 2016, pp. 273–276. ISBN: 978-1-4503-4035-9. DOI: 10.1145/2959100.2959162. URL: http://doi.acm.org/10.1145/2959100.2959162.

[3] Yuyu Zhang et al. “Sequential Click Prediction for Sponsored Search with Recurrent Neural Networks”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. AAAI’14. Québec City, Québec, Canada: AAAI Press, 2014, pp. 1369–1375. URL: http://dl.acm.org/citation.cfm?id=2893873.2894086.

[4] Qiang Liu et al. “Context-aware Sequential Recommendation”. In: *CoRR* abs/1609.05787 (2016). URL: http://arxiv.org/abs/1609.05787.

[5] Yong Kiam Tan, Xinxing Xu and Yong Liu. “Improved Recurrent Neural Networks for Session-based Recommendations”. In: *CoRR* abs/1606.08117 (2016). URL: http://arxiv.org/abs/1606.08117.

[6] Balázs Hidasi et al. “Parallel Recurrent Neural Network Architectures for Feature-rich Session-based Recommendations”. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. RecSys ’16. Boston, Massachusetts, USA: ACM, 2016, pp. 241–248. ISBN: 978-1-4503-4035-9. DOI: 10.1145/2959100.2959167. URL: http://doi.acm.org/10.1145/2959100.2959167.

[7] Charu C. Aggarwal. *Recommender Systems: The Textbook*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 3319296574, 9783319296579.

[8] *Matrix Factorization: A Simple Tutorial and Implementation in Python*. URL: http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/ (visited on 2017-07-01).

[9] Steffen Rendle et al. “BPR: Bayesian Personalized Ranking from Implicit Feedback”. In: *CoRR* abs/1205.2618 (2012). URL: http://arxiv.org/abs/1205.2618.

[10] Mukund Deshpande and George Karypis. “Item-based top-N Recommendation Algorithms”. In: *ACM Trans. Inf. Syst.* 22.1 (Jan. 2004), pp. 143–177. ISSN: 1046-8188. DOI: 10.1145/963770.963776. URL: http://doi.acm.org/10.1145/963770.963776.

[11] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594.

[12] Geoffrey E. Hinton. *Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair*.

[13] Zachary Chase Lipton. “A Critical Review of Recurrent Neural Networks for Sequence Learning”. In: *CoRR* abs/1506.00019 (2015). URL: http://arxiv.org/abs/1506.00019.

[14] D. E. Rumelhart, G. E. Hinton and R. J. Williams. “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1”. In: ed. by David E. Rumelhart, James L. McClelland and CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press, 1986. Chap. Learning Internal Representations by Error Propagation, pp. 318–362. ISBN: 0-262-68053-X. URL: http://dl.acm.org/citation.cfm?id=104279.104293.

[15] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2627435.2670313.

[16] Ian Goodfellow, Yoshua Bengio and Aaron Courville. “Deep Learning”. Book in preparation for MIT Press. 2016. URL: http://www.deeplearningbook.org.

[17] Sepp Hochreiter and Jürgen Schmidhuber. “Bridging Long Time Lags by Weight Guessing and "Long Short Term Memory"”. In: *SPATIOTEMPORAL MODELS IN BIOLOGICAL AND ARTIFICIAL SYSTEMS*. IOS Press, 1996, pp. 65–72.

[18] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078 (2014). URL: http://arxiv.org/abs/1406.1078.

[19] Junyoung Chung et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. In: *CoRR* abs/1412.3555 (2014). URL: http://arxiv.org/abs/1412.3555.

[20] Martin Görner. *Tensorflow and deep learning - without a PhD*. URL: https://cfp.devoxx.be/2016/talk/ULT-2698/Tensorflow_and_deep_learning_-_without_at_PhD.html (visited on 2017-06-05).

[21] Christopher Olah. *Understanding LSTMs*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 2017-06-07).

[22] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *CoRR* abs/1301.3781 (2013). URL: http://arxiv.org/abs/1301.3781.

[23] Ellen M. Voorhees. "The TREC Question Answering Track". In: *Nat. Lang. Eng.* 7.4 (Dec. 2001), pp. 361–378. ISSN: 1351-3249. DOI: 10.1017/S1351324901002789. URL: http://dx.doi.org/10.1017/S1351324901002789.

[24] Feng Yu et al. "A Dynamic Recurrent Model for Next Basket Recommendation". In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '16. Pisa, Italy: ACM, 2016, pp. 729–732. ISBN: 978-1-4503-4069-4. DOI: 10.1145/2911451.2914683. URL: http://doi.acm.org/10.1145/2911451.2914683.

[25] Shengxian Wan et al. "Next Basket Recommendation with Neural Networks". In: *Poster Proceedings of the 9th ACM Conference on Recommender Systems, RecSys 2015, Vienna, Austria, September 16, 2015*. 2015. URL: http://ceur-ws.org/Vol-1441/recsys2015_poster15.pdf.

[26] Yang Song, Ali Mamdouh Elkahky and Xiaodong He. "Multi-Rate Deep Learning for Temporal Recommendation". In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '16. Pisa, Italy: ACM, 2016, pp. 909–912. ISBN: 978-1-4503-4069-4. DOI: 10.1145/2911451.2914726. URL: http://doi.acm.org/10.1145/2911451.2914726.

[27] S. Wu et al. "Personal recommendation using deep recurrent neural networks in NetEase". In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 2016, pp. 1218–1229. DOI: 10.1109/ICDE.2016.7498326.

[28] Gabriel Dulac-Arnold et al. "Reinforcement Learning in Large Discrete Action Spaces". In: *CoRR* abs/1512.07679 (2015). URL: http://arxiv.org/abs/1512.07679.

[29] Hanjun Dai et al. "Recurrent Coevolutionary Feature Embedding Processes for Recommendation". In: *CoRR* abs/1609.03675 (2016). URL: http://arxiv.org/abs/1609.03675.

[30] Chao-Yuan Wu et al. “Recurrent Recommender Networks”. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. WSDM ’17. Cambridge, United Kingdom: ACM, 2017, pp. 495–503. ISBN: 978-1-4503-4675-7. DOI: 10.1145/3018661.3018689. URL: http://doi.acm.org/10.1145/3018661.3018689.

[31] How Jing and Alexander J. Smola. “Neural Survival Recommender”. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. WSDM ’17. Cambridge, United Kingdom: ACM, 2017, pp. 515–524. ISBN: 978-1-4503-4675-7. DOI: 10.1145/3018661.3018719. URL: http://doi.acm.org/10.1145/3018661.3018719.

[32] Yoav Goldberg and Omer Levy. “word2vec Explained: deriving Mikolov et al.’s negative-sampling word-embedding method”. In: *CoRR* abs/1402.3722 (2014). URL: http://arxiv.org/abs/1402.3722.

[33] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980.

[34] Balázs Hidasi. Personal communication. 2016.

[35] Thierry Bertin-Mahieux et al. “The Million Song Dataset”. In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*. 2011.

[36] *The Instacart Online Grocery Shopping Dataset 2017*. URL: https://www.instacart.com/datasets/grocery-shopping-2017 (visited on 2017-06-16).

[37] Greg Linden, Brent Smith and Jeremy York. “Amazon.Com Recommendations: Item-to-Item Collaborative Filtering”. In: *IEEE Internet Computing* 7.1 (Jan. 2003), pp. 76–80. ISSN: 1089-7801. DOI: 10.1109/MIC.2003.1167344. URL: http://dx.doi.org/10.1109/MIC.2003.1167344.

[38] *How many products are sold on Amazon.com – January 2017 Report*. URL: https://www.scrapehero.com/how-many-products-are-sold-on-amazon-com-january-2017-report/ (visited on 2017-06-20).