NTNU – Trondheim
Norwegian University of
Science and Technology

Recurrent Neural Networks for Session-based Recommendations

Ole Steinar Lillestøl Skrede

December 2016

Department of Computer and Information Science
Norwegian University of Science and Technology

Supervisor: Massimiliano Ruocco

Abstract

Contents

Chapter 1

Introduction

There are vast amount of information and products available on the web. Even within a single websites the number of items can be overwhelming for users. Examples of sites where this applies are news sites, streaming services, and e-commerce sites. Recommender systems can help creating 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 or help offering a better product.

In this paper, we will explore recommender systems, specifically session-based recommender systems. We will look at the use of deep learning and recurrent neural networks(RNN) for this, as compared to classical methods.

This introduction chapter introduces session-based recommender systems, some of the challenges in the domain, recurrent neural networks and how they fit into the problem. The second chapter, State of the art, looks at different methods that exist, but focuses on what has been done in terms of deep learning and RNNs. In the third chapter, Core, we explain our own work. Our experiments and results are laid out in the fourth chapter, Experiments and results. Lastly follows a the conclusion and further work, which summarize what has been done, which challenges remain and needs to be studied further.

1.1 Motivation

As mentioned, recommender systems can help both users looking for content and the providers of the content. 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. Spotify helps users discover new content introducing them to new music, tailored to the user, through their Discover Weekly[1] playlists. When a user buys something on an e-commerce site like Amazon, the site display items to the user, that other users bought together with the original item. These are examples where recommender systems helps users discover new content and helps user find products they are interested in.

For the provider, or seller, there may be different reasons to use recommender systems depending on their business model. Google earn money when users click on ads, so it is important for them to show relevant ads with high probability of getting clicked by the user. Similarly, YouTube earns money by showing commercials in their videos, and thus they can earn more money by making the user watch more videos. This can be accomplished by suggesting videos that the user wants to watch. Or like in the Amazon example, suggesting additional items might make the buyer but more than he had initially intended. Recommender systems are also used by Facebook and news feeds which tries to filter out what is shown to the user.

In the examples above, recommendations are made based on the user profile, which has information about the user interests. This profile is built up over time, as the user interacts with a system/site. Or in the Amazon example, the recommendation may only be based on a single or few items that the user bought. To make good recommendations in these scenarios, it is assumed that one has access to a user profile, but this might not always be the case. Multiple users may share an account, maybe the user is new, or maybe the user is not logged in at all. Sites can use cookies to get some historical data about the user when he is not logged in, but this is not very reliable either. There might not exist any cookies because the user is completely new to the site, or he may have deleted them. And even though we have access to cookies with information, the traffic might come from a computer shared by family members for example. Son and mom are probably not buying the same clothes, or listening to the same music. But even when we assume that it is possible to know who the user is, and what his interests are, he might be searching for content/products that is not related to his interests at all. An example of this could be a professional angler that wants to buy beginner equipment for his nephew, or maybe the angler suddenly has decided to start playing football and is looking for football equipment. Another example is a programmer who wants to listen to calm classical music while he works, but when he goes to the gym after work, he wants to listen to energetic techno. In these situations the recommender system will not be able to make good recommendations based on what it knows about the user. Only the current session might be relevant to what the user is looking for.

Recurrent neural networks intuitively fits well as a recommender system, especially in the session-based scenario. They do not suffer from the cold start scenario, they can perform well with minimal information about items, and they make it easier to extract good features [2]. Also, RNNs does not suffer from some of the assumptions that classical neural networks do. With recent advances in architectures for RNN, ways to circumvent training problems, and better hardware, RNNs have become very feasible to use. Much research has been done on RNNs recently, some of this research focuses on using RNN as a session-based recommender system. The results are promising, and it seems like RNNs are fully capable of competing with state of the art recommender systems.

1.2 Objectives

The goal of this paper and the work described in it, is to implement an RNN and test how it performs as a recommender system. We explore recommender systems, particularly those that deal with session-based recommendation. Specifically we look at the use of RNNs as session-based recommender systems. We want to know if they can compete with other models currently in use. Furthermore, we look at different architectures and ways of optimizing them to improve results. Our work is inspired, mainly, by the work in [3] and [4]. The first paper explores the use of an RNN for session-based recommendations compared to other methods. The second paper explores how adding context information as input to the network can improve the results.

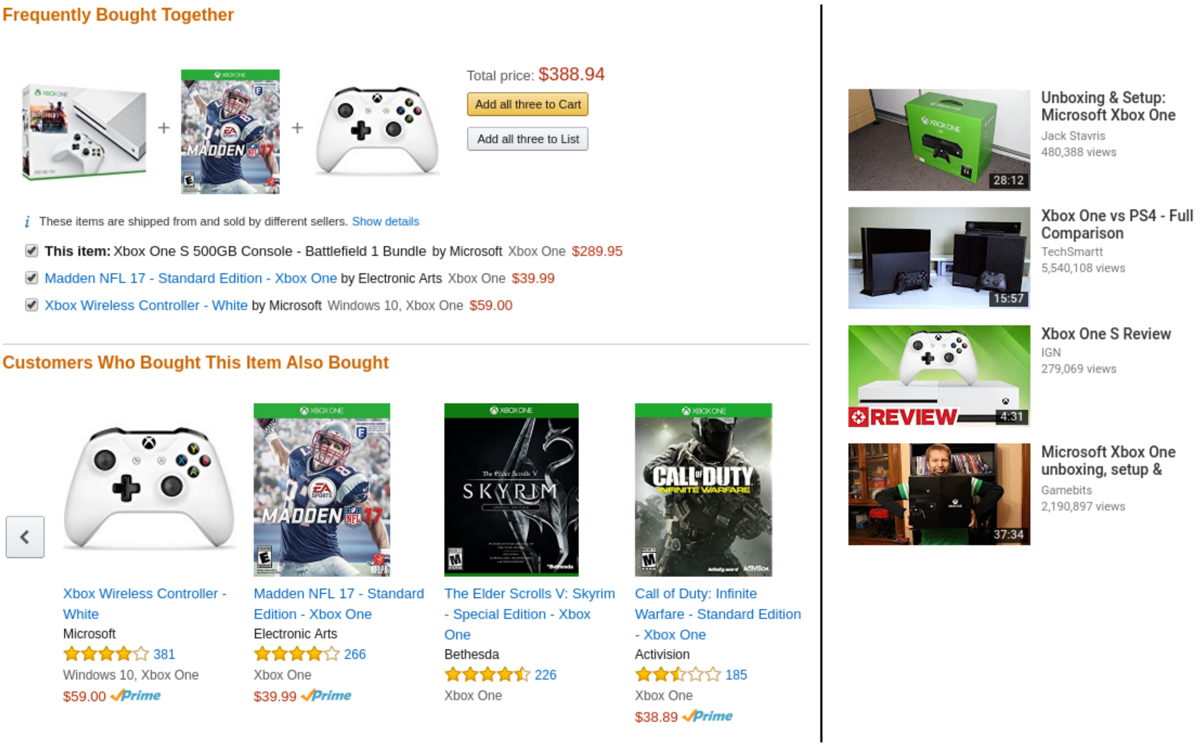
1.3 Background

In this section we explain, on a high level, terms and concepts used in this paper.

1.3.1 Recommender systems

As touched upon in 1.1, recommender systems are systems that try to predict a users 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. Figure 1.1 shows an example of recommendations made by Amazon and YouTube when looking at a Xbox One.

Note that there is a subtle difference between predicting which items a user will like, and which items he is interested in interacting with. The former case involves predicting what rating a user will give a movie or whether a user is likely to purchase one or any item. On the other hand, the latter case does not worry about how the user will interact with an item, but whether the user will choose to interact with the item, given the choice. To exemplify this, lets look at a movie streaming site. Should the site recommend movies that the user will rate



Frequently Bought Together

Total price: $388.94

Add all three to Cart

Add all three to List

i These items are shipped from and sold by different sellers. Show details

☑ **This item:** Xbox One S 500GB Console - Battlefield 1 Bundle by Microsoft Xbox One $289.95

☑ Madden NFL 17 - Standard Edition - Xbox One by Electronic Arts Xbox One $39.99

☑ Xbox Wireless Controller - White by Microsoft Windows 10, Xbox One $59.00

Customers Who Bought This Item Also Bought

Xbox Wireless Controller - White
Microsoft
★★★★☆ 381
Windows 10, Xbox One
$59.00 ✓Prime

Madden NFL 17 - Standard Edition - Xbox One
Electronic Arts
★★★★☆ 266
Xbox One
$39.99 ✓Prime

The Elder Scrolls V: Skyrim - Special Edition - Xbox One
Bethesda
★★★★½ 226
Xbox One

Call of Duty: Infinite Warfare - Standard Edition - Xbox One
Activision
★★½☆☆ 185
Xbox One
$38.89 ✓Prime

Unboxing & Setup: Microsoft Xbox One
Jack Stavris
480,388 views
28:12

Xbox One vs PS4 - Full Comparison
TechSmartt
5,540,108 views
15:57

Xbox One S Review
IGN
279,069 views
4:31

Microsoft Xbox One unboxing, setup &
Gamebits
2,190,897 views
37:34

Figure 1.1: Recommendations from Amazon (left) and YouTube (right) when looking at Xbox One.

highly if he watches them, or should it recommend movies that the user is likely to watch independent of how he will rate it? The answer will probably depend on the business and other factors, the point is that there is a difference between the two approaches. In the movie example, it is reasonable to think that how a user would rate a movie is less dynamic than what he want to watch. His ratings would probably depend a lot on his personal preferences, which usually change slowly over time. What he is interested in watching at any point, however, might depend more on circumstances. This was illustrated earlier with the worker who wanted to listen to different genres at during work and his workout session.

It is therefore clear that the goal of recommender systems may vary with different businesses and settings. In this paper we are mainly concerned with predicting the next item the user will choose to interact with.

There are two classical approaches to recommender systems, collaborative filtering and content-based filtering. These two methods can be combined into a hybrid approach.

1.3.2 Collaborative filtering

Collaborative filtering uses information about users preferences to recommend items highly rated by similar users. 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 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 and more movies, the system is able to make better recommendations. To make the recommendations, the system groups together similar users. 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 similar they rate movies. This is illustrated in Figure 1.2. The system predicts what rating the question mark will be by guessing the same rating as those given by similar users (highlighted in green).

Many sites have a huge amount of items, and the average user only interacts with a handful of these, this creates a sparsity problem for collaborative filtering methods. These methods also face the cold start problem.

One of the most popular collaborative filtering algorithms is matrix factorization [5].

1.3.3 Cold start

The cold start problem occurs when a recommendation approach is based on having a large amount of data on the user. When a new user occurs the model is not

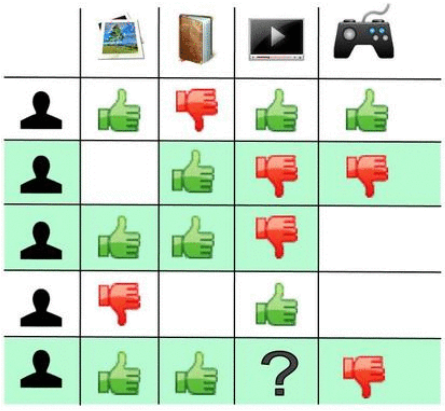


Figure 1.2: Illustration of collaborative filtering

Source: https://en.wikipedia.org/wiki/Collaborative_filtering#/media/File:Collaborative_filtering.gif

capable of making accurate recommendations due to the lack of user information. Collaborative filtering methods suffers from the cold start problem, but when they do have access to rich user information, they are usually able to make better recommendations than content-based filtering methods.

Due to the cold start problem, collaborative filtering approaches alone, are not well suited for session-based recommendations.

1.3.4 Content-based filtering

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 profile 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. The content-based approach does not suffer from the cold start problem to the same degree as collaborative methods, but content-based methods are more limited in their recommendations. They are only able to recommend items similar to those the user has already interacted with. With respect to this, the collaborative filtering approach might offer more valuable recommendations.

Some content-based approaches include decision trees, neural networks, and cluster analysis.

1.3.5 Session-based recommender systems

Session-based recommender systems tries to do predictions mostly based on the current user session. Sessions are often limited in time, and, as we have seen, the users interests might vary between sessions. Generally we no information about the user, and since the sessions are short, we do not get much information either. There may be many reasons for this lack of information. Some examples are new users, users sharing accounts, and small e-commerce sites where each unique user only visits it a couple of times. Furthermore, even though we have access to information about the users interest, this might be of minimal use if the users interests vary greatly between sessions. A session usually consists of multiple user actions on a number of items, in the order the actions happen, constrained to a short timespan. What ”short timespan” means will depend on the domain, but it is often in the order of minutes or a few hours.

1.3.6 Neural networks

Neural networks are models inspired by the human brain. They consist of layers of neurons, often referred to as nodes or units, who can receive an input signals, and depending on the strength of the signals, output signals of its own. The nodes are grouped together in layers, and the nodes in each layer output values to the next layer based on the input from the last layer. Values can also be sent back into the same layer they originated from. Each neuron can receive values from multiple neurons, and each neuron can output values to multiple neurons. Each input value to a neuron is weighted, and all the weighted inputs are summed together. A bias might also be added to this sum. This sum is sent through a function, called the activation function, and the function value is passed on to the next neurons. This is illustrated in Figure 1.3. Mathematically, the layers can be represented as vectors, where each index corresponds to one node, the weights can be stored in a matrix and matrix multiplication can then be used to successively compute the vector for each layer.

An example application might be to let a neural network guess whether it is going to rain in the afternoon, by presenting it with values representing different conditions in the morning. These conditions, or features, could be temperature, air humidity, and whether it rained yesterday. To feed the network these features, they need to be encoded into a vector of real values. This could be done by using a vector with 5 dimensions, where the first index is a 1 or 0, indicating whether it rained or not yesterday. The second index could be the air humidity, scaled down to the range (0,1). And finally, the last 3 indexes could be used to represent the temperature, for example, by creating three buckets. If the temperature is above 20 °C, the third index would be set to 1, the fourth if the temperature is

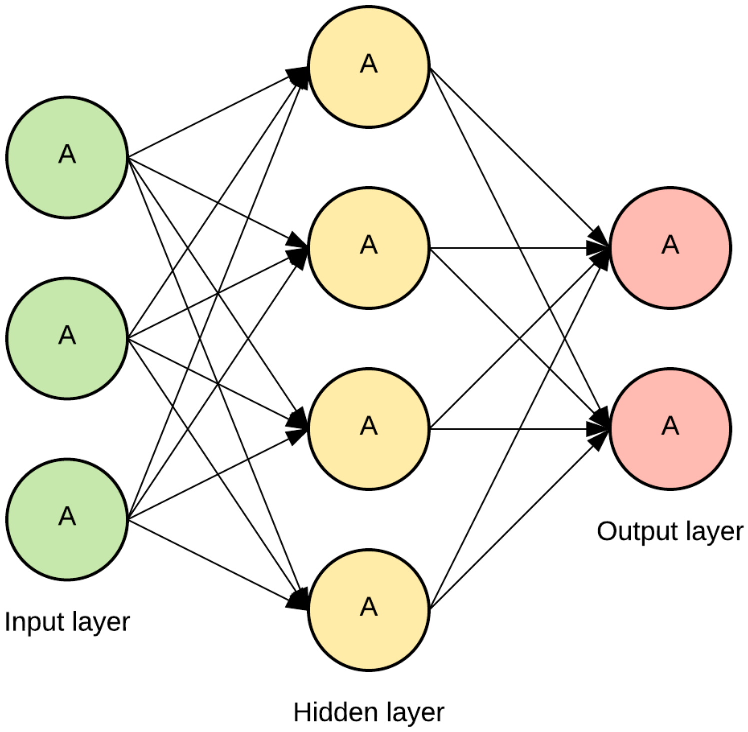


Figure 1.3: A feed forward network. Exmaples are inserted into the network by setting the values in the input layer. The values of the nodes in each layer are computed as a function of the values in the nodes in the prior layer.

between 0 °C and 20 °C, and the fifth index would be used if the temperature is below 0 °C. The output layer could be a vector with 2 dimensions, where one index would indicate rain, and the other would indicate no rain. Then the index with the highest value could be used as the networks prediction.

The layers between the input and the output layers are called hidden layers. A neural network can any number of hidden layers, this includes not having a hidden layer at all. The width of a layer is the number of nodes in the layer. Adding more layers to a network increases its capacity, which means that it becomes capable of solving more complex problems. However, with more layers, the network also becomes harder to train, both in terms of number of calculations and because more layers creates a bigger, and more complex search space.

The most popular ways of training uses backpropagation. Different variations exists, but the main idea is the following. First the network is presented with an example, and the values of all the nodes in the network is computed. Error values are calculated for the output layer, based on its difference from the target output. These errors are then backpropagated through the network. This means that the error in each node is dependent on the input it gets from the previous, upstream, nodes. So the error of the upstream nodes can be calculated by how much they contributed to the error in the downstream nodes. Based on this, the weights in the network can be adjusted. Let us look at a very basic example. A single output node, O, ends up with the value 1. This is the sum of 2 and -1, which O got from nodes A and B respectively. If the target value of O was 0.5, then the weights from A and B to O, should be adjusted so that the sum of the inputs gets skewed towards 0.5. So after the adjustments, A and B would output 1.75 and -1.25 instead, on the same example. Exactly how the weights are updated, depends on the chosen training algorithm and the parameters used, but the main idea is mostly the same.

In more correct terms, the output error is calculated with a loss function. Greater error means greater loss. But the loss does not have to be as simple as the difference between the target values and the computed values, more delicate functions can be applied. The most successful algorithm for training neural networks is called backpropagation and was introduced by Rumelhart et al. in 1985 [6] [7]. Backpropagation uses the chain rule to calculate the derivative of the loss function L with respect to each parameter in the network. The weights are then adjusted by gradient descent [6].

Figure 1.3 shows a fully connected feed forward network. Fully connected means that each node in a layer is connected to all the nodes in the next layer. Feed forward means that there are no cycles or recurrent connections, all connections go to the next layer, towards the output layer.

Note that the nodes in each layer in a neural network can be represented by real valued vectors, with dimensionality equal to the number of nodes. The weights between layers are then simply matrix multiplications. The activation functions can be applied to each value in a vector.

1.3.7 Dropout

Dropout is a regularization technique that can be used when training neural networks. As the name implies, a subset of the nodes are deactivated. Deactivating subsets of the nodes during training, can help avoid overfitting and it can speed up the training [8]. Speed up is achieved because there are less nodes to train. By randomly deactivating node, the network becomes more robust, since it cannot rely to much on any one node to produce good results.

1.3.8 Minibatches

When training neural networks, one can calculate the error for multiple training examples and then update the weights based on the average of these examples. 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 batches. [9]

1.3.9 Deep learning

The small network shown in Figure 1.3 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, with it 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. 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 are able to learn and extract good features themselves [9]. 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 [9].

1.3.10 Recurrent neural networks

Recurrent neural networks(RNN) area form of neural network that is suited for processing sequences of data. Standard neural networks have no form of memory between examples, they assume that each example is independent, which often is not true. RNN solve this by using loops where information from the one 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 1.4 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.

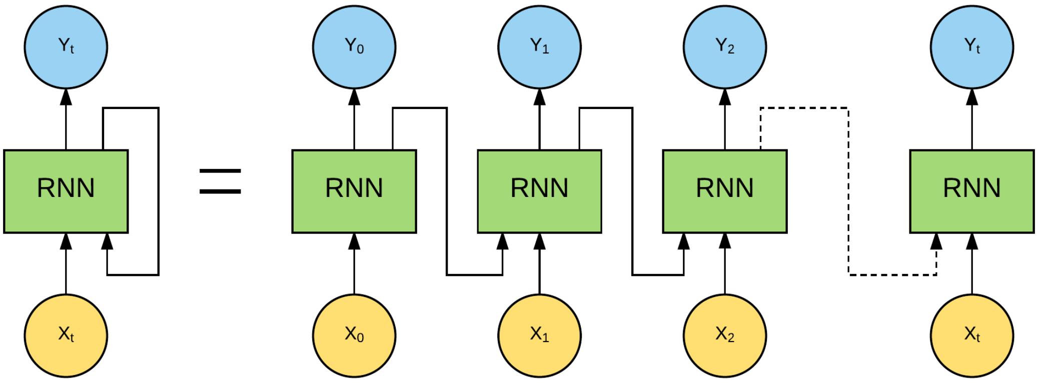


Figure 1.4: A simple RNN. Represented as a loop(left), and unrolled to t timesteps(right).

At each time step, the network takes in external input and input from the last time step, 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 RNNs, or add other types of layers such as a plain feed forward layer.

RNNs are not constrained to neither fixed sized inputs nor outputs. 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 fixed sized images.

1.3.11 LSTM and GRU

Early RNN had troubles 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 [6]. This was solved by introducing a memory cell. The new model introduced by Hochreiter and Schmidhuber in 1997 and is called *long short-term memory*(LSTM) [10]. Improvements to the original model has been made over time. 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 at each time step, and how much new information they want to add.

More recently, in 2014, Cho et al. [11] 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 gated recurrent unit (GRU). RNNs using either of the two units have been shown to perform well on tasks that require capturing long-term dependencies [12].

1.3.12 One-hot vectors

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

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

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 have no more meaning than as an identification number. That is, the numbers does not describe the classes, and the assignment of numbers is indifferent. For example, if we have a set of 10 clothing items, then we can assert a number, an id, to each of them. However, item number 3 is not an average of item number 2 and 4, even though its id is. So if we wanted to feed the items into a neural network, it would make more sense to use a one-hot encoding. Item number 3 would be encoded as a one-hot vector where the value at index 3 (or 2 for 0-indexing) is one, and the others are zero. This way, it is much easier for the network to separate between the items. In a similar way, if we wanted the network to output one of these items, it is recommended to 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 item that corresponds to the chosen index, is the correct output.

1.3.13 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 down to a lower dimensional space, 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 embeddings, it can learn to map vectors that represent similar things to similar vectors. This is used in Word2Vec [13] to map words with similar words to similar vectors in vector space.

1.3.14 RNN as a session-based recommender system

Session-based recommender systems face the task of doing recommendations based only on short session data. Collaborative filtering approaches fall short here, and usually content-based filtering is used instead, i.e. recommending similar items. The data in a session consists of a sequence of user actions, that may vary in length, where the actions within a session are likely to be dependent. These properties fits well with RNNs, and therefore it is interesting to explore RNNs potential as session-based recommenders.

Intuitively, a RNN should be able to capture dependencies between items, like content-based filtering, but with its memory capabilities a RNN should also be able to consider the whole session, which could lead to more accurate predictions. Recent papers have shown promising results in applying RNNs as session-based recommenders [3] [14] [2], and in the next chapter we look at these and other relevant papers.

Chapter 2

State of the Art

In this chapter we examine work that has been done in using deep learning for recommender systems. In particular, we are interested in the use of RNNs and the session-based recommender system setting.

2.1 Other approaches

Here we briefly mention some of the most used or successful non-RNN approaches to recommender systems.

2.1.1 The item-to-item approach

In recommender systems where a user profile is available, Matrix factorization and neighborhood models have been among the most popular approaches. Due to the missing user information on the session-based scenario, these methods do not work well there. In session-based recommendations, a popular approach is item-to-item recommendation. The item-to-item method precomputes a similarity matrix for the items. Items who are similar, based on the available session data, get high scores, and recommendation is done simply by recommending similar items. Similarity is calculated by looking at which items are often clicked in the same sessions. This method is simple and effective, and has been widely used [3]. However, it only takes into account the last click.

2.1.2 Markov models

Markov models are able to representing time dependencies. Hidden Markov models model an observed sequence as probabilistic dependent on a sequence of unobserved states [6]. The problem is that as the time window grows, the state space grows exponentially. This makes Markov models infeasible on longer sequences. RNNs do not have this problem, because the number of states that can be represented grows exponentially with the number of nodes in a layer. In addition, the complexity of inference and training grows at most quadratically [6].

2.1.3 Deep learning

Restricted Boltzmann Machines (RBM) have successfully been used for collaborative filtering [15] [3], and has been shown to be one of the best performing models for that approach. The RBM model performs recommendations based on modeling user-item interaction.

Deep models have also been used to extract content features, and then combined with a collaborative filtering model to enhance the results [16] [17] [3].

In 2015, Wan et al. [18] experimented with a three layered neural network for next basket recommendation. A basket here refers to the shopping basket on an e-commerce site. So the setting is a e-commerce site where users have done some purchases, where each purchase consists of a basket with one or more items. In this scenario the recommender system has access to the users history, so it is not a session-based problem. Wan et al. argues that recommendations should be based on the full user history, not only the last purchases. To illustrate, a user might purchase a computer, then he might purchase groceries in his next basket, and then he wants to buy some accessories to his computer. They feed the network with the user history by creating vectors that represent the average of each basket, and creating an embedded concatenation of these vectors. Their network outperforms the baselines they compare it to.

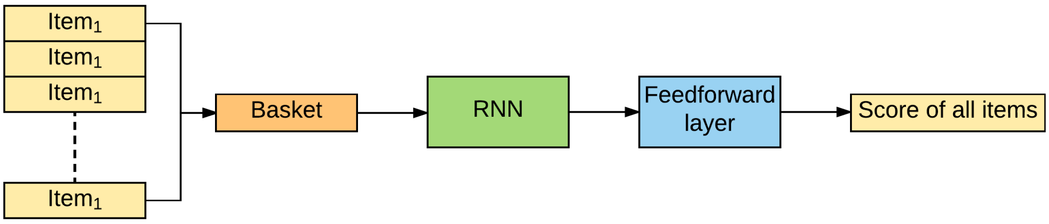


Figure 2.1: General architecture of the RNN from ”A dynamic recurrent model for next basket recommendation” by Yu et al. [19]

A paper by Yu et al. from 2016 [19] suggests using a RNN to perform next basket recommendation. Their network architecture is illustrated in Figure 2.1. They represent each user basket by pooling together the item vectors in the basket. The basket representation is sent through a simple RNN and outputs a vector with a score for each item. Their model performs better than the baselines over two datasets. This is not a session-based recommender system, but in the next section we look at a paper that applies a similar model as a session-based recommender.

2.2 Session-based recommendations with recurrent neural networks

Hidasi et al. published the paper ”Session-based Recommendations with Recurrent Neural Networks” in 2016 [3]. This paper proposes to use a RNN for session-based recommendations, and shows that a basic RNN can achieve remarkable results. They also deal with sparsity issues, and introduce a new ranking loss function for training the network. Their general network architecture is shown in Figure 2.2. The proposed network is pretty straightforward, this is also the first RNN architecture discussed in this paper, therefore we will go through the layers in detail.

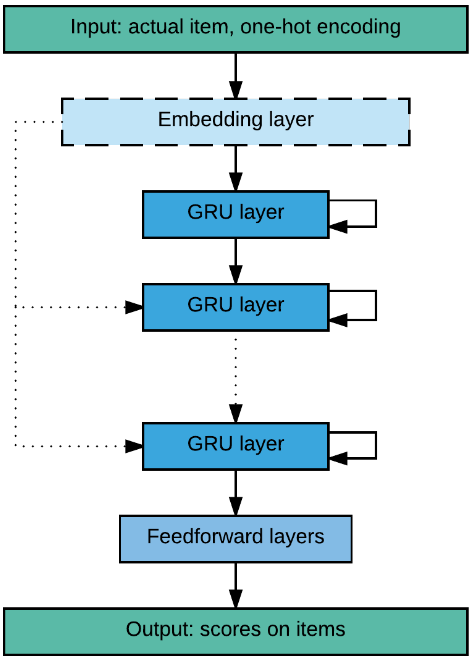


Figure 2.2: General architecture of the RNN from ”Session-based recommendations with recurrent neural networks”. [3]

They experimented with two different datasets. Both datasets contains sequences of user clicks with timestamps. One dataset has clicks on items from an e-commerce site from the RecSys Challenge 2015 [20], while the other contains clicks on videos from a YouTube-like platform. Various modifications of the network in Figure 2.2 were tested. We discuss these as we explain the architecture, layer by layer.

2.2.1 Input layer

Two different ways of encoding the input were compared. The first encoding was a pure one-hot encoding. The second was a weighted sum of the first representations, where earlier events are discounted, and the entire vector was normalized before fed into the network. Adding information about all earlier events could possibly help the to reinforce the memory effect. However, in this case the RNN performed better with the pure one-hot encoding.

2.2.2 Embedding layer

In the paper, the authors mention that they experimented with using an embedding layer between the input- and the RNN layer. Their results were always better without the embedding layer. However, if the performance loss is small, an embedding layer can still be useful to speed up calculations, as mentioned in 1.3.13. In the recommender system setting, the number of possible items can be very large, also the system need to be responsive, so potential speed optimizations should be considered.

2.2.3 GRU layers

GRU units outputs a hidden state and an output which is the prediction for the time step. The hidden state from each layers is inputted back into the same layer for each time step, but it is also fully possible to output the hidden state to the next layer, instead of or in addition to the prediction output. The authors experimented with using multiple GRU layers, where they used the hidden state as output to the next layer, except for the last GRU layer which outputted the prediction output. Best performance was achieved with a single GRU layer. It is not clear why additional layers did not improve the result, but as the authors mention, it might be because the sessions have short lifespans. If the sessions spanned over weeks, then additional layers could catch user features at different time scales. I.e. one layer could learn the more static features of the user, features which does not change over the whole session. While another layer could learn the more temporal user features, features that were only present for the past hour for example. It was also found that, when using multiple GRU layers, feeding the input, the one-hot- or the embedded vector, into the deeper GRU layers resulted in better performance.

2.2.4 Feedforward layers

In the paper, they found that using a single feedforward layer gave the best results. The feedforward layer maps a smaller vector to the same size as the input layer, where each item corresponds to one index again. By having a feedforward layer between the GRU layer and the output, the GRU layer can be much smaller. Since a GRU layer is more complex than a feedforward layer, it is also slower in terms of computations. In recommender systems, the number of items can span from thousands to millions. Therefore, keeping the GRU layer small can be necessary to achieve a feasible runtime.

2.2.5 Training

The e-commerce dataset has about 40 000 items, while the video dataset has about 330 000 videos. Therefore, the paper samples the output, and only calculates scores for some of the outputs. The network outputs a score for each item, since the output vector contains one value in each index. The correct output is one where the desired item, the next one in the session, has a higher score than the others. There are at least two reasons why the other items in each time step were not clicked. One possibility is that the user did not see the item. Another possibility is that the user saw the item, but chose to not click it, because it was not interesting to the user. For popular items, the second reason is more likely, and it is important that the network learns to give these items a low score. In the paper, this is solved by using the items from the other examples in the current mini-batch as negative examples, and of course, the next click is the positive example. By using clicks from the other examples in the mini-batch, computations for sampling is saved, in addition, these clicks are sampled by proportion to how popular they are, because popularity is decided by number of clicks.

In the paper, Bayesian Personalized Ranking [21], TOP1 (devised by the authors [3]), and cross-entropy are used to calculate the loss. With 100 units in the GRU layer, cross-entropy performed best. With 1000 units, TOP1 performed best overall, but was beaten by BPR on one metric. TOP1 performed better with 1000 units, than cross-entropy did with 100.

2.2.6 Evaluation metrics

The paper suggests two performance metrics suited for recommender systems.

Recall@N

In cases where the recommender system can recommend a set of N items, and the full set is visible to the user, it is important that good recommendations are in the set, but the ordering does not matter. The Recall@N measure is then how often the correct next click is present in the top N recommendations by the recommender system. The top N recommendations are the N items with the highest score given by the system.

MRR@N

When the user needs to do some form of scrolling to see all recommendations, the order of the recommendations becomes important. The best recommendations should be the first visible to the user. Mean reciprocal rank (MRR) covers this case. MRR@N measures the average of the reciprocal rank of the correct click in the systems ordered N recommendations. If the system recommends 20 items, then it gets a score of 0 if the correct prediction is not in the list of recommendations. If the correct prediction is in the list, the system gets a score of $\frac{1}{i}$, where i is the rank in the list. So if the correct item is first in the list, the score is $\frac{1}{1} = 1$, and if it is number five in the list, the score is $\frac{1}{5} = 0.2$. The MRR@N is the average of these scores over all the test examples.

2.2.7 Results and Conclusion

The RNN in the paper was compared to several baselines. The best performing baseline was the item-KNN. Item-KNN recommends items based on similarity, where the similarity is calculated based on items co-occurrences in sessions. This similarity is calculated with the cosine similarity between the vector of the sessions where both items occur. Specifically it is ”the number of co-occurrences of two items in sessions, divided by the square root of the product of the number of sessions in which the individual items occured” [3].

This simple item-to-item method is usually a strong baseline, and often used in practice. The paper's proposed RNN significantly outperforms the item-KNN baseline with about 15 - 30 % higher accuracy scores on the two evaluation metrics. The paper by Hidasi et al. has given us a solid RNN, with a relatively simple architecture. It performs well even though it only looks at the sequences of item clicks. No additional information about the items are fed into the network. The next papers we discuss looks at methods that can be used to further improve results.

2.3 Improving RNNs for session-based recommendations

Tan et al. [14] looks at various ways of improving the model proposed Hidasi et al. in [3]. They experiment with techniques that have worked well when neural networks have been applied to other problems, to see if those techniques can improve performance of a RNN session-based recommender as well. The same evaluation metrics, Recall@20 and MRR@20, were used, and they experimented on the same dataset [20]. They suggest four techniques, and train one model with each technique. The four models are then compared with each other and a baseline which is the model proposed by Hidasi et al. in [3].

2.3.1 Data augmentation

Based on [22], they generate multiple sequences from each sequence in the original dataset. For a sequence [x_1, x_2, ..., x_n], they create sequences [x_1, x_2], [x_1, x_2, x_3], ..., [x_1, x_2, ..., x_{n-1}], in addition to the original sequence. The last click in each sequence is used as the target click.

Another form of data augmentation they suggest is dropout performed on the click in the training sequences. This is based on results from [23]. By randomly dropping clicks in the training sequences, the model could become less sensitive ti noise clicks, which makes the model less prone to overfitting on noise. Both methods are illustrated in Figure 2.3. The models described in this section will be referred to as M1.

2.3.2 Pre-training to adapt to temporal changes

In the session-based setting, the items available for recommendation changes over time. One of the consequences is that users interest change over time, as newer items are often more exciting than old ones. These changes will be reflected in the training sequences, where the most recent examples more correctly reflects users interests. Naturally, the recommender system should be in sync with the users current preferences. One obvious solution is to discard examples from the dataset that are older than a certain threshold. However, this means a lot of data is lost. It would be better if this data could be utilized. Tan et al. suggests that a good solution is to apply pre-training. First the model is trained on the whole dataset, then it is fine-tuned on only the most recent examples. This results in a model that makes use of the full dataset, but which is focused more on the recent tendencies. This idea is based on the fine-tuning often used on image-based networks [24]. The models described in this section will be referred to as M2.

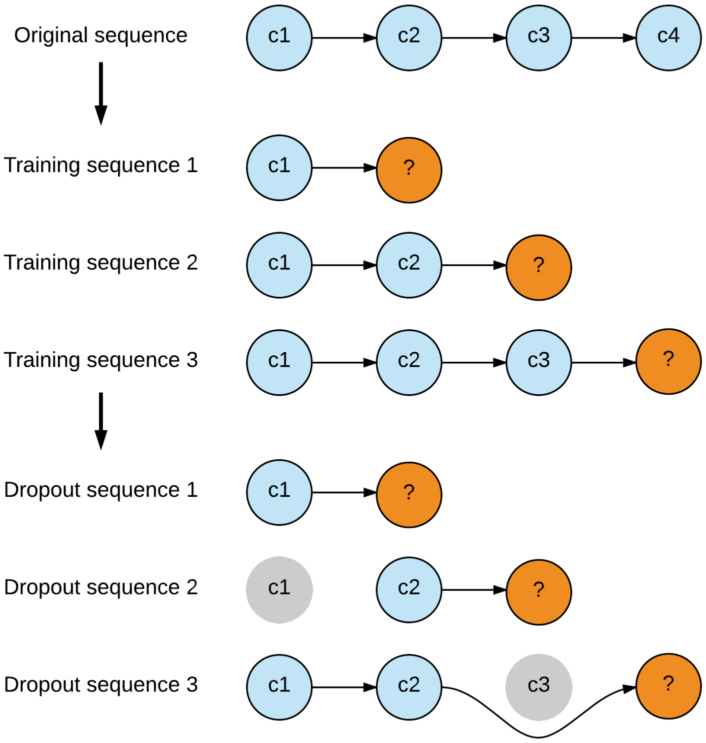


Figure 2.3: Data augmentation for training sequences as suggested in [14]. Multiple training sequences are generated from a original sequence. Random dropout is applied to the clicks in the sequences.

2.3.3 Privileged information

When training on a subsequence of a session, there is still the remaining part of the sequence, which contains information about the session. Tan et al. proposes to use this information as well when training. Look at Figure 2.4 for an illustration.

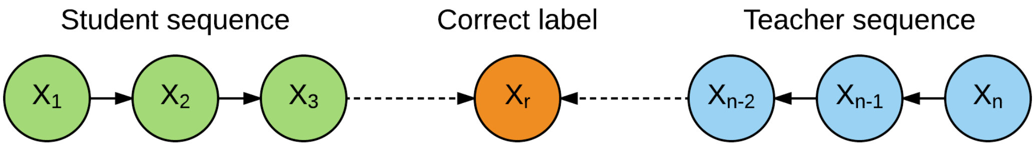


Figure 2.4: Data augmentation with teacher labels as suggested in [14]. The student model is trained to predict a tradeoff between the real label and the label predicted by the teacher model. The label produced by the teacher model is the teacher model's prediction of the correct label but on the remainder of the sequence in reverse.

For some sequence $[x_1, x_2, ..., x_r, ..., x_{n-2}, x_{n-1}, x_n]$, where x_r is the label, the model is being trained to predict x_r based on $[x_1, x_2, ..., x_{r-1}]$. This is normal training as we have already looked at. The addition is that the label that the model is trained to predict is substituted with a tradeoff between the correct label and a label produced by a teacher model. The label produced by the teacher model, is the teacher model's prediction of the correct label, x_r based on the reverse of the remaining sequence, which is $[x_n, x_{n-1}, ..., x_{r+1}]$. This approach can be useful when the available training data is small, such as for new website [14]. The models described in this section will be referred to as M3.

2.3.4 Output embeddings

Since the number of possible items to recommend can be huge, which means that outputting a vector of size equal to the number of items requires a lot of calculations, which means that the model can become less responsive (slow to calculate an output). To deal with this [14] suggests to output an item embedding instead of a full item vector. Since the embedding is a smaller vector, this means less calculations and faster predictions. The top N recommendations are extracted from the output embedding by finding the items most similar embeddings in the embedding space, where similarity is calculated with cosine similarity. For this to work, the model needs to have good embeddings to train on. The authors suggests using the embeddings found with one of the other models for this. The models described in this section will be referred to as M4.

2.3.5 Results and conclusion

The RecSys Challange 2015 dataset was preprocessed and split into test and training sets in the same way done by Hidasi et al. in [3]. To better evaluate their models, the authors sorted the training set by time and compared using the most recent fractions ($\frac{1}{256}$, $\frac{1}{64}$, $\frac{1}{4}$, $\frac{1}{1}$) of the training sequences. The same evaluation metrics were used. As baseline, the pest reported results from [3] were used.

Except when only using the most recent fraction of the training data ($\frac{1}{256}$), all proposed models, M1-M4, performed better than the baseline. M4 generally performed worst of the four models. M2 performed best, especially on the most recent fractions. M2 was not trained on the whole training set, since that would be equivalent to recreating the baseline. M1 and M3 performed best when trained on $\frac{1}{4}$ and $\frac{1}{16}$ of the most recent training examples. At best, M2 achieved about 10 percentage points better on Recall@20 and about 8 percentage points better on MRR@20 than the baseline. However, it might have been a bit unfair to use the reported performance from [3] as a baseline. Hidasi et al. did many optimizations to make their model run faster [25], and some of the tricks they used might have lowered the models performance (in terms of Recall@20 and MRR@20) in a tradeoff for faster runtime. The M2 model trained on the full training set would probably have been a more fair baseline to compare the models to. Based on the results for M2 on the fractions of the training set it was trained on, it seems reasonable to believe that M2 would have gotten a significant better score than the baseline when trained on the full training set.

Still the paper presents some nice ideas on how to improve a RNN session-based recommender. Dropping some of the oldest data can improve the results for any model. Also, the M4 model had about half the prediction time of the other models, which is useful in cases where one wants to do a tradeoff.

2.4 Utilizing context information

So far we have looked at models that performs predictions solely based on the items clicked, where the items are only represented by an id in the form of a one-hot vector. Clearly, additional information, both about the item and about the sessions, could help improve the predictions. Some possible additional information about the item is be the category of the item, an image, and a textual description of the item. Additional information about the session could be timestamps of the clicks, geolocation of the user, and weather. In this section we discuss a paper that examine if and how additional information can improve a RNN recommender.

We start in this section by looking at ”Context-aware sequential recommendation” by Liu et al. [4].

The paper suggests that modeling the time of a session and the transition time between events in the session both can give better performance. Modeling time, such as hour of the day and day of the week, can help the model capture tendencies that depend on when a session occur. As an example, think of a music streaming service. During working hours on the weekdays, users might listen more to calm music which helps them concentrating. On the evening in the weekend, there might be a rush in users listening to party music at parties. And music preferences might vary with the seasons of the year. The authors refer to this as input contexts.

Time between actions in a session might give indication about how relevant actions are to each other. Intuitively, actions that happens in short time intervals should be more relevant to each other, and visa versa. This could help the RNN decide when to what to forget and what to remember. The authors refer to this as transitional contexts.

Liu et al. proposes to define a finite number of input and transitional contexts and then give the recommender model one layer for each context. When an example is given to the model, it uses only the appropriate layers, and similarly training is done on the layers that were used for the example. More technically, the item input vector is multiplied with a input context matrix, and the hidden state vector is multiplied with a transition context matrix in each step in a sequence. It is these matrices that the authors suggest to generate for each possible context. The input contexts used in the paper are 24 hours of the day, seven days of the week, three ten-day periods in a month, and holiday or not. There can be an infinite amount of time intervals between two user actions, so the authors suggests to split the transitional contexts into bins. E.g. less than one hour, 1-2 hours, 2-3 hours, and more than three hours.

The authors does not explain how the different context matrices are trained on different data, but an approach that seems reasonable is the pre-training from [14]. A common matrix could be trained on the full dataset, independent of the different contexts, and this matrix could then be used as the initialization for the multiple context matrices. Instead of using multiple context matrices, the input context could be fed to the model as additional input. This would make the model simpler.

Two datasets were used in the paper. Taobao [26] and Movielens-1M [27], which both have contextual information in the form of timestamps. The authors compared their context-aware model to several baseline. Simple baselines such as only recommending the most popular items, context-aware baselines, and sequential methods such as standard RNN were used. Also, to compare the effect of the two contexts, the authors created two models with only input or transition contexts.

The context-aware model performed significantly better than all the compared baselines, on both datasets. It also performed better than the two models restricted to only one context. Those two models performed similarly to each other. All three context models performed significantly better than the standard RNN which performed best among the baselines.

These results strongly supports the idea that recommender systems should consider available context information in their recommendations. Note that even though, the paper only experiments with contextual information in RNNs, it seems reasonable to believe that the benefit could be applied to other recommender systems as well. The benefits of utilizing input context should not be constricted to only sequence prediction either.

2.5 Multi-rate learning

In ”Multi-rate deep learning for temporal recommendation” by Song et al. [28], they test how modeling both long- and short-term user preferences can improve a recommender system. The authors proposed model uses available user history, but the ideas should be transferable to the session-based setting.

The basic assumption of the paper is that user interests change over time. As an example, in [29], it was shown that users who visited spligle.de, a popular German news portal, were likely to be interested in football related news. The reason was that the data was collected around the time of the football world cup of 2014 [28]. Similarly, user interests may change over time, e.g. during summer and Christmas. The authors propose to use a model that combines static and temporal user features. The static features are learned by using the full training set, while the temporal features are learned by only training on the most recent examples. The question is then what the threshold for what is considered recent is. The results from [14] showed that using only very recent examples gave worse performance than with with a more relaxed threshold. At the same time, accepting too old examples means that the model will not be able to capture the current user interests. Song et al. proposes a multi-rate model, which use two separate RNNs of different temporal granularity to deal with this. To deal with the large amount of parameters the two RNNs entail, they use pre-training.

Pre-training is done differently here than in [14], which we discussed earlier. Song et al. pre-trains embeddings of the item features and the user static features. These embeddings are then used as input in the multi-rate model. Since the embeddings are smaller in dimensionality than original input vectors, the result is fewer parameters to train in the multi-rate model.

The proposed model that combined a RNN adopted to very recent user interests and a RNN adopted to more long term shifts, outperformed the compared baselines significantly. The results supports the idea that a session-based recommender system should focus on recent user behaviors, without discarding old behaviors. However, these results might be affected by the smaller amount of training examples available when only focusing on the most recent behavior.

2.6 Modeling event information

Often, there is more information about a session available than just the items clicked and timestamps. For an e-commerce site this information might include what type of action the user performed, such as viewing an item, adding it to the basket, removing it from the basket, or buying it. Websites often have a search field , which both get a lot uf user interaction as well as information through the search queries. In "Modelling contextual information in session-aware recommender systems with neural netwrks" [30], Twardowski proposes a RNN model that makes use of this information for it's recommendations. The proposed model sends embedded event information through a RNN layer, the output is concatenated with an embedded item representation, before being sent through feed forward layers to produce a prediction. On a dataset with rich search contextual information, the proposed RNN model performs significantly best. While on a dataset with less events and data, the RNN model performed worse than a matrix factorization model that was also customized to utilize event information.

2.7 Parallell RNN for feature-rich sessions

In "Parallell recurrent neural network architectures for feature-rich session-based recommendations" [31] Hidasi et al. expands upon the work from [3] and try to model richer representations of the clicked items in sessions. Since e-commerce sites often have both a picture and textual description of items, it is desirable to use this information to make better predictions. The authors suggests four different architectures that take both the item id, represented by a one-hot vector, and an image feature vector (or text feature vector) as input and computes scores on items. Also, they used two models that only had one of the vectors as input, to form baselines for comparison. The best performing architecture they proposed is illustrated in Figure 2.5.

The two input vectors are fed into two separate GRU layers, and the outputs are combined by weighting each output. Lastly, the combined output is sent through an activation layer, giving scores for all possible items. The target label is the one-hot representation of the next event in the session. Two datasets were used, one with images of the items, and one with textual descriptions. The authors suggest different ways of training the models. Standard backpropagation can be

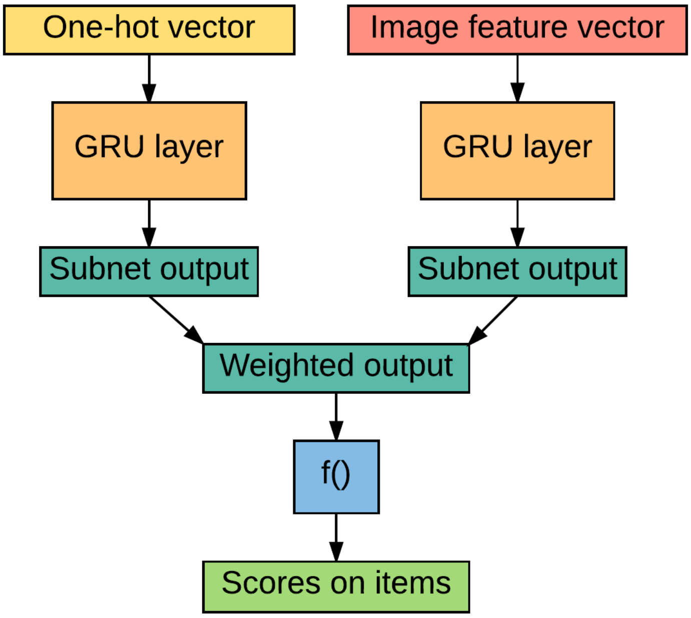


Figure 2.5: One of the proposed architectures from [31]. The image feature vector can be replaced with a text feature vector. f() is a nonlinearity applied to the output, in the paper they used tanh.

suboptimal due to the parallelism of the architectures. The problem is that the different components can end up learning the same relations from the data [31]. As a baseline, the models are trained with simultaneous backpropagation on the whole model. To improve on this, they suggest alternating the training of the parallel components, i.e. the two GRU layers. Alternation can be done per mini-batch, per epoch, or across multiple epochs. Results are compared for the different architectures and the different training approaches.

Item-KNN was used as a baseline to compare the RNNs to. Despite being simple, item-KNN is a strong basline in session-based recommendations, and often used in practice [31]. First the authors tested with 100 units in all GRU layers, the result was that all architectures outperformed the item-KNN baseline, except for the feature only model. The significantly best model was the one that is illustrated in Figure 2.5. Afterwards, the best performing model, and the baseline models were tested again, but with larger GRU layers. The parallel model was tested with 500 units in each layer, the baseline models were tested with 1000 units in their GRU layer. Performance only slightly increased in the parallel model, but the feature only model was able to outperform the item-knn baseline. On the Recall@20 metric, the ID only (only had one-hot vector as input) model was able to perform as good as the parallel model. However, on the MRR@20 metric the parallel model was still significantly better. Also, comparison of the training approaches showed that the proposed ways of doing alternate training performed better than the simultaneous approach. When using large GRU layers, the best training approach was to interleave training of the GRU layers between each mini-batch. The results shows that even when increasing the capacity of the network has diminishing returns, the performance can still be improved by adding additional data sources [31]. And also that to include the additional data effectively, appropriate models with customized training should be used.

Chapter 3

Core

In this chapter we explain the work we did on implementing a RNN for session-based recommendations. We used the Tensorflow [32] library, and our code is available here [33].

3.1 Goal

From the papers we have looked at in Chapter 2, it is clear that a RNN can perform really well as a session-based recommender. It is also clear that there are possibilities to improve performance through different techniques and architectures. To be able to do our own experimentation and exploration in the domain we needed to implement a model of our own. We decided to use Tensorflow to implement a model heavily inspired by the model by Hidasi et al. in [3]. This would allow us to familiarize ourselves with the software library and give us a model that could be used for further experimentation. We wanted our model to achieve similar performance to the one created by Hidasi et al.

3.2 Implementation

We have described the model created by Hidasi et al. in Section 2.2 and in Figure 2.2. They used Theano for their implementation, the code is available at GitHub [34]. In this section we describe our implementation. Since our model is similar to the one by Hidasi et al., the reader is referred to 2.2 for more details. Here we focus on differences between the two models, and only briefly describe the similar parts. Our model is illustrated in Figure 3.1

The input is a one-hot representation of the clicked item in each sequence, which is sent into a GRU layer with 100 units. Between each sequence, the hidden state of the GRU layer is reset. Dropout is applied to the output from the GRU

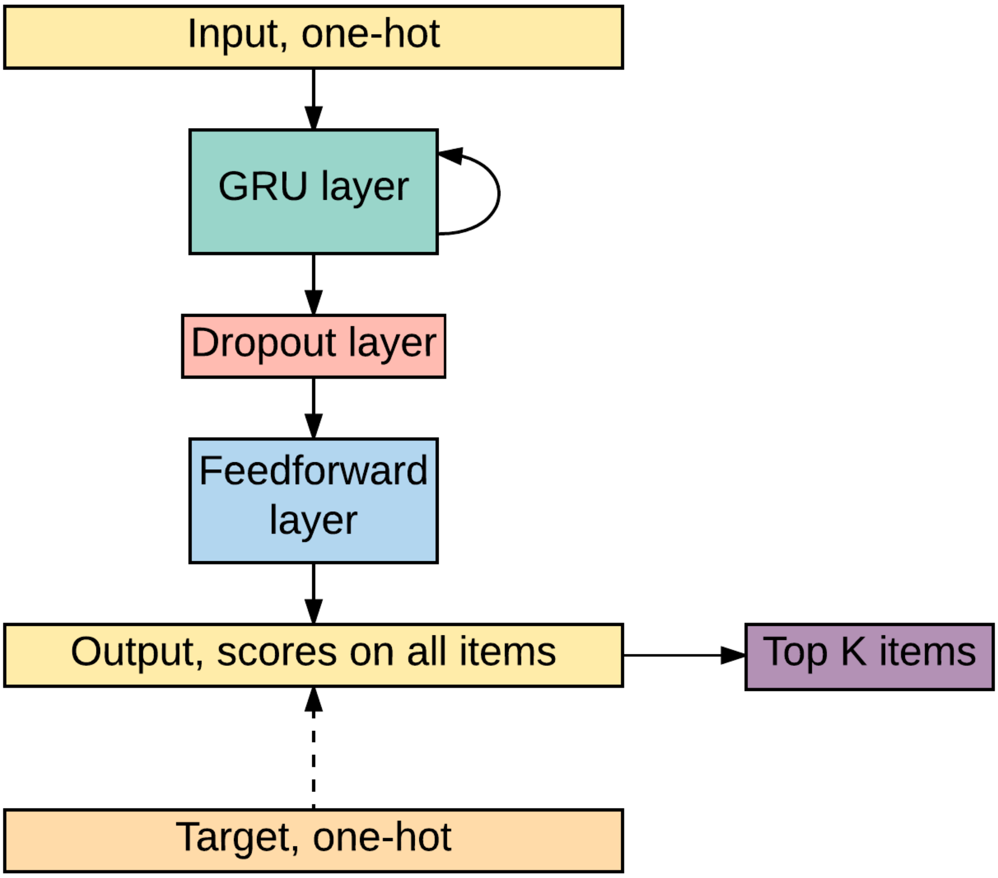


Figure 3.1: Illustration of the model we implemented, inspired by the model from [3].

layer, with a dropout rate of 50%. Dropout is not applied during testing. Then the output is sent through a feedforward layer and outputs scores for all items. Softmax cross-entropy is used on the output, with the one-hot representation of the next click as the label, to compute the loss. Adam [35] is used for training, with a learning rate of 0.001. During testing, the indexes of the top K scores are retrieved as the models predictions.

Some of our choices and parameters are chosen somewhat random based on what we know worked well in similar models. We wanted to test with different parameters and solutions to optimize our model, unfortunately we were stuck for a long time, trying to get the model to run at a feasible speed. Thus, some of the parameters can probably be optimized.

Cross-entropy for calculating loss was chosen because Tensorflow already has support for it. We added dropout as we wanted to see what effect it would have on performance (both accuracy and runtime). All sequences were padded to the same length, for example length 10. This was because Tensorflow requires it. The padded clicks are outputted as all-zero vectors from the GRU layer. We wanted to experiment with adding bias in the feedforward layer, but because we only recently were able to use masking to ignore the padded all-zero vectors, bias has not been used. The problem is that the bias would have been applied to the padded vectors, which would affect training. Training was done in mini-batches of size 200, and the examples in each mini-batch were sampled randomly from the dataset.

We tested the effect of using a feedforward layer by comparing the model with and without that layer. We only tested this on our smallest dataset, but the model was both faster and more accurate when a feedforward layer was used.

Chapter 4

Experiments and results

In this chapter we explain our experiments and results. We used two datasets, and we compared our model, the model from [3], and an item-KNN baseline on both datasets.

4.1 Datasets

We compared the models on two datasets. The first dataset is Movielens-1M [27], which was also used in [4].

4.1.1 Movielens-1M

Movielens-1M consists of movie ratings by users from https://movielens.org/. The ratings have a user-ID and a timestamp. The dataset also contains additional info about the users (age and occupation) and the movies (genre), but this was not used. The actual rating was not used either, we only used the sequences of ratings. Most of the users have rated less than 200 movies, and only users with 20 or more ratings are in the dataset. Many of the ratings are very close in time, i.e. within the same hour. This is because users enter the site and leave a batch of ratings for movies they have seen. The dataset does not represent ratings the users have given shortly after watching a movie, and not the order ni which users have watched movies either. The dataset gives info about what order users have performed ratings in (at least that is the data we are using). A problem with the data is that for a single user there are a lot of cases where ratings have the same timestamp. This means that we do not know the actual order of these ratings.

We preprocessed the data by sorting the ratings by timestamp per user. Movies that were rated less than 6 times were removed. First the ordered list of ratings for one user was treated like a session, so we had one session per user. Then 80%

of the sessions were put in a training set, and the rest in a test set. The sessions in the training set were further processed as follows. We defined a maximum length for the sequences, and longer sequences were split into multiple sequences of the desired length. This sequence splitting was done as illustrated in Figure 4.1.

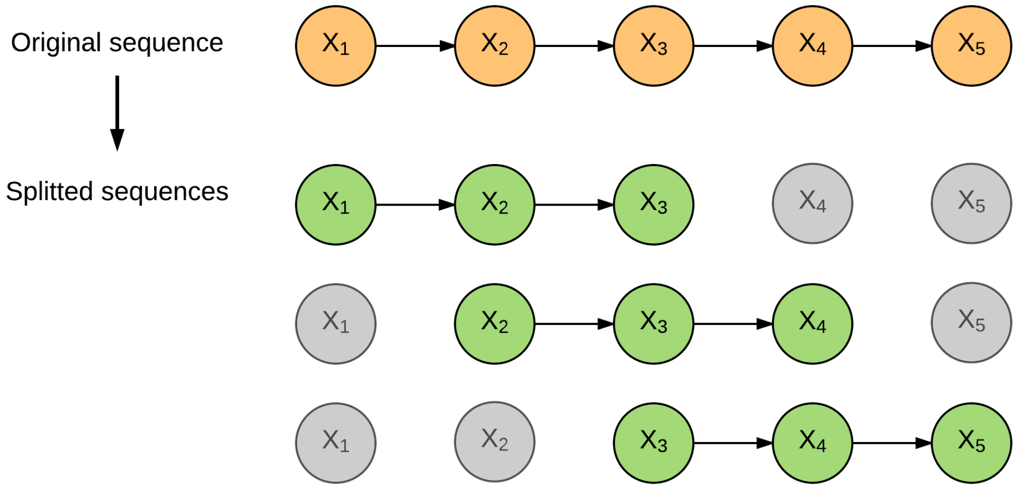


Figure 4.1: Preprocessing of sequences. Longer sequences are split into multiple sequences with the specified maximum length. Here the maximum length is 3.

We used the same maximum length as before to split the test sessions. However, we did not use overlapping when splitting because we were unsure which effects that would give on the results. So for each test sessions, we extract smaller sessions of the specified maximum length. The last extracted session is padded with zeros if necessary.

This left us with 385 272 and 19 679 sessions in the training- and testing set respectively. The sessions contained 3 377 unique movies.

The maximum length was used because Tensorflow needs a specified maximum length for RNN layers. Because of troubles with a unfeasible large runtime when training on the largest dataset, we used a small maximum length to be able to train faster.

Finally the training sessions were distributed evenly across four files. During training, the examples are read in parallel from the four files, and queued into batches.

Both RNN models used the same preprocessing of the data. For the item-KNN baseline, none of the sessions were split into smaller sessions.

4.1.2 RSC15

The second dataset we used is the same one that was used in [3] and [14]. It is the dataset from the RecSys Challange 2015 [20], from here on referred to as RSC15. First we preprocessed this dataset in the same way as was done in [3]. We did this by using their preprocessing code, available here [34]. The model from [3] used the preprocessed data from this, and the result should therefore be the same as in [3]. Their description of their preprocessing is [3]:

> The first dataset is that of RecSys Challenge 2015. This dataset contains click-streams of an e-commerce site that sometime end in purchase events. We work with the training set of the challenge and keep only the click events. We filter out session of length 1. The network is trained on $\sim$ 6 months of data, containing 7,966,257 sessions of 31,637,239 clicks on 37,483 items. We use the sessions of the subsequent day for testing. Each session is assigned to either the training or the test set, we do not split the data mid-session. Because of the nature of collaborative filtering methods, we filter out clicks from the test set where the item clicked is not in the train set. Sessions of length one are also removed from the test set. After the preprocessing we are left with 15,324 sessions of 71,222 events for the test set. This dataset will be referred to as RSC15.

For our own model we had to use a maximum length for the sessions. Since our model struggled with a high runtime on this dataset, we cut off all sessions at the maximum length of 10. I.e. we only used the 10 first clicks of each sessions, this was applied to both the training- and testing set. We also split the training set into four files as described for the Movielens-1M dataset.

4.2 Experiment

We ran all our experiments on the same hardware. The testing computer had GeForce GTX 1060 6GB GPU, 16GB RAM, and a i5-6600 CPU. Tensorflow version 0.11, Theano version 0.8.2, Cuda Toolkit 8.0, and cuDNN v5 is the software we used.

The first comparison we did was with and without a feedforward layer on our own model only, we only tested this on the Movielens-1M dataset. This was because the Movielens dataset is much smaller, and it was the only we had been able to get a practically feasible runtime on at the time. We found out that using a feedforward layer improved both speed and accuracy on Movielens-1M. Since [3] also had most success with a single feedforward layer, we decided to use that in our model.

We computed the item-KNN baseline, referred to as **Item-KNN**, on the Movielens-1M dataset. This was done with the same training and test sessions as for our RNN model, but without splitting any of the sessions. Also, we computed a baseline that always recommended the K most popular movies (by number of ratings) on the same training and test set as for item-KNN, we refer to this as **TopK**. And, of course, we tested the performance of our own model, referred to as **S-RNN**, and the model from [3], referred to as **H-RNN**. All models gave 20 predictions for each query click, and performance was measured with Recall@20 and MRR@20. The evaluation metrics are explained in 2.2.6.

Then we compared our S-RNN with H-RNN on the RSC15 dataset. Again we used Recall@20 and MRR@20 for evaluation. As a baseline we report the item-KNN baseline from [3], which was the best performing baseline in their results.

4.3 Results

We first look at the effect of using a feedforward layer. Recall@10 was used to compare the performance of the two models. The result is shown in Figure 4.2. Using a feedforward layer also gave significant speed improvements.

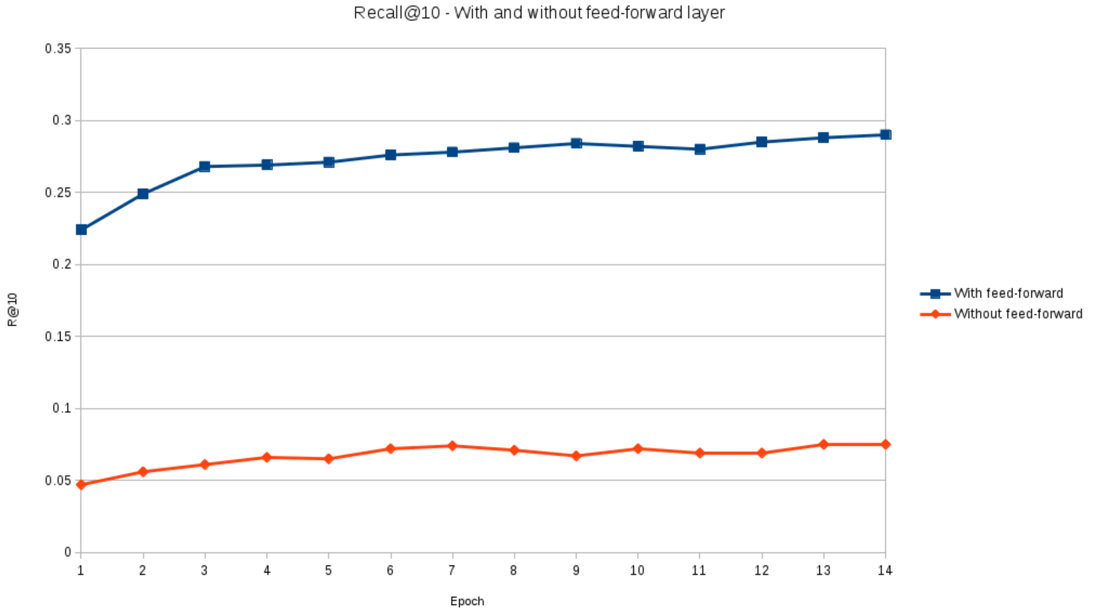


Figure 4.2: Accuracy (Recall@10) with and without a feedforward layer.

Model	Recall@20	MRR@20
TopK	0.0542	0.0106
Item-KNN	0.1002	0.0227
H-RNN	0.1482	0.0406
S-RNN	0.3012	0.0908

Table 4.1: Comparison of models and baselines on the Movielens-1M dataset.

Model	Recall@20	MRR@20
Item-KNN	0.5065	0.2048
H-RNN	0.5858	0.2290
BH-RNN	0.6206	0.2693
S-RNN	0.6812	0.2999

Table 4.2: Comparison of models and baselines on the RSC15 dataset.

Next we compare S-RNN with H-RNN, along with baselines. The results are shown in Table 4.1.

Table 4.2 shows the results from comparisons on the RSC15 dataset. Here we also report the best result reported in their paper as **BH-RNN**.

4.4 Discussion

Our results supports the results we have seen in other papers, where RNNs outperform baseline methods. S-RNN performed significantly better than H-RNN on both datasets, in terms of recall and MRR. This is a bit surprising since S-RNN is very similar to H-RNN in architecture. However, Hidasi [25] mentions that they optimized their implementation as heavily as they could to make it run fast. This is reflected in the runtime of S-RNN and H-RNN. On the RSC15 dataset, S-RNN used about 10 times as long time to finish one epoch as H-RNN did. However, S-RNN still performed better than H-RNN after S-RNN had ran for only one epoch. It is hard to do a proper comparison here. First, we suspect that there is an error in the mini-batch processing in S-RNN, which might lead cause each epoch to contain too many mini-batches. I.e. the whole training set is used, but examples might be used multiple times in each epoch. Second, [36] and [37] have found that Tensorflow (used to implement S-RNN) is slower than Theano (used to implement H-RNN) on several benchmarks. Also, according to [38], Theano trains faster and uses less memory than Tensorflow on RNNs. Note that the versions of Tensorflow we use was released after [38] was written.

More importantly, we now have a decent RNN model that can be used to explore how to further improve RNNs as session-based recommender systems.

Chapter 5

Conclusion and further work

In this final chapter, we summarize our findings and look at interesting areas for further exploration.

5.1 Conclusion

In short, RNNs are very promising in the domain of session-based recommender systems. We have seen that they can compete with and outperform other state of the art models in this area. Companies are becoming interested in using RNNs in practice [2].

We have seen that there are several ways to improve the basic RNN recommender. One of the most interesting improvements is to utilize more information than just the item clicks. Timestamps can be used in two different ways. Using the time between actions in a session can help the recommender understand which actions are more or less relevant to each other. A long timespan between two actions might suggest that they are less relevant to each other, and that the RNN should forget much of what it know of the session. Time can also be used as an external context to the session. Time of day, day of week, month in the year, and whether it is a holiday, can further help the RNN to predict the most relevant items. Furthermore, additional information about both the action event and the corresponding item can be used. Often text or image is present and contains description of items. To use this efficiently, a good architecture must be chosen, and customized training can also help. We have looked at one such appropriate architecture, which used separate parallel GRU layers for the one-hot item encoding and the embedded description vector. Training the two GRU layers in an alternating fashion was shown to improve performance.

Different data augmentation techniques can also be helpful. One of the techniques was to pre-train the model on the full training set, and then fine-tune it on only the most recent examples. Intuitively this makes sense. We utilize the full training set, but focus the model on the most recent, up to date, examples. Older examples might contain trends that have later disappeared.

News sites and other applications where items have a very short life cycle, the items in the training examples are not the same that will be recommended. Unless the model is retrained very frequently, but this requires a lot of fresh data. The solution in these situations can be to embed items into a denser space where items with similar features are close in the embedded space. The model can then "recommend" an embedding, and the actual recommendations can be found by finding items that are close to the recommendation in the embedded space.

In applications where items have a longer life-cycle, such as in many e-commerce sites, the problem is sometimes the large amount of items. The obvious solution is again to use embeddings. However, we have seen that this might reduce performance.

5.2 Further work

There are two areas we want to look further into. First, improving the use of context information. In addition to using all the context and item information, we are interested in how we can use information from user actions that are not directly related to an item. E.g. search queries contain a lot of information about what the user is interested in, and should therefore be valuable information that can be utilized.

Second, as the RNN processes a session, it learns an internal representation of the user-session. In the case were we have access to user identification, it could be possible to combine the user-session representations for a user to construct a dense representation of the user. We are interested in how this can be done, and if it can be used improve personalized recommendations.

Appendix A

List of Acronyms

GRU Gated recurrent unit

LSTM Long Short Term Memory

MRR Mean Reciprocal Rank

RBM Restricted Boltzmann Machines

RNN Recurrent Neural Networks

Bibliography

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

[2] Tobias Lang. Deep learning for understanding consumer histories. https://tech.zalando.com/blog/deep-learning-for-understanding-consumer-histories/.

[3] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks. *CoRR*, abs/1511.06939, 2015.

[4] Qiang Liu, Shu Wu, Diyi Wang, Zhaokang Li, and Liang Wang. Context-aware sequential recommendation. *CoRR*, abs/1609.05787, 2016.

[5] Albert Au Yeung. Matrix factorization: A simple tutorial and implementation in python. http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/.

[6] Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[8] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[10] 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*, pages 65–72. IOS Press, 1996.

[11] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[12] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[14] Yong Kiam Tan, Xinxing Xu, and Yong Liu. Improved recurrent neural networks for session-based recommendations. *CoRR*, abs/1606.08117, 2016.

[15] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 791–798, New York, NY, USA, 2007. ACM.

[16] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. *CoRR*, abs/1409.2944, 2014.

[17] Aäron van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 2643–2651, USA, 2013. Curran Associates Inc.

[18] Shengxian Wan, Yanyan Lan, Pengfei Wang, Jiafeng Guo, Jun Xu, and Xueqi Cheng. Next basket recommendation with neural networks. In Pablo Castells, editor, *RecSys Posters*, volume 1441 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[19] Feng Yu, Qiang Liu, Shu Wu, Liang Wang, and Tieniu Tan. 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, pages 729–732, New York, NY, USA, 2016. ACM.

[20] Recsys challange 2015. `http://2015.recsyschallenge.com/challenge.html`.

[21] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 452–461, Arlington, Virginia, United States, 2009. AUAI Press.

[22] Alexandre de Brébisson, Étienne Simon, Alex Auvolat, Pascal Vincent, and Yoshua Bengio. Artificial neural networks applied to taxi destination prediction. *CoRR*, abs/1508.00021, 2015.

[23] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. *CoRR*, abs/1512.05287, 2016.

[24] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *CoRR*, abs/1405.3531, 2014.

[25] Balázs Hidasi. Personal communication, 2016.

[26] Taobao. `https://tianchi.shuju.aliyun.com/competition/index.htm`.

[27] Movielens-1m. `http://files.grouplens.org/datasets/movielens/ml-1m.zip`.

[28] 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, pages 909–912, New York, NY, USA, 2016. ACM.

[29] Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 278–288, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[30] Bartlomiej Twardowski. Modelling contextual information in session-aware recommender systems with neural networks. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 273–276, New York, NY, USA, 2016. ACM.

[31] Balázs Hidasi, Massimo Quadrana, Alexandros Karatzoglou, and Domonkos Tikk. Parallel recurrent neural network architectures for feature-rich session-based recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 241–248, New York, NY, USA, 2016. ACM.

[32] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[33] Skrede rnn implementation. `https://github.com/ruoccoma/master_works/tree/master/skrede`.

[34] Hidasi. Gru4rec code on github. `https://github.com/hidasib/GRU4Rec`.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[36] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. *CoRR*, abs/1511.06435, 2015.

[37] glample. rnn-benchmarks. `https://github.com/glample/rnn-benchmarks`.

[38] Dan Kuster. The good, bad, & ugly of tensorflow. `https://indico.io/blog/the-good-bad-ugly-of-tensorflow/`.