Reproduction of a Deep Learning Recommendation Model for usage as News Recommendation System

Hauptprojekt Master Informatik

Timo Lange

timo.lange@haw-hamburg.de

Hamburg University of Applied Sciences

Faculty of Computer Science and Engineering

Department of Computer Science

Berliner Tor 7, 20099 Hamburg, Germany

October 18, 2021

In this project report the paper *MARS: Memory Attention-Aware Recommender System* by Zheng u. a. (2019) is reproduced to investigate its applicability to a news recommendation dataset. Initially the used datasets of the paper are explored and discrepancies are revealed. The analysis of the dataset is followed by the original experiment description and subsequently the experimental design of this work and implementation details are presented. In the course of the reproduction it turns out that important information in the preprocessing step are missing and assumptions must be made to perform the experiments. Additionally the high cardinality of the data per sample caused by multiplicativity of the possible high user history and document length turns out to be problematic. Finally the results suggest a possible good adaption to the news recommendation dataset.

Keywords – Recommender System, Deep Learning, Natural Language Processing, News

1 Introduction

In this project report the reproduction of the work of Zheng u. a. (2019) and its adaption to a news recommendation dataset is described. This work seeks to implement the papers model and tooling for further experiments with the aforementioned dataset which is covered more deeply in section 2.2. Additionally it is planned to deploy the recommender system, among others, in a production system for A/B testing, which will be covered in further work. The *MARS: Memory Attention-Aware Recommender System* from Zheng u. a. (2019) was chosen as it distinguishes itself from the competition through the attention mechanism which models the users diverse interest through *deep adaptive user representations* and is the first of its kind to the best knowledge of the author. The model attends to a kind of context item, which can be for example the current viewed content of a website, and derives a user profile dynamically adapted to the item in question. Moreover, the representation of the items and the user profile is trained and optimized jointly which is also not common. Furthermore, at a first impression, the neural network topology seems to be easily implementable for it's relative simplicity. Additionally, through the use of standard CNNs and basic dense networks the speed to yield a recommendation for a user should comply with the strict latency requirements in a real world recommendation scenario. Finally the architecture is suitable to adapt the user profile to a live stream of user interactions without the need to precompute user profiles. Despite the lack of source code of the implementation of the papers model, the reproduction of the model can be validated through the use of public available datasets used in the paper.

This work is structured as follows: At fist the public datasets used in the original paper and the news recommendation dataset are described. Thereafter the original paper with its experimental design, model design and training is outlined. This is followed by the description of the experimental design and implementation details of this work. Before the conclusion, the experimental results are depicted and discussed.

2 Dataset Overview

The used datasets, especially the news recommendation dataset 2.2, got many fields which can be leveraged for recommendations. As not to go beyond the scope of this project, just the used and derived data fields are described. For all datasets the *user* field is derived from the ratings/pageviews and constitutes of the rated/viewed items per user. The words per content is computed by simply counting continuous word characters (`re.finditer(r'\\w+', s)`).The word count may vary depending on the used technique for tokenization. A more detailed analyses of the news recommendation dataset 2.2 will be carried out in future work.

2.1 Dataset from original paper

The statistics about the dataset corresponds to the filtered data according to Zheng u. a. (2019).

2.1.1 Yahoo! Movies

The movie metadata is filtered for missing *movie_id* and *synopsis*, which are the only used fields from the content description data. The user ratings data is filtered for ratings of 5 (scala is 1 to 5). This are actually the *converted ratings* as described by the dataset readme file, which are derived from a original rating from 1(F) to 13(A+) but the MARS paper don't mention this. Furthermore all ratings are dropped for which no movie description exists and just users with at least 3 ratings are taken into account. In table 1 you can see statistics about the synopsis and their lengths and number of ratings per user. Note that in contrast to the amazon datasets there is no synopsis/content without words. The values for synopsis in parentheses are movies left after all movies are dropped which don't appear in any user ratings after filtering the user ratings.

No.	Field name	Type	Unique	Words/Entries			
				min	avg	max	Std.
1	synopsis	str	106 673	1	48.41	675	41.71
		str	(8686)	(4)	(86.89)	(612)	(63.95)
2	user	List[int]	7271	3	15.26	715	23.37

Table 1: *Yahoo! Movies* dataset: Statistics about synopsis and number of ratings per user for the *Yahoo! Movies* dataset.

2.1.2 Amazon Video Games

For the *Amazon Video Games* dataset all ratings below 5 (from 1 to 5), users with less then 10 ratings and items without description are dropped. In table 2 you can see the statistics for word count in descriptions and user history entries. Values in parenthesis are where all items filtered out with descriptions which contain no words, but only special characters, HTML code or punctuation.

No.	Field name	Type	Unique	Words/Entries			
				min	avg	max	Std.
1	description	str	47 063	0	116.54	4711	176.10
			(42 710)	(1)	(128.42)	(4711)	(180.68)
2	user	List[str]	2572	10	17.81	433	16.91
			(1743)	(10)	(17.87)	(406)	(17.52)

Table 2: *Amazon Video Games* dataset: Statistics about game descriptions and number of ratings per user for the *Amazon Video Games* dataset.

2.1.3 Amazon Movies and TV

Table 3 shows item and user statistics for the *Amazon Movies and TV* dataset. Similar to *Amazon Video Games*, only ratings of 5 (from 1 to 5) and users with 10 or more ratings are preserved. Also, items without description are filtered out. Values in parenthesis are where all items filtered out with descriptions which contain no words, but only special characters, HTML code or punctuation.

No.	Field name	Type	Unique	Words/Entries			
				min	avg	max	Std.
1	description	str	23 599	0	117.59	2590	111.86
			(23 347)	(1)	(118.86)	(2590)	(111.79)
2	user	List[str]	5187	10	24.67	819	33.96
			(5177)	(10)	(24.67)	(818)	(33.96)

Table 3: *Amazon Movies and TV* dataset: Statistics about game description and number of ratings per user for *Amazon Video Games* dataset.

2.1.4 Divergent Data

Nearly all data after filtering is different from the paper. The number of users for the *Yahoo! Movies* dataset diverges just slightly (7271 to 7642) but the difference in number of items is quite large (106 673 to 11 915). For *Amazon Video Games* the number of items match exactly(47 063) but the number of users differ slightly (2572 to 2670). The numbers for *Amazon Movies and TV* from this work and the original paper differ to a far extend for users (5187 to 22 147) and items (23 599 to 178 086). Also the filtering for items not present in ratings after the ratings are filtered is not consistent, as for *Yahoo! Movies* the original papers authors seem to filter for them because without dropping them, the amount diverges by a factor of 8.95 instead 0.72 between this and the original work. In contrast to this, the items from *Amazon Video Games* are not filtered after the rating filter in this work and the number match exactly with the original paper.

Different dataset versions are not the cause of the different data. For *Yahoo! Movies* there is just one version available from the yahoo dataset download portal. At the time of writing there are three versions of the amazon dataset. The MARS authors referenced the papers of the second dataset version, which is also used in this work. The use of the second dataset is also supported by the fact, that the number of items for *Amazon Video Games* in the original and this work matches exactly.

So the data processing of the original work is non-transparent which makes the reproduction very difficult.

2.2 News Recommendation Dataset

2.2.1 Schickler Dataset

The Dataset is kindly provided by *SCHICKLER Unternehmensberatung GmbH* [1] to support this research, which work together with different publishers and *dpa Deutsche Presse-Agentur GmbH* [2] to enhance the users news reading experience. The dataset is fully anonymized and does not contain any personal data.

The dataset consists of 313 565 551 pageviews from 56 199 311 users and 823 947 (+1 with no textual content) corresponding articles. There are 47 fields for articles and 35 fields for the pageviews. The dataset constitutes of many fields but for the implemented model only a small subset will be used, which is described in the following table:

No.	Field name	Type	Unique	Words/Entries			
				min	avg	max	Std.
1	article_full_text	str	823 947	2	398.35	30 642	279.24
2	user	List[str]	56 199 311	1	5.58	59 161	59.20

Table 4: Schickler News Recommendation dataset: Statistics about article texts and number of ratings per user for *News Recommendation* dataset.

A more deeply and throughout analysis will be carried out in a follow up work.

2.2.2 Additional DPA Data

It is planned to enrich the dataset from Schickler described in the previous section with additional data from dpa, with similar data fields described in Lange (2020) and can be done with articles for which a unique dpa article id is present. This will augment the article information with valuable data like *mediatopic*, *keywords*, *slugline*, *genre* and *subject*. This may positively impact the recommendation performance, which have to be proven in further work.

3 MARS: Memory Attention-Aware Recommender System

In the following the work of Zheng u. a. (2019) and its reproduction is described.

3.1 Original Experiment Description

In the original paper a new kind of model is developed, which is tested on the before mentioned datasets *Yahoo! Movies*, *Amazon Video Games* and *Amazon Movies and TV*. Each of the datasets

[1] https://www.schickler.de
[2] https://www.dpa.com

represent a tougher class for a recommendation system due to it's descending density and thus relative viewer samples to learn from. The model is tested against some notable recommendation models to establish baselines and test it's capabilities. The baseline models are:

1. Bayesian Personalized Ranking
2. Neural Collaborative Filtering
3. Collective Matrix Factorization
4. Collaborative Topic Regression
5. DeepFM
6. Collaborative Deep Learning
7. Wide & Deep

To compare the models, the metric *recall@N* and *Mean Average Precision(MAP)* are used, shown in equation 1 and 2. Recall measures how good the model retrieves all relevant documents and MAP measures how well the system ranks the retrieved documents.

$$recall@N = \frac{\text{\# items the user likes among the top N}}{\text{total number of items the user likes}} \tag{1}$$

$$\begin{aligned}
MAP &= \frac{1}{|U|}\sum_{i\in U} AveP(i)\\
AveP(i) &= \frac{1}{|K|}\sum_{k=1}^{K'} P_i(k) rel_i(k)\\
P_i &= \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}\\
rel_i &= \begin{cases} 1, & \text{if } k \text{ in } \{\text{relevant documents}\}.\\ 0, & \text{otherwise}\end{cases}
\end{aligned} \tag{2}$$

Where $AveP(i)$ is the average precision per query, respectively user. P_i denotes the precision for document i and rel_i the relevance of document i, which leads to irrelevant documents being ignored.

Additionally, a ablation study is carried out, in which the authors investigate to what extend the different components of the model contribute to the recommendation performance. They conclude all examined parts, *item embeddings*, *CNN* and *attention* contribute to the models performance.

Also a case study is carried out in which the interpretability of the model through the attention mechanism is underlined.

3.1.1 Model Description

The overall model architecture is depicted in figure 1. The structure of the model from the bottom to the top is as follows:

Input Y^{n_i} and Y^j are the documents associated with the liked item by the user, respectively the item to attend to.

CNN for Item Representation $f_{user}(:;\Psi)$ and $f_{item}(:;\Omega)$ are the CNN's to learn the item representations for the users memory component and the item representation of of item j. Where Ψ and Ω are the learnable parameters.

Memory Component & Item Representation C represents the users memory component which contains the item representations of all items the user likes and V_j is the item representation of the item to attend to.

Attention Vector The attention vector α^{ij} assigns each item representation i in C a weighting, build with $softmax(C^T v_j)$

Adaptive User Representation The adaptive user representation u_j^i is build by applying α^{ij} to C which builds a weighted sum of all items in C with $u_i^j = C\alpha^{ij}$.

Preference Score The preference score $r_{ij} = {u_i^j}^T v_j$ is build by the inner product of the user representation u_i^j and the item representation v_j

The final item recommendation list is build by computing the preference score r_{ij} for all candidate items, for which a recommendation should be generated and sort the items according to their score in descending order.

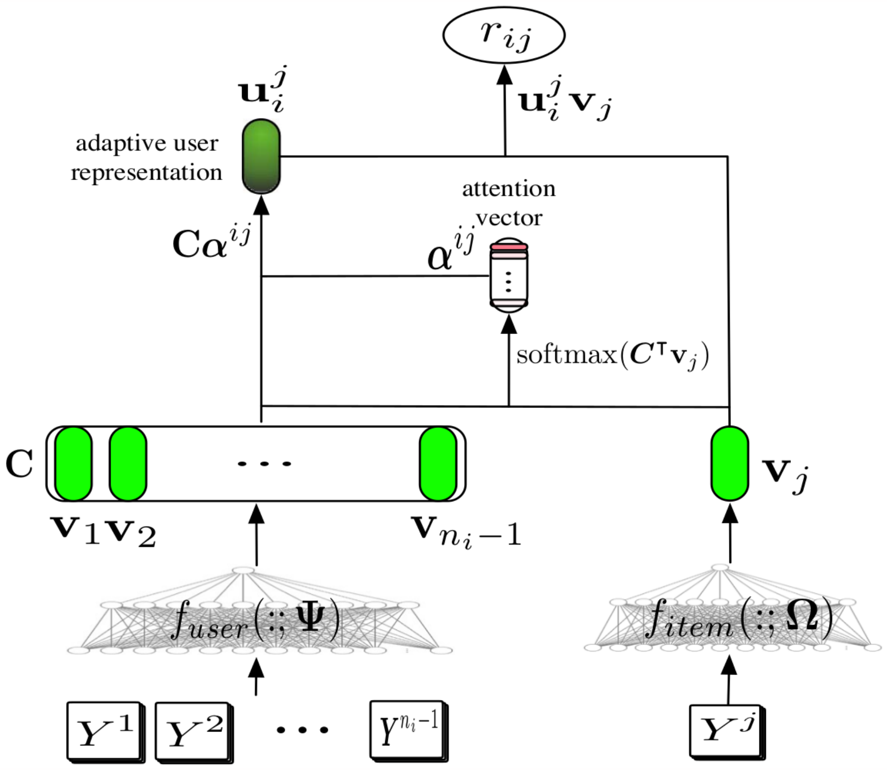
r_{ij}
$\mathbf{u}_i^j$
$\mathbf{u}_i^j \mathbf{v}_j$
adaptive user representation
attention vector
$\mathbf{C}\alpha^{ij}$
α^{ij}
$\mathrm{softmax}(C^\intercal \mathbf{v}_j)$
$\mathbf{C}$
$\mathbf{v}_1 \mathbf{v}_2$
$\mathbf{v}_{n_i-1}$
$\mathbf{v}_j$
$f_{user}(:; \boldsymbol{\Psi})$
$f_{item}(:; \boldsymbol{\Omega})$
Y^1
Y^2
Y^{n_i-1}
Y^j

Figure 1: Overall MARS architecture. Zheng u. a. (2019)

The item representation is computed with a CNN model according to the architecture outlined in figure 2. The neural network topology for the memory component representations and the context item representation are the same and only differ in the input, hence the depiction in figure 2 is valid for both CNN models. From bottom to top, the model starts with the input layer with the words of one document. This is followed by a word embedding layer. The word embeddings are further processed by one convolutional layer, which extracts contextual features, where each kernel captures one feature. The most important feature value for each feature is extracted by a Max-pooling layer. Finally the extracted features are projected through a dense layer into the final item representation space.

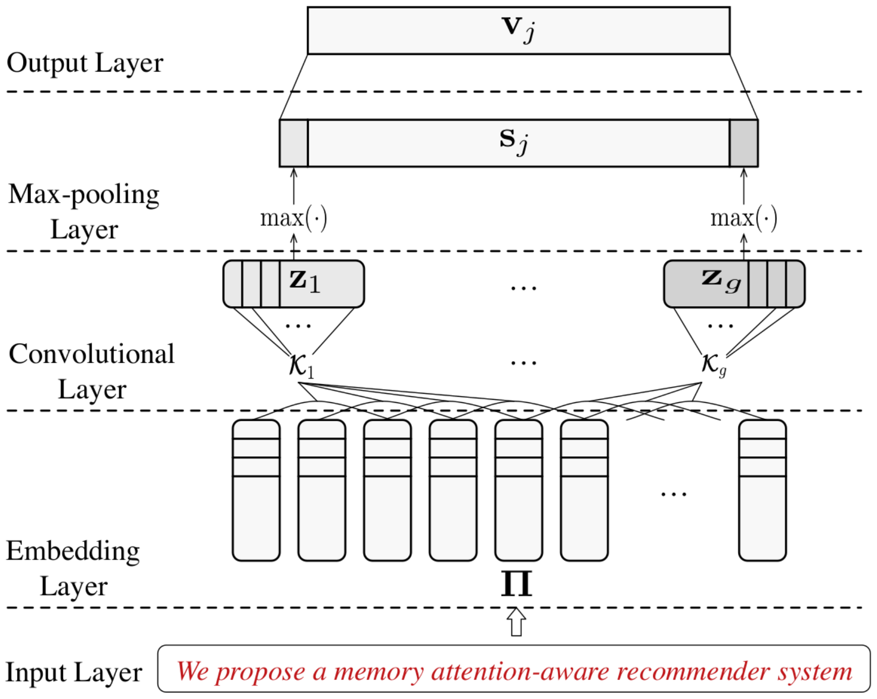

Figure 2: MARS CNN architecture used for learning item representations. Zheng u. a. (2019)

3.1.2 Model Training

To train MARS optimized for ranking, the authors took inspiration from Rendle u. a. (2009). The main components they adopted are the data generation method for training and the loss function.

Training Data The training data is formalized as:

$$\mathcal{D} = \{(i, \mathcal{I}_i^+ \setminus j, j, j') | i \in \mathcal{U} \land j \in \mathcal{I}_i^+ \land j' \in \mathcal{I}_i^-\} \tag{3}$$

Where i, j, and j' are uniformly sampled from $\mathcal{U}$, $\mathcal{I}_i^+$ and $\mathcal{I}_i^-$, which describes the set of users, the liked/viewed items by a user and the not liked/viewed items by a user. This data generation strategy, according to Rendle u. a. (2009), accounts for the skew in data where one item is overly present in many users histories and there are typically much more not liked/viewed items in contrast to liked/viewed ones. Additionally as stated by Rendle u. a. (2009), this strategy leads to a much faster convergence then e.g. iterating the training data user wise.

Loss Function Instead of scoring single items, item pairs are used to represent the users u preference of item j over item j'. The loss function is shown in equation 4.

$$\mathcal{L} = -\frac{1}{|\mathcal{D}|} \sum_{(i,\mathcal{I}_i^+ \setminus j,j,j') \in \mathcal{D}} \{ln(\sigma({u_i^j}^T v_j - {u_i^{j'}}^T v_{j'})) + \lambda_u {u_i^j}^T u_j + \lambda_u {u_i^{j'}}^T u_{j'} + \lambda_v v_j^T v_j + \lambda_v v_{j'}^T v_{j'}\} \tag{4}$$

Where σ is the sigmoid function which maps the user i's preference of item j over item j' into probabilities. u_i^j and v_j are the user and item representation for item j, $u_i^{j'}$ and $v_{j'}$ for item j'. λ_u and λ_v are regularization terms.

3.2 Experimental Design

Unfortunately, despite an attempt to get in contact with the original authors, no communication has been established. Due to the lack of information in the paper, some assumptions have been made to get an running PoC (Proof of Concept). The differences in the implementation in this work and the original work may have lead to the different results, but can't be proven without the original authors participation. Additionally, the differences in data after preprocessing, outlined in 2.1.4, will inevitable lead to divergent results.

The goal of this work is to implement an running PoC of the original papers model, inclusive all preprocessing steps and ultimately make this PoC applicable to the domain of news recommendation. The model will be prepared for offline evaluation with the news recommendation dataset of section 2.2 and online evaluation with A/B testing on a in-production news site of one of the publishers which provided the dataset. In a future work the model will be compared with other models in the aforementioned offline and online tests.

As a first step to approach the goal, is to implement the preprocessing pipeline for all three datasets used in the original paper. Secondly, the model will be implemented with *Keras* and *Tensorflow*. Next, the correct implementation of the model will be validated against the three datasets used in the original paper. In this process, the implementation will be continuously further developed and improved to meet demands such as hardware resources, like memory footprint and compute requirements and time requirements, like inference latency and training time.

3.2.1 Missing Information / Assumptions

The following information was missing to reproduce the original work:

Stopwords There is no pointer to the stopword list or library used and hence the vocabulary size will be affected by the concrete implementation.

Tokenization There is no hint how the tokenization is done. If the tokens are just split by whitespace, the vocabulary size will probably be lower as if split smartly e.g. in case of typos, in word separation (missing whitespace) or hyphen between words.

Epoch / number of steps until convergence The paper don't state how much steps of training the model needs until convergence, respectively how much were applied for the results.

train/validate/test split rounding The train split consists of a 0.3 fraction of the data and the rest is evenly divided into validation and test set but the authors don't state how the fractions are split if it results in odd numbers. If the user history is very small this will hugely affect training trough more or less training data per user available and the metrics because the models predicted ranking will have to match more or less correct items per user.

3.2.2 Preprocessing

Data filtering for all three datasets is done mostly with *pandas*, which was possible trough the small amount of data and the easy to read *CSV*, respectively *JSON* file format. For the tokenization process *Spacy* was used. The same tool was used for stopword and punctuation removal. To translate the text strings into a vector representation with a word to *integer* mapping the *tf.keras.preprocessing.text* library from *Keras* was utilized. The final data input for the model is generated with the *Tensorflow Data API* trough *tf.data.Dataset.from_generator()* and a python generator which computes the uniform distributed samples and maps the users/items to their corresponding vector representations.

3.2.3 Model Implementation Overview

Training Model Implementation The implementation of the model is done via *Keras* and *Tensorflow*, where the high-level *Keras functional API* is used when possible and lower level *Tensorflow* operations when necessary. Figure 3 shows the operational graph generated by *Tensorboard*. The big outer box includes the whole MARS model, where the output to the loss function is shown at the top and the input at the bottom, represented by an arrow. The right light purple box named *Model_context_items* is the CNN which computes the representation of the items liked and not liked from the input tuple. Both items are processed by two exact same CNNs sharing their weights and are thus depicted by just one sub model. On the left side in a light yellow box, called *time_distributed*, the item representations for the items in the user history are computed, whose result is the memory component shown in figure 1. The *time_distributed* layer is a built-in *Keras* layer which applies an passed layer to the data stepwise. In this case, the passed layer is the CNN to generate the users history items representation called *Model_user_items*, which is applied to each item in a users history. To eliminate any confusion, in *Keras*, models can also be treated as layers. Finally, with the memory component and the liked/not liked item representations, the attention for the two items is computed from which the user representations for the liked and not liked items are derived. Both, item and user representations are concatenated and feed to the loss function.

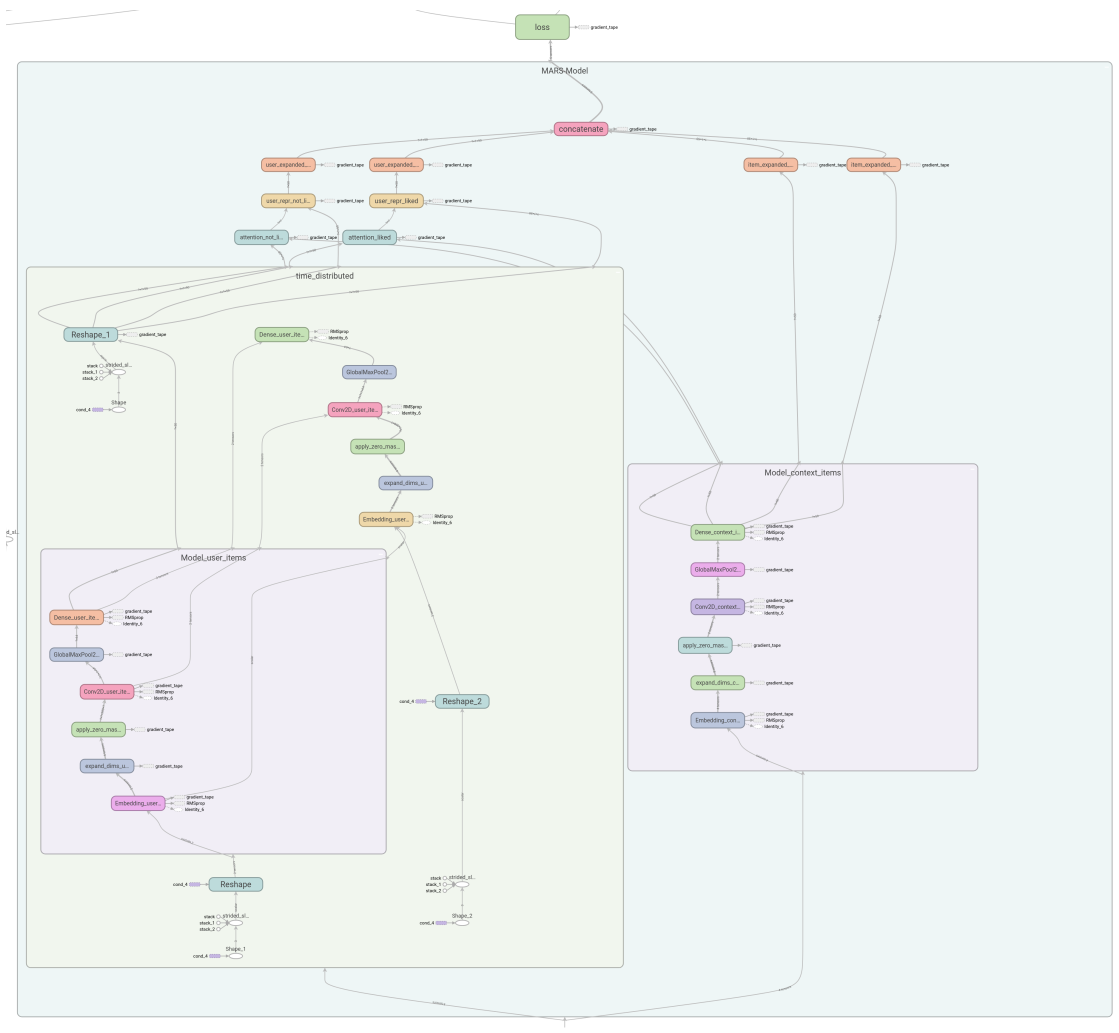

Figure 3: *Tensorboard* graph of MARS implementation

Inference Model Implementation The model for inference is different to the model for training insofar, as they have different requirements in terms of different inputs and outputs as well as there is no need for backpropagation. For inference the CNN to compute the item representation is cut out of the whole model to separately compute the representation for all items taken into consideration for the final ranking. This representations can be cached and reused for all users for which a recommendation should be predicted. This drives down latency and compute requirements for inference significantly. To efficiently predict scores for all item candidates, the memory component, inclusive CNN for the viewed/liked items, is cut out and wrapped in a new model which computes the scores for all candidate items in parallel. For this endeavor the *einsum* operator in *Tensorflow* is used as it allows for very compact and expressive code, shown in listing 1

```python
import tensorflow as tf
...
# each batch entry corresponds to the items history of one user
user_history_batch, items_repr_batch = inputs
# compute the memory component
user_memory_batch = self.user_memory_model(user_history_batch)

# compute the scores for each item with the memory components of each user
scores = tf.einsum("ijk,ilk->ilj", user_memory_batch, items_repr_batch)
# compute the attention for each item to the user memory components for
# all users
attention = tf.nn.softmax(scores, axis=-1)
# multiply the the embedding vector of each user memory component with the
# corresponding item attention and sum the embedding vectors of all memory
# components together to get the deep adaptive user representation
user_repr = tf.einsum("ijk,ilj->ilk", user_memory_batch, attention)
# compute the scores for each item with the deep adaptive user representation
# generated individually for each item
all_scores = tf.einsum("ijk,ijk->ij", user_repr, items_repr_batch)
return all_scores
```

Listing 1: Inference model using *tf.einsum()* operator

To compute the scores even more efficiently, the CNN to compute the item representations for the memory component can also be cut out to compute the representations separately and utilize caching. This is not done yet but will be considered for in-production deployment.

Hurdles

View History and Document Length Tensors impose the requirement, that data within a dimension in the tensor have to be the same length, so all view histories in a batch have to be zero padded to match the longest sequence as well as all documents have to be zero padded to match the biggest document. The view/like history of some users is quite long and differ much from the mean as well as some documents have much more words as the average. So the required memory of different batch sizes don't scale linearly and can differ greatly. If it happens, that a very long user history and a very big document appears in a batch, a batch becomes very large and exceed quickly the amount of available video memory. To this end, two different techniques are utilized. One possibility is to limit the maximum history length and document size. This is very effective and easy to accomplish but incur a loss of information. The other possibility is to incorporate *Tensorflow Ragged Tensors* which are essentially the tensor equivalent of nested lists. Both techniques were implemented but for the experiments only the first technique is employed as some *tensorflow operators* and *Keras layers* suffer from performance issues when used with *Ragged Tensors*, have severe bugs or aren't implemented at all. Explaining the *Ragged Tensor* implementation and it's issues would go beyond the scope, so it is covered in future work.

Unbalanced Replica Batches The model training is distributed via *Tensorflows MirroredStrategy()* which uses data parallelism. For each available GPU a copy of the model, named *replica*, is executed on that specific GPU, where the global batch is evenly partitioned among the replicas. So each *replica* has its own sub-batch. Although each batch consists of the same amount of samples, the length of the users histories and the size of the documents per sub-batch differ. For this reason the different length histories and documents makes the computational and memory costs of batches among model replicas unbalanced. This introduce inefficient distributed training/computation because the *replicas* have to synchronize at the end of each step. Thus the GPU utilization is not optimal, because *replicas* with less data idle, while waiting for *replicas* with heavier load to finish.

Padding & Masking Working with sequences of different length (history and documents) when using normal, non *Ragged Tensors* requires the use of padding to align the sequences to the same length. This padding also rise the need to mask this padding for different layers. The difficulty arise when some layers like *Conv2D* are don't meant for the use of masks or the padding has to be masked for custom (*Lambda*) layers. Higher dimensional sizes at some stages in the network, e.g. `tensor_shape = [batch, documents, words, embeddings, channels]`, also makes things more complicate .

3.3 Experiments

To verify the implementation, for each dataset of the paper, trainings with batch sizes of [16, 32, 64, 128, 256, 512] were performed. It is assumed that the optimal learning rate roughly scales linearly with the batch size and is scaled down from the original papers 0.001 for batch size 512 with `lr = 0.001 * (batch_size / 512)`. Training with different batch sizes were performed to investigate any regularizing effect of the batch size and its effect to the model performance. This was necessary since even with a history length cut down to a maximum of 50 and a maximum document length of 500, the original papers optimal batch size of 512 can't be applied for all datasets with the available hardware. It was possible to train the model on the *Yahoo! Movies* dataset with a maximum batch size of 512 but on the *Amazon Video Games* dataset the maximum achievable batch size was 256. Because of high computation time, the training on *Amazon Movies and TV* was just performed with batch size 16 for 54 epochs. For the used data sampling method, with practically unlimited samples, (Rendle u. a., 2009) suggests that running a training over as many samples as positive feedbacks exists is sufficient for convergence when the dataset is large (~300 million in their case). As the three datasets are not very large, it is choosen to run 20 times the number of positive feedbacks. To track the progress more fine-grained a epoch constitutes of a tenth of positive feedbacks, so the training runs for 200 epochs.

3.4 Results

Figure 4 shows the MAP (Mean Average Precision) score of all three datasets for all conducted training runs. On the top you see all runs for the *Yahoo! Movies* dataset which reaches the peak performance of 0.136 after 6 epochs with a batch size of 32 (green line). Notably all batch sizes performed very well already after the first epoch and are relative close together and start to decrease between the 5th and 10th epoch. On the bottom you see all runs on the *Amazon Video Games* and *Amazon Movies and TV* datasets which are also close together and very difficult to distinguish. They show a gain in performance over the first epochs with a peak at the 26th run of batch size 128 with 5.2655×10^{-3} for *Amazon Video Games* and 2.8372×10^{-3} at step 37 for *Amazon Movies and TV*.

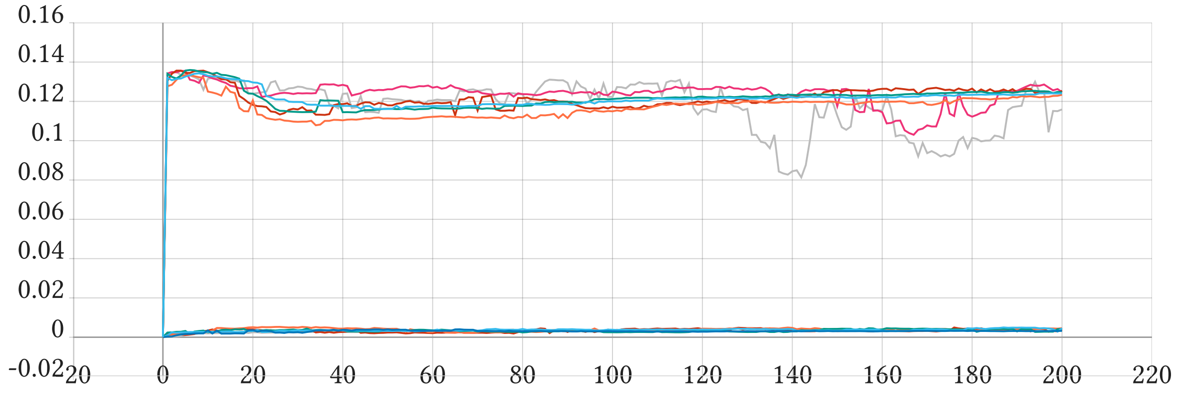

Figure 4: *Tensorboard* scalar graph of MARS MAP@500 metric

Figure 5 shows the Recall for all three datasets. The graph looks similar to the MAP graph with all *Yahoo! Movies* runs on top and the runs of the *Amazon* datasets at the bottom. The best performance on the *Yahoo! Movies* dataset is represented by the red line for the 128 batch size run with a maximum recall of 0.4449 at step 177. The top performance on *Amazon Video Games* is reached at step 145 with a recall of 0.038 44 by the run with batch size 16 shown as the blue line. On the *Amazon Movies and TV* dataset, the peak score is reached at step 28 with 0.0227.

Figure 6 shows the loss of all training runs. You can see three clusters of lines, the upper cluster of lines (brown (BS 16), pink(BS 32), gray(BS 64), dark(BS 128) and light blue(BS 256)) are the runs on the *Amazon Video Games* dataset. The lower cluster are the trainings on the *Yahoo! Movies* dataset and the orange line between them is the single run on the *Amazon Movies and TV* dataset. The lines overlap at some points but are clearly separated and dont't cross each other until they get very close. For the *Yahoo! Movies* trainings the different batch sizes are more close together then for *Amazon Video Games* but it applies to both datasets that the smaller batch sizes have a higher loss and the bigger batch sizes have a lower loss ordered from BS 16 to 512.

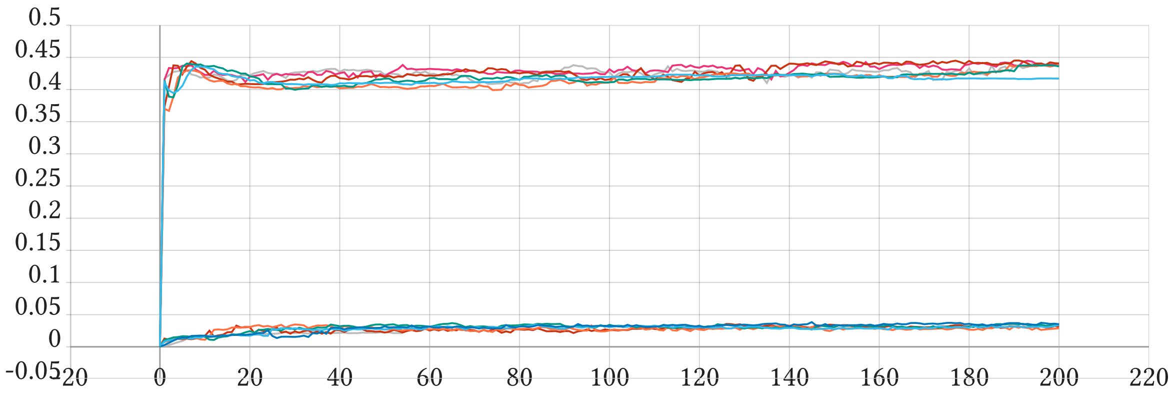

Figure 5: *Tensorboard* scalar graph of MARS Recall@50 metric

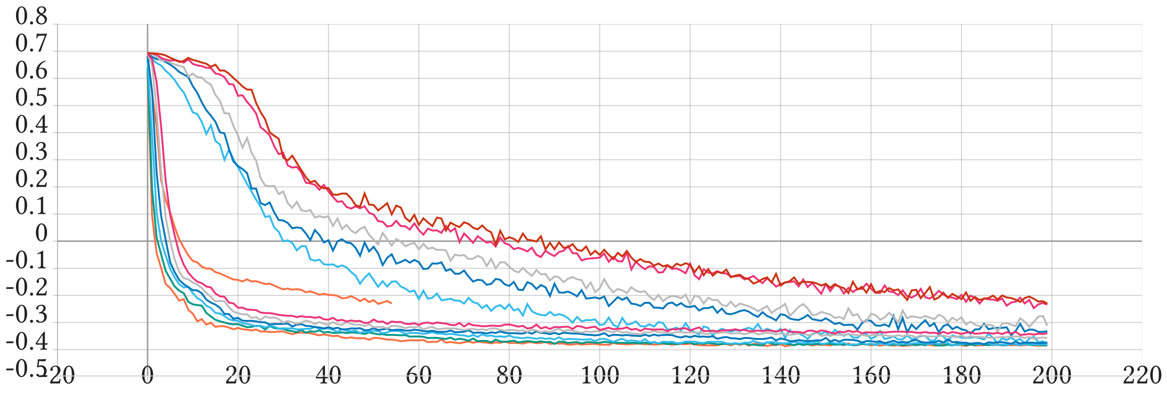

Figure 6: *Tensorboard* scalar graph of MARS epoch loss metric

3.5 Discussion

The results on the *Yahoo! Movies* dataset shows that the implementation basically works. The result on the MAP metric are worse then the original papers results (0.136 to 0.1692) but the recall metric shows better results then the original papers numbers (0.4449 to 0.3230). The score for *Amazon Video Games* for MAP (5.2655×10^{-3} to 0.0934) and recall ($0.038\,44$ to 0.1337) differ greatly from the original papers numbers and are much worse. Similarly the scores for *Amazon Movies and TV* are much worse then the original papers scores. The correspondence in the drop of the loss value and the rise of the MAP and recall metric shows that the loss function is suitable to train a model for the recommendation task. Also the closeness of the trainings of different batch sizes shows there is a minor affect of the batch size to the model performance. Further, the graph of the metrics shows, the model reaches good results just after a few epochs, which supports the assumption of the effectiveness of the data sampling method. So the training on bigger datasets can be done safely with smaller batch sizes on less powerful hardware with just a few epochs of training. This maybe lead to a more cost effective model in contrast to more complex and deep models which will require more resources for training and may lead to fewer model updates on new data due to training time. In addition the used data sampling method may improve the training of other models.

It is presumed that mainly differences in the data preprocessing to the original paper are responsible for the deviant results. Also the assumptions of section 3.2.1 made for missing information and the technically necessary cut of the user history and document length will have an impact on the results. As already discussed in section 2.1.4 the processed data partly differ to a great extend and changes in the training and test data have a very big influence on the models performance. Especially the fact that the *Amazon Video Games* dataset consists of roughly 67 % item descriptions which contain just special characters and no words, which are finally replaced with padding when you stick to the papers described preprocessing, underlines this assumption. The differences in results between the *Yahoo* and *Amazon* datasets can probably be explained by two factors. 1) The *Yahoo* dataset contains descriptions consisting only of flowing text without special characters and the *Amazon* item descriptions contain many special characters and *HTML* code. 2) The density (proportion of samples to possible item/user interactions) of the *Yahoo* dataset with 0.24 % to 0.037 % for *Amazon Video Games* and 0.0128 % for *Amazon Movies and TV* is much higher. The low density of the *Amazon* datasets makes the training for a model much harder and more sensitive to variations in data.

3.5.1 Proposed Improvements

The following represents a list of proposed improvements, which may enhance the capabilities of the model architecture:

Pretrained Embeddings Initialize the word embeddings with pretrained ones on large text corpora.

Transfer learning for item representations Use models pretrained on large text corpora for the item representation.

RNN/Transformer to Capture Temporal User Features Generate the user representation from the memory component by first weighting the individual memory components with the attention vector, like the original architecture and instead of summing them up, use an RNN, like LSTM or GRU, to capture temporal features from the users history and use it as the user representation. Alternatively use a Transformer based model instead of the RNN. Here one have to try if the attention mechanism is still useful or can be completely replaced with the RNN. It is to mention, that in this case, the liked item by the user have to be sorted by time and the dataset must provide a sorted history or timestamps.

Replace representation CNN Replace the CNN for item representation generation with RNN or transformer based models to better model the content.

Bigger NN's The authors just do a grid search for the embedding layer size and the final dense output layer which also makes for the item/user vector representation size. As the CNN and dense output network have just one hidden layer, the authors didn't leverage the power of deeper networks. Thus deeper networks may extract richer features and perform better. As the model capacity rises, additional regularization techniques like dropout can compensate this to prevent overfitting. Alternatively a bigger kernel size may capture more dependencies in the content or kernels of different sizes capture more diverse features.

4 Conclusion & Outlook

4.1 Conclusion

In section 2 the used datasets are described, with deviations to the original paper after preprocessing stressed in 2.1.4. The original paper were sketched in section 3.1. The experimental design of this work including model implementation details, preprocessing steps and assumptions made, because of missing information, are presented in section 3.2. To this point it can already be seen how difficult it is to reproduce a paper for which the author don't have made their source code publicly available. When it comes to that the authors are not available to discuss their work or answer questions about omitted information it gets even more difficult.

The results of section 3.4 shows that the implementation works on one of the three datasets but fail on the other two, probably by deviating preprocessing to the original paper. The results also shows that the model needs just a few epochs of training to reach usable performance and small batch sizes perform equally to bigger ones. This indicates, that the model can be trained comparatively cheap in terms of hardware and training time requirements. The fast convergence also suggests that the data sampling method is effectively to speed up training and may benefit the training of other models. This also indicates that training on the much bigger news recommendation dataset should work without implementing advanced techniques like *Ragged Tensors* when utilizing small batch sizes. Also to train the model on the news recommendation dataset don't need much adaptations as long as just the text content is used for item representation.

4.2 Outlook

In the follow up master thesis, the implemented model will be tested with the news recommendation dataset and also online A/B tests will be carried out. Additionally some of the proposed improvements of section 3.5.1 will be evaluated. Along with the model of this work, other neural network architectures and techniques will be looked at. Also the rich metadata of the news recommendation dataset will be incorporated to investigate their effect on model performance. Finally, promising components of the evaluated methods will be picked out to derive a new architecture to possible surpass the examined methods on their own.

One key takeaway is, in further work to focus on papers which deliver source code of their experiments or get in touch with the authors before starting to reproduce their work. Otherwise one can't rely upon others work and have to question their results or methodology or get distracted by implementation details at the expense of own developments.

Acknowledgment

I would like to thank Dr. Christoph Mayer from *SCHICKLER Unternehmensberatung GmbH* for providing the great dataset and Prof. Dr. Kai von Luck for supervising and supporting this project and providing valuable suggestions.

References

[Lange 2020] Lange, Timo: Building a data pipeline for News Recommendation with Deep Learning. URL https://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2020-proj/lange.pdf, Juni 2020. – Forschungsbericht. – 27 S

[Rendle u. a. 2009] Rendle, Steffen ; Freudenthaler, Christoph ; Gantner, Zeno ; Schmidt-Thieme, Lars: BPR: Bayesian Personalized Ranking from Implicit Feedback. (2009), S. 10

[Zheng u. a. 2019] Zheng, Lei ; Lu, Chun-Ta ; He, Lifang ; Xie, Sihong ; He, Huang ; Li, Chaozhuo ; Noroozi, Vahid ; Dong, Bowen ; Yu, Philip S.: MARS: Memory Attention-Aware Recommender System. In: *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, Oktober 2019, S. 11–20