Building a data pipeline for News Recommendation with Deep Learning

Grundprojekt 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

June 17, 2020

In this project report an end-to-end data pipeline for deep learning based news recommendation will be described. Purpose of this document is the description of an end-to-end pipeline to generate news recommendations for users and layout an infrastructure for further experiments and implementations of deep learning recommendation methods. The recommendation methods described in this report are based on the work of Okura u. a. (2017) and adapted to a German news text corpus. In the course of this document the dataset from the *Deutsche Presse-Agentur (dpa)* will be analyzed and the implementation of the paper will be presented. Furthermore an overview of the pipeline architecture and *Tensorflow Extended* as a future platform for the data pipeline is given and a review of tools for the pipeline is done. Experiments are conducted, which shows that the pipeline is suitable to process the data and do deep learning based recommendations but shows also that further profiling of the recommendation system is needed to optimize resource utilization.

Keywords – Recommender System, Deep Learning, Natural Language Processing, Data Pipeline, KDD

1. Introduction

Recommender systems (RS) are software tools which provide recommendations for users, and thus support the user in their (online) decision making. The goal of RS is to provide easy accessible and high quality recommendations for a community of users. In times of *Big Data* and an ever growing flood of data a user has to deal with, RS helps the users to cope with an *information overload*. On the business side, RS can help to gain user satisfaction with the provided services and increase revenue of content providers.

One aspect of RS are news recommendations, which will be the focus of this work with a special focal point on Deep Learning (DL) RS. Since the author of this paper has the opportunity to work with a real-world news dataset from the *Deutsche Presse-Agentur (dpa)*, a German press agency, the sections of this document revolves around this dataset. With over 200 000 articles of high linguistic quality and rich metadata, to the best of the authors knowledge, this is a unique kind of dataset which can't be found in similar quality publicly.

Goal of this work is to evaluate how to construct an end-to-end data pipeline to work with this dataset and doing DL based recommendations. To this end, an implementation of the Okura u. a. (2017) paper will be done and a corresponding data pipeline will be introduced to make recommendations for news articles. The methods used in Okura u. a. (2017) will be adapted to the news dataset from the dpa. As the news articles contain a substantial amount of metadata and the quantity of articles in the dataset, with more data to come in mind, is quite large, a thoughtful architected data processing pipeline is needed. To implement the mentioned paper and to develop such a data pipeline, beside the analysis of the dataset, possible tools have to be examined. Finally experiments have to be conducted to verify the proposed architecture and implementation is suitable for the dataset and recommendation methods in the Okura u. a. (2017) paper.

The structure of this document is as follows: In the course of this text, first the news dataset with rich metadata from the dpa will be presented and analyzed. Hereafter the Okura u. a. (2017) paper will be reviewed including the adaption to the dpa dataset, needed preprocessing, implementation and discussion of limits of the methods in the paper and proposed improvements. This is followed by the pipeline architecture with current and proposed architecture, introduction to the *Tensorflow Extended (TFX)* platform and a tool review. Subsequent to this the conducted experiments are presented. Finally this document is closed with a summary and an outlook.

2. Dataset Analysis

The methods described in the following sections are applied and adapted to the dataset from the dpa described below. The dataset consists of over 200 000 German news articles, which are available in the structured *NewsML-G2* IPTC (2018b) data format and contain a lot of metadata. The texts are written from journalists and of high linguistic quality. The content is available

as plain text and structured in *HTML* format. There is also a short summary for some of the articles. The metadata consists of fields like author, date, location, keywords and events. Also the articles are annotated with *IPTC media topics* IPTC (2018a). This is a taxonomy with about 1100 topics which is structured as a tree with 17 top-levels and a depth of 5 levels. The exact structure of the data is discussed in the next section.

2.1. Data Structure & Applicability

The explanation of the structure is partly based on the work of Nitsche und Halbritter (2019), which do a deep analysis of the dataset. So the description here focuses on the key aspects of the structure.

Table 1 shows the data fields of the article documents. The *Unique* and *% Not Null* are mainly taken from Nitsche und Halbritter (2019) and completed where necessary. Beside the completion of data, the table show additional values for minimum, average, maximum occurrences and standard deviation. When the type of the field is a *list*, the data relates to frequencies of list entries. If the field type is *str* the additional metrics relates to number of words. For some fields there are no additional metrics as this don't make sense for types like *datetime*, *int* or *str* with single unique identifiers. The categorials are a fixed number of symbols represented as string identifiers. These string identifiers have an association to German translations, which can be mapped via a table.

The most important field is *contentplain*, which contains the article plain text and is most important for recommending articles. The *contenthtml* field contains, structured as *HTML*, the text and additional, redundant to other fields, *headline*, *slugline*, *keywords*, *genre* and *publication date* among others. This explains the much higher word count in contrast to the plain text.

Very useful to fine grain associate articles to different topics are the *keywords*, *slugline* and *medtop*. As they have relative high non null values, these fields can be used for a big part of the dataset. As already mentioned the *medtop* field contain a list of *IPTC media topics* taxonomy identifier IPTC (2018a) The fields *keywords* and *slugline* are lists of strings which can be customly defined by the editor. The *keywords* field are, as expected, keywords and the *slugline* describes the basic content of the article in a few words.

Also the *subject*, *genre* and *category* are very useful to categorize the articles on a more coarse-grained level (due to their relative few unique values). Especially as *genre* and *category* are present for every article, so these fields can be used for the whole dataset.

Useful for summarisation and to grasp the essence of the article are the *headline* and *description* fields. A drawback of the *headline* is the very short text but on the plus side is its availability for all articles. The *description* has much more words and is much more useful for this task but unfortunately the overall occurrence is relatively low and would shrink the application to roughly a fifth of the whole dataset, which makes it hard for a model to learn.

The *country*, *area*, *location* and *poi* (point of interest) also seem to be a applicable information to improve recommendations. The *creator* and *contributor* information could be consulted, to some extend, to distinguish different writing styles or topic preferences.

No.	Field name	Type	Unique	% Not Null	Entries/Words			
					min	avg	max	Std.
1	guid	str	214725	100.00%				
2	keyword	list[str]	15555	68.40%	1	2.01	11	0.88
3	headline	str	165107	100.00%	1.0	8.77	64.0	3.35
4	description	str	35697	18.46%	0	31.43	138	6.24
5	contentplain	str	186387	99.90%	3	294.60	3849	244.08
6	contenthtml	str	214491	100.00%	76	440.53	9853	280.82
7	slugline	list[str]	59247	100.00%	1	4.18	10	1.30
8	genre	list[category]	30	100.00%	2	2.00	2	0.00
9	subject	list[category]	129	54.31%	1	1.14	6	0.39
10	medtop	list[category]	230	75.32%	1	1.31	7	0.61
11	category	list[category]	6	100.00%	2	2.00	2	0.00
12	area	str	21	22.09%				
13	country	str	202	80.77%				
14	poi	list[str]	39624	39.54%	2	4.05	49	0.86
15	ednotes	list[str]	316937	100.00%	1	4.79	72	2.66
16	sent	datetime	200240	100.00%				
17	version	str	146	100.00%				
18	language	str	1	100.00%				
19	creator	list[str]	827	99.84%	1	1.00	1	0.00
20	contributor	list[str]	35721	99.65%	1	1.69	17	0.80
21	newspublisher	list[category]	32	100.00%	1	3.64	20	1.64
22	urgency	int	4	100.00%				
23	location	list[str]	4993	65.47%	1	1.16	4	0.38

Table 1: Metadata fields of the news articles with different metrics

3. Okura u. a. (2017) Recommendation Methods Review

This section will lead through the recommendation methods that are presented in the paper and to be tested with the described dataset. On a rough overview the recommendation system consists of three main parts beside data preprocessing:

1. **Denoising Autoencoder**
   The autoencoder is used to encode the articles in the form of embeddings.

2. **Deep User Representation Model**
   This Model is used to generate a embedding for the users from the history of viewed articles.

3. **Article Matching**
   The actual article recommendation, which is a simple dot product between the user embedding and article embedding vector, which results in a score for each article for the specific user.

3.1. Methods

Autoencoder To generate the article embeddings, the authors of the paper used a variant of denoising autoencoder from the paper Okura u. a. (2016). The input of the autoencoder are binary hot encoded vectors of size 10 000. Each position of the vector corresponds to a word in a vocabulary. As noise the input vector is stochastically masked with zeros, with a 0.3 corruption rate. The actual input to the autoencoder are triplets (x0, x1, x2) of articles which consists of x1 the article to encode, x2 a article of the same category as x1 and x3 a article of a different category as x1. The data generation is explained in section 3.3.2 in detail. To train the autoencoder the following objective function is used:

$$\begin{aligned} L_T(h0, h1, h2) &= log(1 + exp(h0^T h2 - h0^T h1)) \\ Loss &= L_R(yn, x_N) + \alpha L_T(h0, h1, h2) \end{aligned} \tag{1}$$

Where L_T is the penalty function for article similarity, α a hyperparameter for balancing and L_R is the element-wise cross entropy function. Figure 1 illustrates this method.

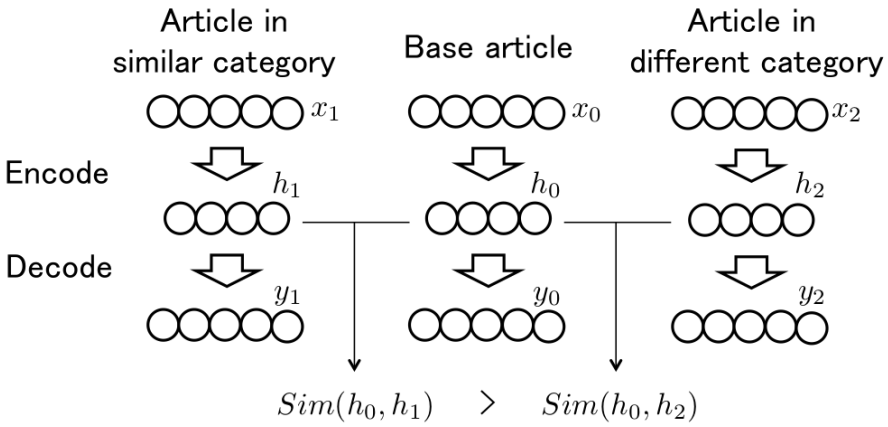

Figure 1: Encoder for triplets of articles Okura u. a. (2017)

Deep User Representation Model For the user representation Okura u. a. (2017) compared a simple Recurrent Neural Network (RNN), a Long-short Term Memory (LSTM) and a Gated

Recurrent Unit (GRU). The input for the model is the users viewed article history as a sequence of article embeddings. The objective function for all models is the same and as follows:

$$\sum s_t^u \sum_{\substack{p+\in P_+ \\ p-\in P_-}} \frac{log(\sigma(R(u_t, s^u_{t,p_+}) - R(u_t, s^u_{t,p_-}) + B(p_+, p_-)))}{|P_+||P_-|} \tag{2}$$

Where s_t^u are the sessions of user u at time t and $p+ \in P_+$ and $p- \in P_-$ are the viewed and not viewed articles at this session that are displayed to the user. $R(.,.)$ denotes the relevance function of an article and $B(.,.)$ is a bias term and a parameter to be learned by the model which is not described in detail by the authors. This $B(.,.)$ term describes the bias, which results from the position of an article when the articles are displayed in a vertically arranged list to the users. The relevance term $R(.,.)$ is defined as a simple inner product between the user and article embedding $R(u_t, a) = u_t^T a$.

While doing experiments they find that the simple RNN is not suitable when the input sequence is to long due to vanishing and exploding gradients. The LSTM model occasionally failed when not using gradient clipping. Just the GRU model did not cause problems with vanishing and exploding gradients and yielded the best recommendation performance.

Article Matching The actual article recommendation is done via simple dot product between the user embedding and all article embeddings. This yields a relevance score for every article for the current user which can be used for relevance ranking. Also a article-article dot product is used for de-duplication, so the user don't get recommendations for different but similar articles.

3.2. Adaption to the dpa Dataset

As the dpa dataset and user data is different to the data the methods where originally constructed for, some adaptions has to be made. The original paper constructed the system to make recommendations for end users. The articles in the dpa dataset are not meant to be displayed directly to end users. The articles are offered to editorial offices of newspapers and other companies, which choose to publish the article as it is or rewrite it.

3.2.1. Session

One obvious difference are the sessions of end users and editorial offices. In the original paper the authors defined a session to be the articles that are shown in the *yahoo news app* to the users, which are the first 20 displayed articles arranged in a vertically list. P are all displayed articles, respective their position in the list. $P+$ are all viewed articles, in this case there is just one article viewed per session, namely the article clicked by the user. $P-$ are all not viewed/clicked article in the session.

There are three main differences in sessions defied for the dpa data to the Okura u. a. (2017) paper:

1. **Session Size**
   As the articles are not shown to a single user but instead to a whole editorial office, in this case one session is defined to be all articles published on one day. As the dataset spans a period of approximately one year with about 214 000 articles, there are around 586 articles on average per day. So the session size is much larger then for end users.

2. **Viewed Articles**
   Since there are several editors which choose various articles to be published, the count of viewed articles is more then one and can possibly range from zero to all articles on that day.

3. **User History Length**
   The history length of the users can grow into many thousands, since one editorial office may pick tens or hundreds of articles per day. This is also the biggest obstacle or challenge, because the chosen neural nets struggle more the longer the sequences get. As previously described the simple RNN and even the LSTM have difficulties to deal with histories even by individual users.

3.2.2. Article Categories

In the original paper an article seems to be assigned to a single category. The dpa dataset provides detailed metadata with possible several categories, keywords and mediatopics per article. So the choice which articles are similar and which are not is not just to look which articles are in the same or different category. With the dpa dataset the mentioned metadata fields must be put into account with possibly a range of choices.

3.3. Data Preprocessing

In the following section the data preprocessing steps are described which consists mainly of the three phases of *parsing*, *training data generation* and the generation of *artificial user data* as of the time of writing real user data is not jet available. The expected user data will be gathered thru crawling news sites for published articles and compare these articles to the articles in the dataset. As the published articles may be rewritten versions of the original article the text can not be compared one by one. So there will be a similarity measurement with a threshold to decide whether the published article corresponds to an article of the dataset. With this method it can be measured which articles where relevant to the corresponding news publisher.

3.3.1. Parsing

The dataset consits of single XML files per article in the *NewsML-G2* IPTC (2018b) data format. This format is not suitable as input to the ML framework used to implement the RS. In a previous work Nitsche und Halbritter (2019) developed a parser, which generates JSON documents for

the articles. These JSON documents are stored in a MongoDB, which is the basis for analyzing and exporting the data to other formats. For performance reasons in the model training phase and efficient storage, the data is exported to parquet files. The parquet files are read with apache arrow and converted to *pandas Dataframes* for further processing.

3.3.2. Train Data Generation

There are two phases of training data generation. The first is to generate the input for the *autoencoder*, whose output is also a part of the input in the next phase. The second phase is the input for the *User Representation Model*.

Autoencoder Training Data In the first step of training data generation it is necessary to get triplets of articles as input for the autoencoder. As described in 3.1 it is necessary to compute a similar and a dissimilar article for a given article to encode. Before the triplets are generated, the articles must be converted to a format which is easier to process then text strings. For this purpose, the *Keras* build-in preprocessing module for text is used. At first the texts gets tokenized, where the words will be converted to lower case and splitted by space. The output of this tokinization is a binary word vector, where each position corresponds to a word index. The vector size is cut to 10 000 words like described in the paper. The top most 10 000 words in the whole corpus are used and the rest are discarded. To generate the triplets, first the metadata fields are chosen which shall serve as labels and a dictionary with a mapping from the article *guid* to a set with all values form the chosen metadata fields is generated. Next the article with the highest and lowest similarity is chosen. When there are several articles with the same similarity the first match is taken. As metric to measure the similarity of articles the *Jaccard Index*, also known as *Intersection over Union*, of the sets of labels is used. The *Jaccard Index* is computed by dividing the intersection of two label sets by the union of the label sets. Equation 3 shows how the index is build, with set A of labels from article a and set B of labels from article b. In the final step zero masking noise is applied to the word vector as described in 3.1.

$$\begin{aligned} J(A,B) &= \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \\ 0 &\leq J(A,B) \leq 1 \end{aligned} \tag{3}$$

An enhancement to be more general would be to, instead of first matching min and max similar articles, return a list of all max and all min similar articles, if several present. Then shuffle the min and max similarity articles corresponding to the trained article each training epoch. This would also avoid, that just a little bunch of articles are the min similarity articles for every other article (e.g. a single article is labeled with categories distinct to all other articles categories). Currently this enhancement is not implemented and the article triplets are computed beforehand training and written to a file. Another possible improvement would be to remove stop words,

punctuation marks and use stemming, which can be easy accomplished with filtering by a word list. These techniques are not part of the implementation of the Okura u. a. (2017) paper and will be tested in further work, where different RS methods will be tested against each other.

User Representation Model Training Data To feed the input into the model a *generator* that yields batches of training data, which runs in parallel on the CPU, is used. In this case a implementation of the *keras.utils.Sequence* class was used to guarantee that the ordering of batches will be kept and every input per epoch is used just once. The generator made it possible to augment the data while training. This was necessary because a batch (x, y) with input x and label y has samples with a single size of $\sim 1.3\,\mathrm{GB}$, which would result in to much data to compute beforehand and store it. This comes due to the fact that *numpy arrays* are multi dimensional arrays that have a fixed size for every dimension. So each dimension must have the size of the maximum possible entries for this dimension. Where are less entries then the dimensions size, the non-existent entries have to be zero padded. x and y are *numpy arrays* with dimensions $[bs, hl, es]$ and $[bs, sn, vn, sl, es]$. Where bs is the batch size, hl is the maximum user history length of a user in this batch, es is the embedding size, sn is the maximum number of sessions a user has in this batch, vn are the two bags of viewed and not viewed articles and sl is the maximum length of a session a user has.

To generate the batches the generator is initialized with a dictionary which maps the user id to a list of article *guids* that represent the view history and a list of sessions with the viewed and not viewed article *guids* of this session. Another input for the generator is a dictionary which maps the article *guids* to the corresponding article embedding, which is produced beforehand with the trained autoencoder. At each batch generation first the maximum sizes of the user sessions and session lengths are computed. After the *numpy array* dimensions are known the article *guids* of the user history and sessions are mapped to the corresponding embedding vectors and the *numpy array* is filled with the values.

It's also important to check how big the actual batch size will get, since if the chosen batch size is not a multiple of the dataset size the last batch on an epoch will be smaller then the chosen batch size. Since *Keras* allows different batch sizes during training the whole dataset can be used, regardless if its not exactly divisible by the batch size. Another subtle point is to pay attention to the datatype a *numpy array* is initialized with. E.g. the standard *numpy float* type is $64\,\mathrm{bit}$ and *Tensorflow* as *Keras* backend uses $32\,\mathrm{bit}$ *float* as standard. If the *precision* of the types don't match, the batch may unnecessarily take up a huge amount of RAM and the types have to be converted internally which uses additional CPU resources.

3.3.3. Artificial User Data

Since there is no real user data available for this news dataset by now, artificial user data have to be created to start implementing the Okura u. a. (2017) paper with the adaptions, the corresponding data pipeline and do testing. As the generated user data will be replaced by real human user data by the time they are available, the artificial data have to be as similar to real

world data as possible. This includes the amount of users, the length of article view history, the size of session data and to some extend plausible viewed articles history to learn a distribution of user interest.

To model the interest of the artificial users, metadata fields are chosen which are used to model interest of the users, in this case *keywords*, *subjects* and *mediatopics*. Additionally for each field a fraction has to be provided which states how much values from this field are used per user. For each user a random selection from the metadata fields values are picked while taking the corresponding fraction into account. The fractions chosen are: *{keywords: 0.01, subjects: 0.1, mediatopics: 0.1}*. These values are discovered empirically to generate a history of sufficient length to be plausible but don't grow to a abnormal size. Now the view history of each user is make up of all articles tagged with these labels, sorted by publication date. As mentioned before in section 3.2.1, a session consists of all articles published on one day. The viewed articles are those ones which are contained in the users history and the not viewed those ones that are published and not present in the users history.

3.4. Recommendation System Implementation

In the following, implementation details for implementing the described paper will be shown. As mentioned before the models were implemented with *Keras* and *Tensorflow* as backend.

3.4.1. Models

Autoencoder The *denoising autoencoder* implementation consists of the input and output layer and one hidden layer. The hidden layer serves as encoder and the output layer as decoder. Input and output layer have a size of 10 000 units, which logically have to correspond to the size of the input token vector. The encoder layer have a size of 500 units as stated in the original paper. Both, the encoder and decoder layer are densely connected and use the *sigmoid* activation function also as stated in the paper. The training is done via *Keras 'multi_ gpu_ model'*, which allows to train the model on multiple GPUs. The model parameters are kept in main memory to relive the GPU RAM from additional stress. The optimizer used for training is standard *stochastic gradient descent*. There may be optimizer which converge faster like *Adam* or *Adadelta* but this is subject to hyperparameter tuning in further work.

User Representation RNN The user representation model is implemented as a simple RNN as the simplest version and simultaneously as more sophisticated LSTM and GRU networks. To be clear, these models don't work in conjunction but are for comparison with each other. Beside the input layer, the model consists of a *masking layer* and one *recurrent layer*.

The *masking layer* is necessary because each input batch consists of several users histories as samples with different lengths as input sequence for the *recurrent layer*. As the input are *numpy arrays*, which require that all sequences in one batch must have the same length, the sequences are zero padded to match in length. The zero padded entries should not be feed into

the *recurrent layer*, as they are not informative data and would harm the performance of the model. For this reason the *masking layer* masks this values, so they get skipped by the model.

The *recurrent layer* size match that of the encoder with 500 units. For the activation function *tanh* is used and a bias vector is utilized. The optimization is done via *stochastic gradient descent*.

3.4.2. Loss Function

Two loss functions have to be implemented since the *autoencoder* and the *User Representation RNN* both use non-standard loss resp. objective functions.

Autoencoder Loss Function Like described before, the *autoencoder* is trained with the loss function in equation 1. To compute the loss as seen in the equation, the embeddings $h0$, $h1$ and $h2$ for the articles $a0$, $a1$ and $a2$ have to be computed inside the loss function with the current model parameters of the current training step. Two things must be done that are not mentioned in the *Keras* documentation and hard to find out, since they seem to be rarely used and barely discussed in communities like *Stack Overflow*.

The first is to make a *python closure* function with the encoder model as parameter which return the actual loss function. Next, in the actual loss function, the layers of the encoder model have to be extracted to build up a graph of the layers (excluding the input layer) and evaluate it with the $a0$, $a1$ and $a2$ token vectors to get the embeddings $h0$, $h1$ and $h2$. Now the embeddings, generated with the current model parameters in this training step, can be used to calculate the similarity penalty which get added to the binary cross entropy loss for measuring the actual reconstruction error.

Second, *Keras* loss functions expects two parameters, where *y_ true* is the label for this data sample and *y_ pred* is the prediction generated by the model. The typical case and expected by *Keras* is, that there is one label per sample. If there are several labels per sample it gets more complicated, since one can pass just one *numpy array* as *y_ true* value to *Keras*. The solution to this problem is, that the *y_ true numpy array* can be of arbitrary dimension and size. So you can encode your labels in one *numyp array* and extract these labels inside the loss function as shown in listing 1.

```
1 import keras.backend as K
2 ...
3 def loss(y_true, y_pred):
4         h0 = K.reshape(y_true[:,0], (-1, K.int_shape(y_pred)[1]))
5         h1 = K.reshape(y_true[:,1], (-1, K.int_shape(y_pred)[1]))
6         h2 = K.reshape(y_true[:,2], (-1, K.int_shape(y_pred)[1]))
7         ...
```

Listing 1: Extract multiple labels from *y_ true* in *Keras* loss function

User Representation RNN Loss Function The *RNN* model is trained with the loss function described in equation 2. To implement the equation in pure python would be straightforward

but with *Keras* and *Tensorflow* you must stick to the API of the framework which describe a *dataflow DAG (Directed Acyclic Graph)* of operations. An advantage of this is, that *Keras* resp. *Tensorflow* as backend take care of the parallel execution of the loss function computation, which would also be executed on a GPU if available. A new feature in *Tensorflow 2.0* is *tf.function* with *Autograph* which converts a subset of Python code into a efficient *Tensorflow Graph*. [1] This would ease the implementation of the loss function but for this work the use of the new feature is out of scope since the code was written for *Tensorflow 1.x* and will not be ported to the newest version for now. A simplified *DAG* of the RNN loss function is shown in figure 2. A complete *DAG* of the loss function and detailed description can be found in appendix A.

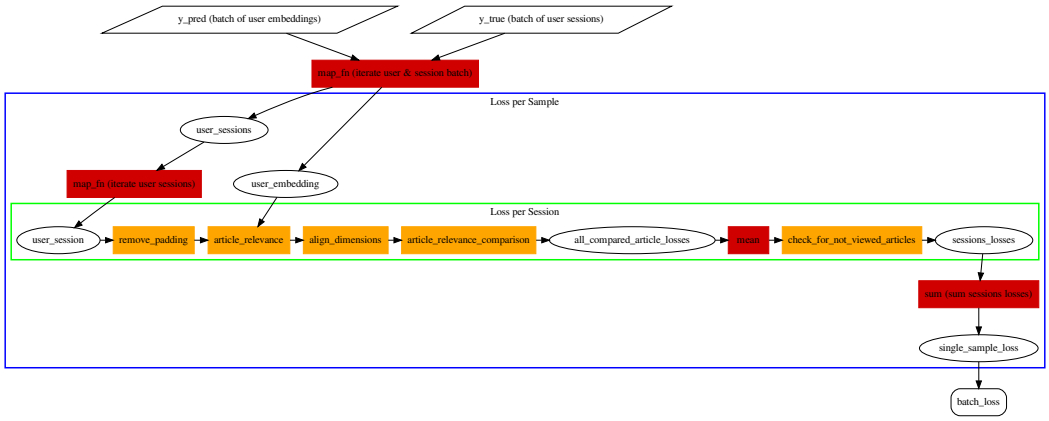

Figure 2: Computation graph of the adapted RNN loss function

The red rectangles indicate *Keras* operations and orange rectangles represent aggregated subgraphs. The rounded box *batch_loss* is the end node of the graph and the parallelograms mark inputs. Edges are tensors and represent the inputs and outputs of the operations and subgraphs. Round nodes are just named tensors, for a more descriptive graph. The blue and green rectangles mark the function which is applied by the map function to its input.

3.5. Limits of the Methods

3.5.1. Limits

There are some limits of the methods in Okura u. a. (2017). Two obvious are the static input size of the autoencoder, which may lead to a loss of information and makes padding necessary, and to not be able to use pretrained sentence or language models. Another dataset specific

[1] https://www.tensorflow.org/guide/function

limitation is the limited maximal sequence length of the RNN and even LSTM and GRU models with regard to the extraordinary user history length compared to usual end users.

3.5.2. Proposed Improvements

A small selection of improvements may be:

1. **RNN Autoencoder**
   Use an autoencoder which has no direct restriction in input length, e.g. an RNN autoencoder.

2. **Word embeddings**
   Use (pretrained) word embeddings instead of word tokens for the autoencoder.

3. **Deep User Representation Model "without" restricted sequence length**
   Use of an model which can handle big sequence length like Hierarchic Attention Networks (HAN) Yang u. a. (2016) etc.

4. **Use Metadata**
   Incorporate the rich metadata of the dpa dataset into the model.

5. **Collaborative Filtering**
   Combine the User Representation Model with an Neural Collaborative Filtering (NCF) He u. a. (2017).

4. Pipeline Architecture

In this section a data processing architecture will be proposed, adapted to specific requirements that the dataset and recommendation algorithms introduce.

4.1. Current and Proposed Architecture

The aim of this pipeline architecture is to provide a way to define different ML pipelines with the flexibility to maintain different workflows to satisfy the needs of multiple ML methods. Primarily the architecture should provide a flexible and as simple way to experiment with different recommendation methods and paper implementations. Some requirements are to keep track of the data version, preprocessing code, hyperparameters, model version and logs which are used in an experiment, to analyze and compare the outcomes easily. Figure 3 shows the current data processing. At the current state each step in the data processing is triggered manually and the components are configured separately with no automated orchestration of the different tasks. For initial prototyping, to create a single Proof of Concept (POC), where the main goal is to try out different things and be flexible this is sufficient. For the next phase

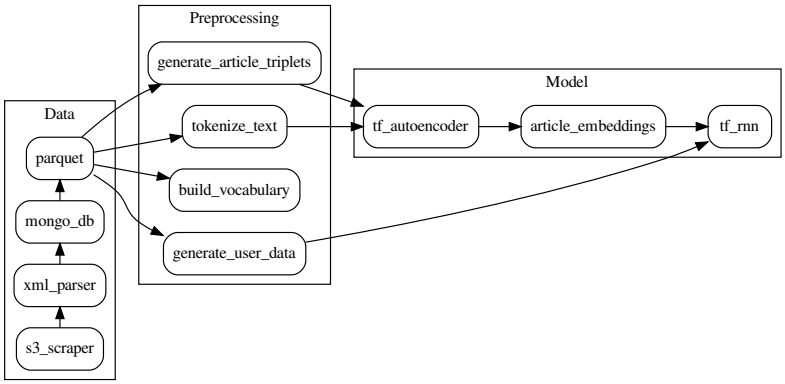

Figure 3: Current data processing state

(implement different papers, compare the methods performance and hyperparameter tuning) a more sophisticated architecture is needed.

To ease the deployment and tool selection it comes handy that with *Tensorflow Extended (TFX)* a platform was open sourced by google which perfectly match the mentioned requirements of task- and data-aware pipelines. There are some limits in the use of TFX which includes:

1. **Datasource Format**
   Supported data formats are just *CSV* files, *TFRecord* files with *TFExample* data format, and results of *BigQuery* queries.

2. **ML Framework**
   Limited to the *tensorflow* framework for models.

3. **tf.estimators instead of *keras* models**
   TFX just fully support models written using the *tf.estimators* API.

This deficiencies are acceptable at this time, since to rewrite the *keras* data generators to be compatible with TFX should be easy and as *tensorflow* is anyway the main ML framework used, the restriction to *tensorflow* as ML framework is tolerable. A workaround to use *keras* models with TFX instead of *tensorflow estimators* is by using the *tf.keras.estimator.model_to_estimator* function to convert *keras* models to estimators.

If some papers provide a implementation, for example in *PyTorch*, it is to consider to move to a more flexible pipeline architecture or evaluate if an re-implementation in *tensorflow* is a worthy strategy. In the case of shifting to another platform/architecture it is convenient that some parts of TFX can be used independently form TFX and some parts of TFX are anyway

from other open source projects like *Apache Beam*, *Apache Airflow* and *Kubernetes Pipelines* and of course can still be used within another platform/architecture.

4.2. Tensorflow Extended

In the following the architecture of TFX is briefly presented, with an eye on integration of the previously manually orchestrated tasks/dataflow. Essentially TFX is made up of three mayor building blocks, the *Components* of which data pipelines are build of, a *metadata store* which stores information about the data which every component in the pipeline generates (the actual data is stored elsewhere) and an orchestration tool to plug all tasks resp. components together into a pipeline. The technologies underlying TFX are *Apache Beam* for data processing with interchangeable backends like *Apache Flink* or *Apache Spark*, *Tensorflow* for model training and *Apache Airflow* or *Kubeflow* for orchestration. The *Metadata Store* has a pluggable backend like *SQLite*, *MySQL* or other databases with SQL capabilities. The store enables one to trace back the execution of the pipeline with the version of data and code specific to this run and establishes a linage of the different artifacts produced from the components, cache intermediate steps and compare results (based on *Tensorboard*). TFX support different deployment environments like *bare-metal*, *Kubernetes* and *cloud platforms* but for this installation *Kubernetes* will be used. As fast and reliable data store *HDFS*, as well deployed on *Kubernetes*, will be used.

Figure 4 shows the different components of which TFX is composed. With the top bar indicate which components are powered by *Apache Beam* as distributed data processing framework and the row below to which category each component belongs to. Below that, the different components are shown.

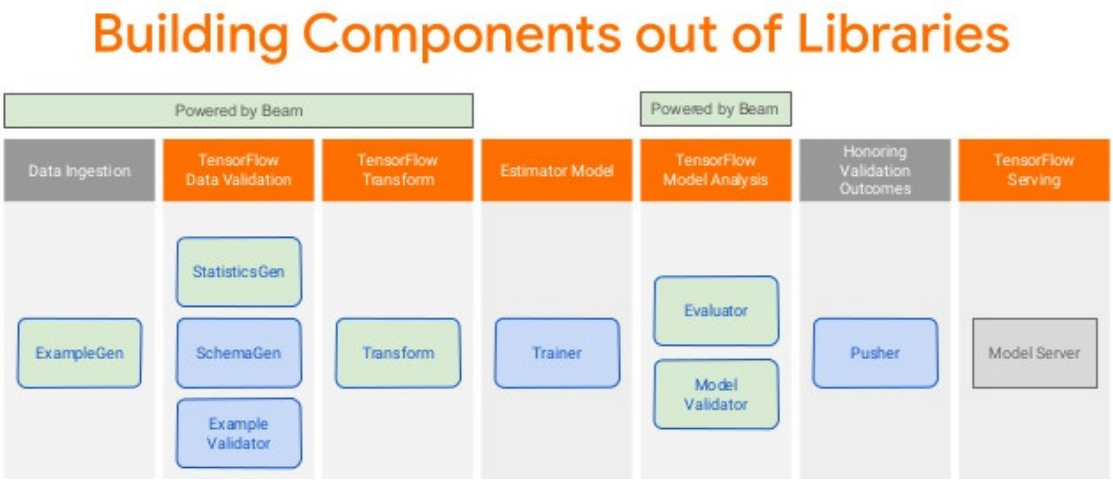

Figure 4: TFX components Crowe (2019)

The Components have the following purpose:[2]

1. **ExampleGen**
   is the initial input component of a pipeline that ingests and optionally splits the input dataset.

2. **StatisticsGen**
   calculates statistics for the dataset.

3. **SchemaGen**
   examines the statistics and creates a data schema.

4. **ExampleValidator**
   looks for anomalies and missing values in the dataset.

5. **Transform**
   performs feature engineering on the dataset.

6. **Trainer**
   trains the model.

7. **Evaluator**
   performs deep analysis of the training results and helps to validate the exported models, ensuring that they are "good enough" to be pushed to production.

8. **ModelValidator**
   checks the model is actually servable from the infrastructure, and prevents bad model from being pushed.

9. **Pusher**
   deploys the model on a serving infrastructure.

The components of *Data*, *Preprocessing* and *Model* from figure 3 can be mapped to the TFX components *ExampleGen*, *Transform* and *Trainer*. Where the format for saving the parsed data have to be changed from *Parquet* to *TFRecords* to be compatible with the TFX components. This has the benefit of having the same on disk and in memory format which can be directly feed into *Tensorflow* without conversion and thus save main memory and CPU cycles.

4.3. Tool Review

In this section the chosen tools from the current data processing and the proposed architecture are reviewed in short and possible alternative tools are outlined.

[2] https://www.tensorflow.org/tfx/guide

4.3.1. Data Storage

HDFS As data storage technology the Hadoop Distributed File System (HDFS) is chosen. HDFS provides a highly fault tolerant data storage distributed over a cluster of nodes with data replication. It provides high availability, high throughput and makes very large datasets possible. Notable limitations are a write-once-read-many access model where written files can not be modified but just appended and a high latency in data access. Both limitations do not concern the data access pattern of the pipeline and therefore do not constitute restrictions for the data pipeline.

Data Format

Parquet Parquet is a widely used and efficient column based data format with fast file access. The columnar nature makes data analytics over a subset of the dataset efficient and fast. The file output can be compressed with a granularity on column level which provides economical use of storage and can also speed up I/O further, depending on the chosen compression algorithm.

TFRecords & tf.Example *TFRecord* is a simple format for storing a sequence of binary records and can only be read sequentially. For each record the file contains a *CRC32C* hash for integrity checking. *tf.Example* is a *Protocol Buffers* message and is essentially a *{"string": tf.train.Feature}* mapping where the *tf.train.Feature* is one of the three types *tf.train.BytesList*, *tf.train.FloatList* and *tf.train.Int64List*. So *tf.Examples* are to serialize structured data. Although there is no requirement to use *tf.Example* in *TFRecord* files it is beneficial as the *ExampleGen* emits *tf.Examples* and the other TFX pipeline components expect this format.

Apache Arrow Apache Arrow is a cross-language platform for in-memory data with a columnar memory format. It provides zero-copy reads with a shared memory on the system between different processes and thus without serialization overhead. Furthermore it is integrated into projects like *Parquet* and *Pandas* which allows to read *Parquet* files directly into *Pandas Dataframes*, which is used by the current data processing.

4.3.2. Preprocessing

Tools from Nitsche und Halbritter (2019) As mentioned in section 3.3.1 the articles are available in *NewsML-G2* IPTC (2018b) format which is not suitable for preprocessing and model training. Thus Nitsche und Halbritter (2019) have developed a parser, which converts the *NewsML-G2* articles into structured *JSON*. The *Go* implementation is fast and suitable as a component for a data processing pipeline. The *JSON* format is chosen for its widespread use with very good support in many tools and libraries. Initially the parsed articles are stored in a *MongoDB* from where it can be analyzed and further converted to other formats for which they have also developed a convenient *CLI* application.

Numpy *Numpy* provides the *ndarray* data structure which is a multidimensional array with homogeneous types and fixed allocated memory with efficient memory usage. The same underlying memory representation is used by *Tensorflow*, so *numpy arrays* can easily used as *Tensorflow* input and conversion between *numpy arrays* and *Tensorflow tensors* are cheap and easy. Furthermore *numpy* provides fast algorithms on *ndarrays* and several libraries, e.g. *Pandas*, use this arrays as data structure or provide compatibility to it. This brings the benefit of using a broad tower of tools for preprocessing without complicated and costly conversion between the data structure used for preprocessing and input for *Tensorflow*. So at the current data processing *numpy arrays* are used as data structure wherever possible and as input for *Tensorflow*.

Pandas *Pandas* is a library for manipulation and analysis of tabular data and allows SQL like queries. Its main data structure is the *dataframe* which is based on *numpy arrays Pandas* was mainly used to analyze the dataset and for preprocessing tasks like filter missing values or group articles into time ranges.

Keras Data Preprocessing *Keras* provides some data preprocessing utilities. In the current data processing the *tokenizer* from *Keras* is used to create a word index and transform the whole vocabulary of the text corpus into integer tokens. So each article is represented as a vector of integers. The *tokenizer* can also be used to filter the words by frequency and by a list of characters. Additionally it provides to convert the text to lowercase and replace out-of-vocabulary words during tokenization.

4.3.3. ML Framework

Keras Essentially *Keras* is an API for ML frameworks. It provides high level abstractions to create ML models on top of ML frameworks like *Tensorflow*, *Microsoft Cognitive Toolkit*, *Theano* or *PlaidML* without using low level operations. Thus it makes ML model creation faster and enables to switch the backend for the actual computations. On top of this, *Tensorflow* has chosen to officially use *Keras* as high level API and therefore provides an excellent integration. The high level API and the possibility to seamlessly interweave *Keras* and *Tensorflow* operations was the reason to decide for *Keras*.

TensorFlow *Tensorflow (TF)* is the ML framework in the pipeline doing the actual training of the models. In TF models are expressed as a directed acyclic graph (DAG) of differentiable operations. Each operation provides information for differentiation to enable automatic back-propagation in neural networks. So the user defines the DAG and the actual computation is done internally by TF. The advantage of this is, that TF can optimize and rearrange the graph for optimization and each operation in the DAG can be individually executed on different available devices like CPUs, GPUs or TPUs. Additionally to parallelisation through multi GPUs, TF supports distributed computing in clusters. TF was chosen for its high performance, scalability, broad community support and wide adoption.

4.3.4. TFX ML Platform

As mention before TFX is a platform for end-to-end data pipelines and is build atop of other frameworks and tools beside *Tensorflow*. These are briefly introduced in the next paragraphs. For a introduction to TFX see section 4.2.

Data Processing For data processing TFX relies on *Apache Beam* which is, like *Keras* for ML frameworks, a abstraction and API for *Big Data* frameworks. *Beam* support several runners, respectively distributed processing backends, with *Apache Spark* and *Apache Flink* among the most recognized.

Apache Beam *Beam* provide a programming model and abstractions that simplify the mechanics of large-scale distributed data processing. *Beam* by itself doesn't do any computations but delegate these to runners and thus introduces a unified programming model for several distributed processing frameworks. Main abstractions include:[3]

1. **Pipeline**
   A pipeline warps up the whole data processing into a pipeline from reading input data over data transformations to write output data. A pipeline shapes an arbitrarily complex processing graph.

2. **PCollection**
   Represents a distributed data set that the Beam pipeline operates on. Each step in the pipeline have a *PCollection* as input and output. The data in a *PCollection* can be bounded and unbounded respectively batches or streams.

3. **PTransform**
   Represents a data processing operation, or a step, in the pipeline with one to many *PCollections* as input and zero to many *PCollections* as output.

4. **I/O transforms**
   Predefined *PTransforms* that ships with *Beam* for data ingestion or output to various external storage systems.

Beam Runners *Beam* supports multiple runners which do the actual distributed computations. At the time of writing, supported runners are *Apache Apex*, *Apache Flink*, *Apache Spark*, *Google Cloud Dataflow*, *Apache Gearpump*, *Apache Samza*, *Apache Nemo*, *Hazelcast Jet* and a *DirectRunner* which runs on the local machine and is meant for development. Typically the runners by itself use a DAG to represent the dataflow, where each of the runners provide their own fault tolerance and distribution strategies, failure semantics and other properties. For a *Beam Capability Matrix* see The Apache Software Foundation (2020).

[3] https://beam.apache.org/documentation/programming-guide/

Workflow For workflow management TFX provides out of the box two possible frameworks, with the option to implement an integration for another framework by oneself.

Apache Airflow *Airflow* is a tool for describing, executing, and monitoring workflows and relies on *configuration as code*, where workflows are created with python code. To mange the workflow orchestration *Airflow* uses, as well, DAGs. The DAGs represent the collection of all tasks to be run and reflects their relationships and dependencies. *Airflow* may execute a arbitrary number of DAGs, with each DAG consists of an arbitrary number of tasks. The description whats actually done by a task is defined by *Operators*. Examples are the *BashOperator* which executes a bash command or the *PythonOperator* which calls an arbitrary Python function. The execution of tasks are done by *Executors* where the *Airflow scheduler* determines when to execute which task. Available *Executors* are *Celery Executor*, *Dask Executor*, *Debug Executor*, *Kubernetes Executor* and *Scaling Out with Mesos*.

Kubeflow (pipelines) Kubeflow is the ML toolkit for Kubernetes and was build for making deployments of ML workflows on Kubernetes simple, portable and scalable. TFX can leverage the *Kubeflow Pipelines* component of *Kubernetes* for workflow definitions, which enables composition and execution of reproducible workflows on Kubeflow. A *pipeline* is the description of an ML workflow and includes all *components* of a pipeline and a graph (not acyclic) which defines how the components relates to each other. The *components* makes up the steps in the workflow and are self-contained sets of code, which gets packaged as *Docker containers*. The pipeline with its components and graph are defined through a DSL with a python SDK. To run the pipeline a *Pipeline Service* is called which again calls the *Kubernetes API server* to create the necessary Kubernetes resources to run the pipeline. A set of *Orchestration controllers* execute containers according to the pipeline definition to complete the pipeline run.

5. Experiments

The following segment discusses the conducted experiments and present the results. As the experiments are just a verification of the POC they don't measure the recommendation performance but analyze the training behavior of the models regarding resource consumption.

5.1. Hardware

The hardware on which the test are conducted are summarized in table 2. The used hardware is not a bare-metal installation but a virtual machine with CentOS as guest system.

CPU	9 cores, 18 Threads
RAM	164 GB
GPU	5 x Nvidia Quadro P6000
VRAM	24 GB per GPU

Table 2: Hardware for the conducted experiments

5.2. Training Data

The training is done with $\sim$45 000 articles after removing all articles which have missing *text*, *keywords*, *subjects* or *mediatopics* fields. The article triplets generated for each article, which are input for the *autoencoder*, takes up 6.4 GB as serialized *numpy array*.

5.3. Artificial User Data Generation

The generation of the user data for 1500 users took approximately 30 min and consumed about 10 GB of memory. The method is not implemented to leverage parallel computation and would approximately scale linearly with used CPU cores/threads when adapted to take advantage of multiple cores. Since 10 GB of memory consumption is relatively low and the generation have to be done just once for a particular user amount, no optimization es needed.

5.4. Autoencoder

The *autoencoder* is trained with all 45 000 articles each epoch, over 50 epochs with a batch size of 256, like in the original paper. The training took about 4 hours and consumed approximately 50 GB of memory. No sings of remarkable memory consumption or unexpected behavior regarding CPU and GPU usage has been noticed. As the training is done in a reasonable time and the system resource usage is within scope of available hardware no further optimization is needed at the moment. With more training data it may be worth to analyze the runtime behavior of the model more deeply to improve the training time.

5.5. User Representation RNN

The *user representation RNN* is trained with the history of 1500 artificial users and their corresponding session data. The model shows a high memory usage of about 160 GB. Also the VRAM consumption is high, that only a batch size of five, one batch per GPU, is possible. Higher batch sizes result in out-of-memory failures of the GPU. The CPU usage is periodically high at the beginning of a new step and low afterwards, so this indicates the CPU is mainly used while batch data generation and should not be a bottleneck in the training process. A strange behavior can be observed at GPU utilization. One GPU have a utilization between 20 and 70 percent while the remaining four GPUs shown a utilization of about 15 percent. At the moment the usage of *Tensorboard* to further investigate this pattern is not possible since, despite correctly

initializing the *Tensorboard callback* for training, no profiling data is recorded and the *Trace Viewer* can not be used. A guess would be that, mistakenly, the model parameters reside and get merged in the memory of the GPU with higher utilization instead of main memory. This may explain the unevenly distributed workload among the GPUs. This would also put high load on the VRAM of one GPU and prevent higher batch sizes since the batch is distributed evenly over the GPUs and a full VRAM of one GPU restrict the overall batchsize regardless if the others have free memory. The overall low GPU utilization may be explained doe to a difficulty to parallelize the RNN with a huge input sequence. Also the time per step with about $45\,\mathrm{s}$ is exceptionally high and have to be profiled.

6. Summary & Outlook

6.1. Summary

In section 2 we did a quick dataset analysis of the approximately $200\,000$ German news articles in the *NewsML-G2* IPTC (2018b) data format with a lot of metadata as seen in table 1.

This was followed up by section 3 where a review of the methods (3.1), mainly the DAE (Denoising Autoencoder) for the *Article Representation* and the RNN for the *Deep User Representation Model*, are presented. Also the adaptions to the dpa dataset are explained (3.2) with the definition of a user *session* and similarity with multi category articles opposed to single categories in Okura u. a. (2017). This is succeeded by the necessary data preprocessing (3.3) consisting of parsing based on Nitsche und Halbritter (2019) and generation of *Autoencoder Training Data* with *word token extraction* and *article similarity tripplets* based on the *Jaccard Index*. Also the *User Representation Model Training Data* generation, with sequences of article embeddings as input and multi dimensional labels consisting of article embedding sessions, with live data augmentation while training, is explained. Additionally the *Artificial User Data* generation is explained, necessary for a lack of real user data at the moment. This is followed by implementation insights of the models and especially the loss functions which were a main problem during the implementation of the paper. The paper review section is closed by 3.5, explaining the limits of the paper methods, especially when applied to the dpa dataset, like information loss due to fixed length token vectors to generate article embeddings and models not efficient for very long sequences.

The next section is about the pipeline architecture (4). The current and proposed architecture are put down in 4.1 with the current manual orchestrated pipeline depicted in figure 3 and the proposal to use TFX (Tensorflow Extended) as a platform for the pipeline. After that, TFX with its components is briefly described (4.2) and a tool review with all current used and to be used tools with TFX are explained with their association to the data pipeline (4.3).

The last section (5) explains the conducted experiments with the developed POC (Proof of Concept). The experiments focus on the executability of the data pipeline and its adapted RS model with runtime and resource usage in the foreground. The actual recommendation performance will be tested and compared in the follow up project.

6.2. Outlook

As this document reviews the tools and data processing architecture needed to process the presented dataset and calculate recommendations with the methods of Okura u. a. (2017), the model implementation is a POC to evaluate the data processing pipeline. So the next step would be to do profiling of the model implementation and improve the runtime behavior.

The cumming up project constitutes of the implementation of several other deep RS methods for comparison, with a good recommendation performance in mind, while deploying and leveraging the presented TFX pipeline. To this end, a literature review of current deep RS will be inevitably. Furthermore it may be necessary to test the data processing pipeline with additional media such as pictures and evaluate possible changes to be made.

Acknowledgment

I would like to thank Dr. Gerd Kamp for providing the great dataset and Prof. Dr. Kai von Luck for supervising and supporting this project and providing valuable suggestions. Furthermore a big thanks to the people of the CSTI [4] for making the hardware infrastructure available with an incredible support. Last but not least thanks to all members of the machine learning working group (ML AG) at HAW Hamburg, in which context this project was developed, for great discussions and mutual support for each others projects.

A. User Representation RNN Loss Function Details

Figure 5 shows the complete *DAG* of the RNN loss function. The rectangles are the operations in the graph, where red rectangles indicate *Keras* operations and orange rectangles represent *Tensorflow* operations. One requirement was to use as most as possible *Keras* operations to be flexible to possible exchange the backend but some operations where not present in the *Keras* API, have not the needed parameters (concrete values instead of Tensors) or the *Tensorflow* operation is much more efficient (e.g. *log_sigmoid* operation instead of two separate *log* and *sigmoid* operations). The rounded boxes *Compute Loss* and *batch_loss* nodes are the start and end nodes of the graph and parallelograms mark inputs. The edges represent the inputs and outputs of the operations which are tensors and the round nodes are just named tensors for a more descriptive and clear structured graph. The rectangles around nodes merely structure the graph into logical units and are not part of the actual graph. The blue and green rectangles are special in that they mark the function which is applied by the map function to its input.

The input to the loss function *y_true* is a batch of user sessions where each session contains the viewed and not viewed article embeddings. The input *y_pred* is the batch of predicted outputs of the model and consists the user embeddings. The batches are packed into a list to get iterated by a map function *map_fn* in parallel where the blue *Lost per Sample* rectangle marks

[4] https://csti.haw-hamburg.de/

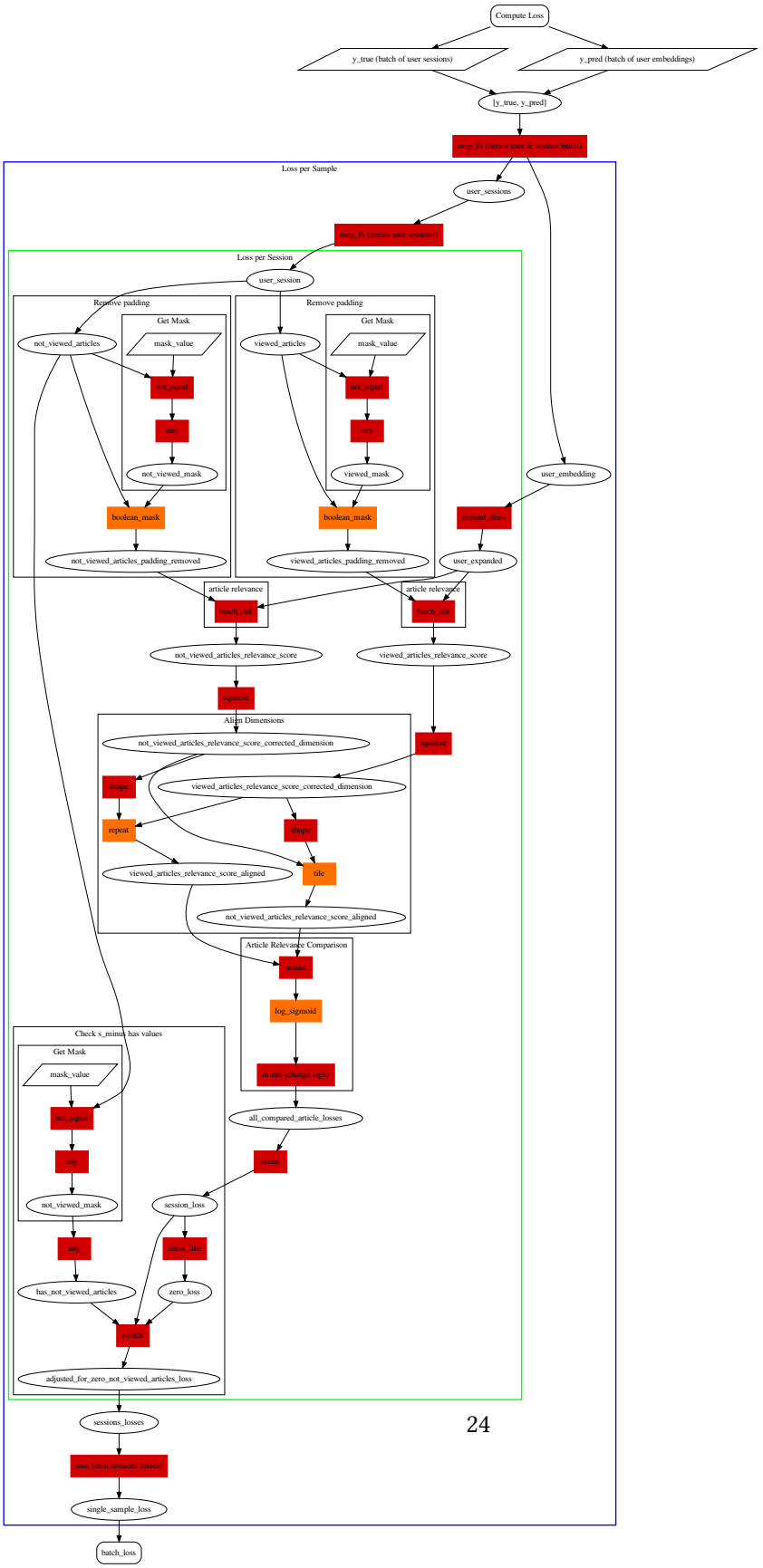

Figure 5: Computation graph of the adapted RNN loss function

the subgraph which is applied by the *map_fn* to every sample in the input batch. In the function of this subgraph, first the *user_embedding* and *user_session* gets extracted from the input tensor. The *user_session* tensor is the input to be iterated by the next *map_fn* where the green *Loss per Session* rectangle indicates the subgraph which is applied by the *map_fn* to every session. In every subgraph of this sample the same *user_embedding* is used for the sessions. In the green subgraph the user session is split into two tensors, where one represents the embeddings of the viewed and the other the embeddings of the not viewed articles. The *Get Mask* subgraph extracts the indices of the zero padded values which is used by the *Tensorflow boolean_mask* operator to generate a new tensor with the padding values removed. This is done to the viewed and not viewed articles in the *remove padding* subgraph. Now the article relevance is computed by doing a *batch_dot* operation for all viewed and not viewed articles with the user_embedding, whose dimension is expanded to be *broadcasted* by the *batch_dot* operator. Broadcasting is the process of making arrays with different shapes have compatible shapes for arithmetic operations. [5] [6] The resulting tensor contains the relevance score for the articles. Instead of doing a nested iteration of the viewed and not viewed articles like in equation 2, the tensors get aligned in that way, that a element wise *minus* operation can be performed to compare the viewed articles with all not viewed articles. The transformation is done in subgrap *Align Dimensions* with resulting tensors as shown as an example in table 3. Now the aligned viewed and not viewed articles

Viewed & not viewed tensor			
Index	0	1	2
Viewed	1	1.5	
Not viewed	0.5	1	3

Aligned viewed & not viewed tensor						
Index	0	1	2	3	4	5
Viewed	1	1	1	1.5	1.5	1.5
Not viewed	0.5	1	3	0.5	1	3

Table 3: Example of aligned viewed and not viewed tensors with the original tensor to the left and the aligned tensor to the right. The colors indicate which values are copied

tensors get compared, which forms a new tensor with the losses for all comparisons of articles. A mean operation computes the session loss of all comparisons, which is a substitution for dividing all comparison values by the sum of all articles in this session and summing them up like in equation 2. This not only makes the definition of the graph easier and more clear but also reduces the number of executed operations. The last step to compute the loss of the session is to check that the tensor of not viewed articles has values. This is realized with the control flow operation *switch* of *Keras* which chooses a zero loss tensor if there are no values in the not viewed article tensor or the computed *session_loss* tensor otherwise. Unlike one would do it in plain Python the decision which tensor is chosen is done after the definition of the before described *session_loss* but the *switch* operator need both input tensors which it can select and the *tensorflow* backend cares about efficient execution of the computation graph. This step has to be done only for the not viewed articles since in the preprocessing phase a session is

[5] https://www.tensorflow.org/api_docs/python/tf/broadcast_to
[6] https://www.tensorflow.org/xla/broadcasting

only constructed if there are viewed articles in the time span defined to form a session. The computation in the green rectangle is done and the computed *session_losses* get summed up in the blue rectangle subgraph, which represents the loss of the current sample. The output of the loss function is the *batch_loss* which is the tensor of all sample losses.

References

[Crowe 2019] Crowe, Robert: *TensorFlow Extended: An end-to-end machine learning platform for TensorFlow*. 2019. – URL https://www.slideshare.net/FlinkForward/flink-forward-san-francisco-2019-tensorflow-extended-an-endtoend-machi – Zugriffsdatum: 2020-05-13. – Library Catalog: SlideShare

[He u. a. 2017] He, Xiangnan ; Liao, Lizi ; Zhang, Hanwang ; Nie, Liqiang ; Hu, Xia ; Chua, Tat-Seng: Neural Collaborative Filtering. In: *Proceedings of the 26th International Conference on World Wide Web*. Republic and Canton of Geneva, Switzerland : International World Wide Web Conferences Steering Committee, 2017 (WWW '17), S. 173–182. – URL https://doi.org/10.1145/3038912.3052569. – Zugriffsdatum: 2018-06-04. – ISBN 978-1-4503-4913-0

[IPTC 2018a] IPTC: *Media Topics*. 2018. – URL https://iptc.org/standards/media-topics/. – Zugriffsdatum: 2018-09-21

[IPTC 2018b] IPTC: *NewsML-G2*. 2018. – URL https://iptc.org/standards/newsml-g2/. – Zugriffsdatum: 2018-09-21

[Nitsche und Halbritter 2019] Nitsche, Matthias ; Halbritter, Stephan: Development of an End-to-End Deep Learning Pipeline. (2019), S. 23

[Okura u. a. 2017] Okura, Shumpei ; Tagami, Yukihiro ; Ono, Shingo ; Tajima, Akira: Embedding-based News Recommendation for Millions of Users. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM, 2017 (KDD '17), S. 1933–1942. – URL http://doi.acm.org/10.1145/3097983.3098108. – Zugriffsdatum: 2018-05-20. – ISBN 978-1-4503-4887-4

[Okura u. a. 2016] Okura, Shumpei ; Tagami, Yukihiro ; Tajima, Akira: Article De-duplication Using Distributed Representations. In: *Proceedings of the 25th International Conference Companion on World Wide Web*. Republic and Canton of Geneva, Switzerland : International World Wide Web Conferences Steering Committee, 2016 (WWW '16 Companion), S. 87–88. – URL https://doi.org/10.1145/2872518.2889355. – Zugriffsdatum: 2019-05-13. – event-place: Montréal, Québec, Canada. – ISBN 978-1-4503-4144-8

[The Apache Software Foundation 2020] The Apache Software Foundation: *Apache Beam Capability Matrix*. Mai 2020. – URL https://beam.apache.org/

documentation/runners/capability-matrix/. – Zugriffsdatum: 2020-05-24

[Yang u. a. 2016] Yang, Zichao ; Yang, Diyi ; Dyer, Chris ; He, Xiaodong ; Smola, Alex ; Hovy, Eduard: Hierarchical Attention Networks for Document Classification. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California : Association for Computational Linguistics, 2016, S. 1480–1489. – URL http://aclweb.org/anthology/N16-1174. – Zugriffsdatum: 2020-04-30