Report on Replicating and Extending the Hierarchical Recurrent Encoder-Decoder model

Maurits Bleeker
maurits.bleeker@student.uva.nl

Maartje ter Hoeve
maartje.terhoeve@student.uva.nl

Jörg Sander
jorg.sander@toologic.com

Thijs Scheepers
thijs.scheepers@student.uva.nl

ABSTRACT

We replicate the work of [13] and present our own implementation of a hierarchical encoder-decoder for query suggestion. The query suggestions that are output by this model look satisfying. We extend the vanilla model by adding an attention layer and separately by using character n-grams as the building blocks of our queries instead of words. The latter implementation is not producing correct suggestions yet. The addition of attention looks promising, yet still needs to be refined.

1. INTRODUCTION

Searching for information is a task most of us do daily. We have to devise a query, enter it into a search box and select a good looking result. These steps we repeat over and over, each time with a slightly different query, until we find what we are looking for. This might sound familiar, coming up with a good query is hard. Therefore most search engines suggest to auto-complete a query while typing into a search box. Furthermore they might suggest a new query on the result page.

While searching for information our subsequent search queries are related. Can we exploit this latent structure to make better query suggestions?

Sequence-to-sequence [15] recurrent neural network (RNN) [16, 11] models are used to predict a new sequence from a given sequence, and are heavily used in neural machine translation (NMT). This type of model is also suitable for predicting a next query given a previous query. However, for query suggestions we like to take more than one sequence, i.e. query, into account. The hierarchical recurrent encoder-decoder (HRED) [13] allows us to do this by introducing hierarchical encoder layers.

The aim of this study is twofold. First we present a replication of [13], where a HRED model has been implemented to perform context aware query suggestion. Secondly, we extend this model in two ways. We enhance the model by adding an attention-mechanism [1, 9]. In a separate experiment we replace the word-based input tokens with character n-gram based input tokens, in an attempt to reduce sparsity in the input.

The model is able to produce candidate query suggestions generatively. From a given sequence of queries we encode each query into a query representation, from which we encode every query representation into a single session representation. A decoder then produces new query suggestions token by token, figure 1 illustrates this. We can generate multiple suggestions by performing beam search. In a qualitative analysis we show the performance of our extensions using these generated queries.

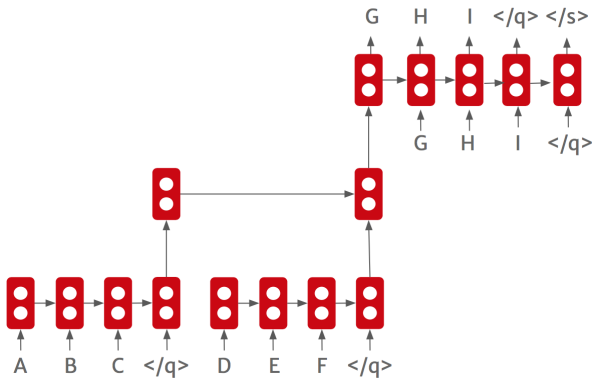

Figure 1: Schematic structure of the vanilla HRED used during inference, that is used for query suggestion

The model is also able to generate features to make an already existing suggestion system perform better. These systems are usually trained using *learning-to-rank* approaches. With a session representation we can enter a candidate into the decoder and determine how likely the candidate query is under the model. This likelihood can be used as a feature in a learning to rank algorithm. In a quantitative analysis we show the performance of our extensions using ranked queries with our extracted feature.

2. VANILLA HRED

In the first part of this study we aim to replicate the model and experiments described in [13]. In this section we describe the model, our own implementation details and experimental results.

2.1 Key idea

When a user enters a query to a modern search engine, the search engine supports the user by providing query suggestions. [13] implements an HRED for this task. Figure 1 shows the architecture of this network. The network consists of three components: a query encoder, a session encoder and a query decoder (we will use the terms query decoder and decoder interchangeably hereafter). The session encoder is fed with the state of the query encoder after the query encoder has encountered an end of query symbol. Note that the query encoder is reset after reading an entire

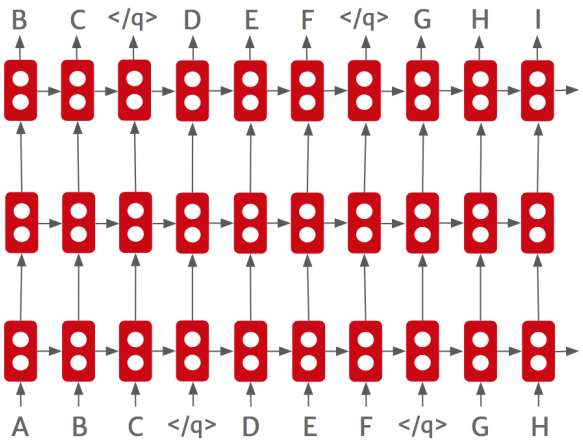

Figure 2: A schematic depiction of a three-layer RNN language model. This structure forms the foundation of our HRED implementation.

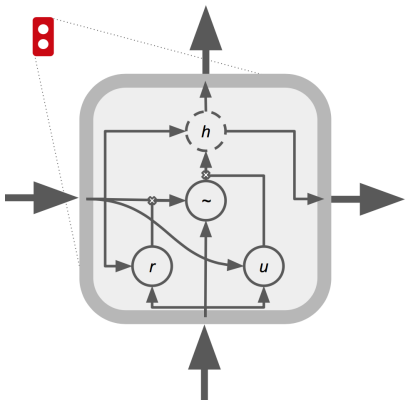

Figure 3: The GRU contains two gates, r and u, which determine the change of $\tilde{h}$ and h_{t+1} respectively.

query. By adding an additional session encoder, the model keeps a summary of the queries seen so far. Only the relevant information is kept in the session hidden states. This allows the model to take the context of the entire session into account when predicting the next query. The decoder outputs this next query suggestion.

2.2 Implementation

The implementation of our HRED model[1] was done using the Tensorflow machine learning framework. The foundation upon which our model is built resembles a three-layer RNN language model—as shown in figure 2. The inputs are passed through an embedding layer, which transforms one-hot-vectors into a token embedding. These embeddings are then passed through three RNN layers, finally resulting in an output layer. Each RNN layer is a modified GRU-RNN (Gated Recurrent Unit denoted GRU hereafter) so the layer has the functionality of either a query encoder, a session encoder or a decoder. All details of the model are described below. All hyper-parameters of our model are summarized in table 2.

[1]An implementation of the model is available at https://github.com/tscheepers/hred-attention-tensorflow

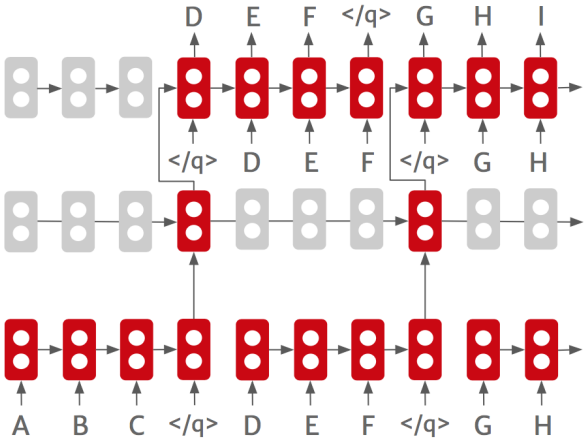

Figure 4: Using the end-of-query mask we give each layer of GRU units a separate functionality. This results in the final architecture used during training.

2.2.1 *Plain Gated Recurrent Units*

We use GRUs [2] as the non-linear transformations, i.e. as the hidden states of our HRED model. The mathematical outline of the GRUs is given by

$$\begin{array}{lll} r & = \sigma(xW_{ir} + h_{t-1}W_{hr} + b_r) & \text{reset gate} \\ u & = \sigma(xW_{iu} + h_{t-1}W_{hu} + b_u) & \text{update gate} \\ \tilde{h}_n & = tanh(xW_{i\tilde{h}} + r \cdot h_{t-1}W_{h\tilde{h}}) + b_{\tilde{h}} & \text{candidate update} \\ h_t & = 1.0 - u \cdot h_{t-1} + u \cdot h_{tilde} & \text{final update} \end{array} \quad (1)$$

, where σ is the logistic sigmoid function, x is an input vector, all $W_{subscript}$ are the weight matrices of the GRU layer, all b are the bias vectors of the GRU layer and h_{t-1} is the previous recurrent state. Figure 3 shows this schematically.

2.2.2 *End-of-query mask and layer-wise GRU modifications*

We create an HRED model (figure 4) from our three-layer RNN foundation (figure 2) by modifying the GRU for each layer. Each modified layer acquires special functionality by incorporating an end-of-query mask. This mask is a binary vector of the session length, and contains ones instead for the time step where an end-or-query token is entered.

The first RNN layer, i.e. the *query encoder*, should result in a vector representation of each query. It therefor needs to accept embeddings directly and forget its current state when starting to encode a new query. This layer uses the end-of-query mask for exactly this. It resets the state of the GRU after each query. This means that the hidden state passed onto the next timestep (h_q^{next}) will be reset to the zero vector when the end-of-query token is passed to the unit. However h_q directly is passed to the the session encoder unmodified.

The *session encoder*, i.e. the second RNN layer, accepts the hidden state from the query encoder as input. It should encode all the query representations into a single session representation. This means the layer should only actually do something when it receives a new query representation.

Therefor, the GRU is modified so as to pass $h_{s,t-1}$ along without modifications and only activate when an end-of-query token is passed to the model. This is illustrated in figure 4.

The *decoder* does not directly accept the hidden state from the session encoder as input, instead it accepts embeddings, just like the session encoder. The previous hidden state of the decoder ($h^*_{d,t-1}$) is determined by:

$$h^*_{d,t-1} = \begin{cases} \tanh(h_{s,t}W_{sd} + b_{sd}) & \text{if end-of-query} \\ h_{d,t-1} & \text{otherwise} \end{cases} \quad (2)$$

We use an activation over the hidden state of the session encoder ($h_{s,t}$), when the end-of-query token is passed, and the regular previous hidden ($h_{d,t-1}$) state otherwise. Figure 4 illustrates this. This is similar to a regular encoder-decoder model.

2.2.3 The output layer and computation of the output probabilities

Before computing the final output probabilities we add an additional output layer, given by

$$y(h_{d_{s,t-1}}, w_{s,t-1}) = h_{d_{s,t-1}}W_h + W_i x_{s,t-1} + b, \quad (3)$$

where h_d is the decoder output, $W_{subscribt}$ are weight matrices, x is a word embedding that served as input to the query layer and b is a bias term. We multiply the outcome of this layer by the output embedding of the decoder state and take a softmax over these scores to compute the output probabilities,

$$P(w_{s,t}|w_{s,1:t-1}, Q_{1:s-1}) = \frac{\exp o_t^T y(h_{d_{s,t-1}}, w_{s,t-1})}{\sum_k \exp o_k^T y(h_{d_{s,t-1}}, w_{s,t-1})} \quad (4)$$

,

where Q represents the queries in the session seen so far and o_t is a word embedding. This type of scoring function is proposed in [13] as it has been shown that if o_t is close to $y(h_{ds,t-1}, w_{s,t-1})$ the word at step t has a high probability.

2.2.4 Beam Search

To generate multiple candidate queries from our model, we first encode an entire session and perform beam search while decoding. The softmax gives us a resulting probability for each token which we use to decide the n-most likely tokens each step of the search. We use a beam size of 25. There is no need to do any recombination during the search, because we use words as our smallest units and no two resulting strings can thus be exactly the same.

2.2.5 Optimization procedure

We use Tensorflow's softmax-cross-entropy loss function to compute the model loss. Furthermore we use the Adam optimizer [7] with a learning rate of 0.0001 and a batch size of 80.

During the optimization procedure we create batches where sessions of a similar length are grouped. If sessions are padded to the length of the longest sentence in the batch. Grouping sessions of equal length results in less padding symbols over the entire batch, and thus will result in less computation time lost.

Padding symbols are masked before applying the cross-entropy loss function, so the model will ignore the gradients over the padding symbols and not overfit on them.

2.3 Experiments

In order to test the ability of the HRED model to propose the next query based on the previous user queries we use a learning-to-rank approach (outlined in section 2.3.3) which is comparable to the one used in [27, 36] for query auto-completion and in [28, 34] for query suggestion. For a given session prefix (consisting of different user queries) we determine the set of suggestion queries based on co-occurrence frequencies (i.e. how often does a specific query follow an other). This is our first system which we denote *co-occurrence based ranker* (ADJ[2]), The second system we evaluate is the supervised *Baseline Ranker* (BR) which uses a) pairwise features that we compute between the suggestion query and the most recent query; b) contextual features depending on the history of previous user queries. The third system is an extension of the Baseline Ranker denoted BRplusHRED which uses an additional log likelihood score that our HRED model can generate for different next-query candidates. The performance of the individual systems is measured using the mean reciprocal rank (MRR) a common metric for these tasks [5, 10].

The key idea underlying this approach is the assumption that a system which has more valuable contextual information at its disposal is more likely to predict a next query that is desired by the user.

2.3.1 Dataset

We are using the famous AOL search dataset which was released in August 2006, and contains detailed search logs. The dataset consists of the search queries of roughly 650,000 users over a three-month period dating from 1 March, 2006 to 31 May, 2006. The dataset is split into a background, training, validation and test set based on the timestamp of the query. Non-alphanumeric characters and words shorter than two characters are removed from the queries[3] and we apply lower casing. Queries are grouped into a so called *session* denoted S where $S = \{Q_1 \dots Q_M\}$ if they belong to the same user and the last query was entered within 30 minutes of the last user's activity. We only keep sessions that consist of at least two queries and where the last two queries are not identical. Table 1 outlines the number of sessions in the four different types of datasets after filtering. From the background dataset we extract the 50K most frequent words which form the vocabulary V together with the special tokens *end of query* denoted `</q>`, *end of session* denoted `</s>` and *unknown token* `<unk>`.

2.3.2 Training the model

In order to train the vanilla model we merge the background, training and test set because due to limited computational resources we are not able to perform hyper-parameter optimization. The validation set is used during training to evaluate the performance. The model is trained with a batch size of 80 for 42 hours using a NVIDIA GeForce GTX 980Ti GPU. Table 2 specifies the model parameters in more detail.

[2]the abbreviation is taken from the original paper [13]

[3]if a query does not contain any other symbols after this operation, the query is removed from the dataset.

Table 1: Details of AOL dataset after filtering

Type of dataset	Number of sessions	timestamp period
Background	1937785	1 March, 2006 until 30 April,2006
Training	448302	1 May, 2006 until 15 May, 2006
Validation	291559	16 May, 2006 until 24 May, 2006
Test	180986	25 May, 2006 until 31 May, 2006

Table 2: Vanilla HRED parameters

Configuration parameter	value
Optimizer	Adam
Learning rate	10^{-4}
Size of embedding layer	128
Size of query encoding layer	256
Size of query decoder layer	256
Size of session encoding layer	512
Maximum sequence length	50

Table 3: LambdaMART settings

Setting	value
Number of trees	50
Learning rate	0.02
The fraction of queries to be used	0.02
Number of features to consider	0.3
Maximum number of leaf nodes	10
Minimum number of samples	200

2.3.3 Evaluation method

As mentioned in the previous section we are using a learning-to-rank approach to evaluate the performance of the HRED model. More formally given a session $S = \{Q_1, \ldots, Q_M\}$, the aim is to predict the *target* query Q_M. The main idea is to rank for a sequence of queries as prefix (hereafter referred to as prefix or session prefix), the set $P = \{Q_1, \ldots, Q_{M-1}\}$, a set of suggestion queries C that most likely follow the prefix, denoted $C = \{Q_M^1, \ldots, Q_M^N\}$ where N denotes the number of candidates and is equal to twenty in all experiments (we will use the terms suggestions and candidates hereafter interchangeable). We denote the penultimate query Q_{M-1} of a session the *anchor* query which plays an important role when we define the set of candidates for a given query prefix. In section 2.3.4 we describe in more detail how these query suggestions are selected. For each pair of query prefix and query suggestion we extract pairwise, suggestion and contextual features that are fed into the ranker.

We utilize LambdaMART[4] as a supervised ranking algorithm. The ranker uses 50 trees and learns the parameters on a combination of training and test set[5]. Table 3 specifies the detailed settings of the LambdaMART implementation we use. The experiments use the mean reciprocal rank (MRR) as an evaluation metric.

The subsequent list describes the seventeen features that we compute for each pair of session prefix (including the anchor query) and target query.

1. frequency of the suggestion query that follows the anchor query in the background set;
2. frequency of the anchor query in the background dataset;
3. frequency of the suggestion query in background dataset;
4. Levenshtein distance between anchor and suggestion query;
5. average Levenshtein distance between the suggestion query and each query in the context;
6. number of characters of suggestion query + number of words suggestion query;
7. character n-gram similarity between the suggestion and the ten most recent queries in the context (resulting in ten features);
8. a score taken from the Query Variable Markov Model (QVMM) [4] which models the context of a query and is able to automatically back-off to shorter query n-grams if the exact context is not found in the background data. Please note that we set the context length (denoted D) to three.

The co-occurrence based ranker (ADJ) only uses feature item 1 from the above list to rank the session prefix suggestion pairs. The supervised Baseline Ranker (system 2) uses all features and the third system BRplusHRED utilizes as an additional feature the log likelihood score (abbreviated HRED score hereafter) that our model can generate for each pair.

In order to explore the different capabilities of the model we evaluate three different experimental designs

Test scenario 1: Baseline next-query prediction;

Test scenario 2: Robust next-query prediction;

Test scenario 3: Long-tail next-query prediction.

2.3.4 Test scenario 1: Baseline next-query prediction

In this setting we determine the set of sessions S and candidates C through extraction of twenty candidate queries from the background dataset for each session in the training, validation and test set. The anchor query of a session must exist in the background dataset and we determine the candidates by choosing the queries from the background dataset that most likely follow the anchor query. In other words we select the twenty queries with the highest co-occurrence counts between anchor and suggestion query. We only use sessions where the actual target query appears in the set C and the cardinality of C must be equal to twenty. Applying this procedure results in 17,960 sessions in the training set, 7,313 sessions in the validation set and 11,330 sessions in the test set. We compute the earlier described features for all

[4]we are using the following LambdaMART implementation https://github.com/jma127/pyltr

[5]again due to limited computational resources available we could not perform any hyper-parameter optimization

three datasets and use LambdaMART to evaluate the performance of the three described systems. We refer to this design as the baseline experiment and results are presented in section 2.4.1.

2.3.5 Test scenario 2: Robust next-query prediction

Common query sessions often contain navigational queries such as yahoo, google or ebay which do not belong to any particular search topic. We will refer to these queries as noisy. A system that bases its prediction on the previous context should learn to distinguish between noisy and relevant history. In order to evaluate whether our HRED model has this capability the following experiment is conducted. First we determine the 100 most frequent queries in the background dataset and label them as noisy. For each session in the training, validation and test set that we identified in the *baseline next-query prediction* experiment the session prefix is perturbed by injecting a noisy query at a random position. Please note that the probability of sampling a noisy query is proportional to its frequency in the background set. As an example, if the original session is *toyota parts* $\rightarrow$ *brake bulb* with target query *toyota car repairs* the noisy query *ebay* is inserted at a random position. We present the results of these experiments in section 2.4.2.

2.3.6 Test scenario 3: Long-tail next-query prediction

The previous two test scenarios use query sessions where the anchor query exists in the background dataset. These queries can be qualified as high frequent in comparison to anchor queries from the other three datasets that are not contained in the background set and which we will refer to as long-tail queries. In this experiment we will therefor retain sessions from the training, validation and test set for which the anchor query cannot be found in the background set. The set of candidate queries can not be determined based on the co-occurrence count and therefore we iteratively shorten the anchor query by dropping terms from the beginning of the anchor query until we have a query that appears in the background data. If this procedure does not reveal a match with any query in the background set we repeat the procedure and drop terms from the end of the anchor query. Finally if this does not result in a shortened anchor query the session is omitted from the experiments. In the end we obtain 2,501 sessions in the training set, 1,631 sessions in the validation set and 913 sessions in the test set.

Using long-tail queries as anchor query results in sessions that are considerable longer than the ones we obtained in test scenario 1 and 2. We would therefore assume that the context-aware HRED model would perform considerable better on this task compared to the other two systems that have less (Baseline Ranker) or no (co-occurrence ranker) contextual information at their disposal. We refer to section 2.4.3 for the results of this experimental design.

2.4 Results

In this section we present the qualitative and the quantitative results of our vanilla HRED model.

2.4.1 Results test scenario 1: Baseline next-query prediction

Table 5 presents the MRR scores for the three models we evaluate in this experiment. As expected the MRR scores between the simple ADJ model and the BR and BRplusHRED model differ considerably. The later two models attain a relative improvement of 68% with respect to the ADJ model. The difference is much larger than reported in the original paper (improvement of 4.3% resp. 7.8% for BR and BRplusHRED model) we are replicating as can be seen in column four of table 5. We presume that this difference can be explained by different pre-processing procedures e.g. the original paper applied spelling correction which we are not utilizing. Unfortunately there is no difference in MRR value between the Baseline Ranker and the BRplusHRED model that was found in the original paper. When performing the same experiment using the generated HRED score from the model built by [13] and replacing those with the HRED scores from our own model we obtain the same result[6].

Impact of Context Length. We also test whether the performance obtained by HRED on sessions that have a length of at least four queries can be obtained using a shorter context. For each long session in our test set, we artificially truncate the context to make the prediction depend on the anchor query, Q_{M1} , only (1 query), on Q_{M2} and Q_{M1} (2 queries resp. 3 queries). The resulting MRR values of the three conditions are shown in figure 5. The MRR values improve with context length although our differences between the three conditions are considerably less than the ones reported in the original paper. [13] reported an absolute delta value of 0.013 between the one-query context length and two-query context length where our results reveal a Δ value of of 0.004. and 0.009 between The original paper found a 0.004 MRR difference between the two-query context and three-query context performance where our results only reveal a Δ of 0.002 between these two experimental conditions.

2.4.2 Results test scenario 2: Robust next-query prediction

Table 6 shows the results of the experiments in which we perturbed the queries from test scenario 1 with noisy aka high frequent queries at a random position in the session context. Different than reported in the original paper our results are roughly the same as in the first experiment. One would expect that the ADJ and Baseline Ranker would suffer significantly from this context corruption and that the performance loss for the BRplusHRED model would be less affected. Based on these results (and the previous) we suspect that there is a flaw in the software that generates the features used in the learning-to-rank approach.

2.4.3 Results test scenario 3: Long-tail next-query prediction

Table 7 presents the results for the experiments in which the anchor query of a session belongs to the so called long-tail i.e. appears infrequently which we determined by selecting only sessions with an anchor query that does not appear in the background dataset. The performance of the ADJ model suffers considerably as expected and the MRR value drops to 0.15 compared to 0.32 in the first experiment. The performance of the other two models suffers less. The MRR value of the Baseline Ranker decreases to 0.44 (a Δ MRR of 0.10 compared to experiment 1) and the BRplusHRED

[6]we trained the model of [13] for roughly 48 hours and used the model to generate the HRED scores for the query suggestions

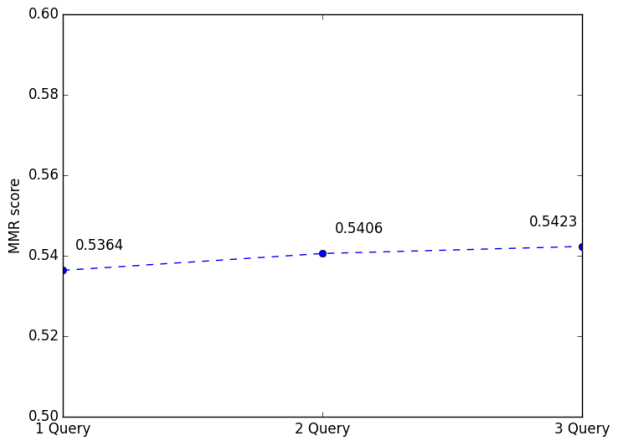

Figure 5: Variation of HRED performance with respect to the number of previous queries considered.

model achieves a MRR value of 0.43.

2.4.4 *Qualitative Results*

It is interesting to have a look at some of the query suggestions our model produces. In table 4 we show some interesting samples. Recall that we use beam search to sample the most likely suggestions. As a general observation we can say that very frequent queries like *google* or *ebay* are often among the sampled query suggestions. This is something we could normalize for in future work (see section 5). Note that the query suggestions in table 4 are not sorted in any way. As the data were preprocessed as to remove dots etc, these are not shown in the table either. As a first observation these query suggestions show that the model has learned to predict queries that are semantically related to the user's search queries. Second, queries that belong to topics that are often searched for in the data (e.g. sample 4) most often give good results. Not all samples produce queries that are as semantically related as the ones in the table. Especially queries that contain many unknown words receive a lot of frequent query suggestions, like *google* or *yahoo*. However, as these results look quite descent, we indeed expect to have made a small mistake in the quantitative analysis of the results, as was presented in the previous section.

3. HRED WITH QUERY-LEVEL ATTENTION

As a first extension we add attention to the original HRED model. In this section we describe our approach and the results.

3.1 Key idea

The main idea behind attention is to specifically focus on those parts of the input that are especially contributing to the correct prediction of the next word. Attention was first proposed in [1] to improve encoder-decoder models for neural machine translation.

3.2 Implementation

We implement attention on the query level. The motivation behind this is twofold. The query level is expected to contain more fine grained information that can be of use while decoding the next query word. Furthermore, the session level has been added to keep the relevant information of the query as a summary, which in a way serves the same goal as attention. In contrast to attention on traditional encoder-decoder architectures we attend on tokens of all previous queries, i.e. we do not attend over only one encoded sequence, but possibly over multiple sequences.

In [1] global attention is proposed: all hidden states of the encoder are attended over. In [9] local attention is suggested. In this type of attention, only specific hidden encoder states are attended over. We choose to implement global attention, where we only attend over full queries we have seen so far. Figure 6 shows a schematic overview of the HRED model with the additional attention layer. For example: when predicting next query word I, we attend over words A to F, but not over word G. We do not attend over end-of-query and end-of-session symbols. We achieve this by adding a mask that behaves similarly to the mask that was used in the vanilla HRED model. This mask contains zeros at the positions we do not want to attend over and ones at the positions we do want to consider for attention. In contrast to the mask used in the GRUs we have a different mask for every decoder state, instead of just one for all the time steps.

Figure 6 shows that we need to implement two additional layers when implementing attention. First we need a context layer (C in figure 6). This context layer receives the hidden states of the query encoder as input and outputs a context vector c_t. This context vector contains the information on what states should receive specific attention while decoding. The context vector is added to the current state of the decoder, h_d, by concatenating the context vector to the vector that is output by the decoder. This concatenated vector is fed to the output layer and from there on we proceed as described in section 2. The attention scores, i.e. how important each hidden encoder state is, are computed by the attention layer (A in figure 6). This attention layer receives a query encoder hidden state, h_q, and the current decoder state h_d as input and outputs an attention score, a_t, per source hidden state. We train these additional layers simultaneously when training the entire model.

3.2.1 *Mathematical details*

We use the same functions for the implementation as given in [9], which we will repeat in this section. First the align function for global attention, which is a softmax over all states we attend over:

$$a_t(s) = align(h_d, \bar{h_q}) \tag{5}$$

$$= \frac{exp(score(h_d, \bar{h}_q))}{\sum_s' exp(score(h_d, \bar{h}_q'))} \tag{6}$$

And as a scoring function for the weight of each query hidden state we will use the "general" approach as described in [9].

$$score(h_d, \bar{h_q}) = h_d^T W_a \bar{h}_q \tag{7}$$

Table 4: Some samples of query suggestions produced by the vanilla HRED model.

Sample 1		
Input	</q> princess cruise lines </q> princess cruise lines </q>	
Output	ebay </q>	google </q>
	cruises </q>	royal caribbean </q>
	princess cruise lines </q>	carnival cruise lines </q>
	royal cruise lines </q>	princess cruise ships </q>
	international cruise lines </q>	princess cruise line </q>
	royal caribbean news </q>	disney cruise lines </q>
	princess cruise deals </q>	american cruise lines </q>
	royal caribbean com </q>	norwegian cruise lines </q>
	royal caribbean cruises </q>	tom cruise lines </q>
	alaska cruise lines </q>	royal caribbean magazine </q>
	kings cruise lines </q>	princess cruise ship </q>
	princess cruise today </q>	princess cruise inc </q>
	carnival cruise ships </q>	
Sample 2		
Input	</q> st charles north high school </q>	
	st charles north high school </q>	
Output	google </q>	myspace </q>
	mapquest </q>	ebay </q>
	yahoo </q>	porn </q>
	bankofamerica </q>	maps </q>
	myspace com </q>	american idol </q>
	www myspace </q>	wells fargo </q>
	www google </q>	www myspace com </q>
	www google com </q>	www yahoo com </q>
	american idol com </q>	st john bulldogs </q>
	st george mason </q>	st john states </q>
	st patrick bad </q>	st john west </q>
	st john saint </q>	st patrick bad east </q>
	st george mason high school </q>	
Sample 3		
Input	</q> lake <unk> florida </q> lake <unk> florida </q>	
Output	google </q>	goggle </q>
	yahoo </q>	flowers </q>
	mapquest </q>	lake geneva florida </q>
	lake powell florida </q>	lake city college </q>
	lake city florida </q>	lake bell florida </q>
	lake holiday college </q>	lake forest florida </q>
	lake geneva texas </q>	lake geneva college </q>
	lake erie florida </q>	lake geneva california </q>
	lake state college </q>	lake city weather </q>
	lake geneva nj </q>	lake holiday world </q>
	lake arthur florida </q>	lake city california </q>
	lake powell college </q>	great adventure florida </q>
	lake holiday high school </q>	
Sample 4		
Input	</q> pornstar dynamite </q> pornstar dynamite </q>	
	pornstar dynamite </q> pornstar dynamite	
	</q> pornstar dynamite </q>	
Output	pornstar </q>	ebay </q>
	pornstar forum </q>	pornstar com </q>
	pornstar volt usa </q>	ebay usa com </q>
	ebay jr com </q>	pornstar autotrader com </q>
	pornstar forum usa </q>	pornstar on line </q>
	pornstar volt international </q>	ebay store usa </q>
	pornstar classic usa </q>	pornstar volt company </q>
	ebay motors usa </q>	pornstar volt com </q>
	pornstar classic international </q>	pornstar store usa </q>
	pornstar classic classic </q>	pornstar video codes </q>
	ebay jr usa </q>	ebay motors com </q>
	pornstar volt co </q>	pornstar sports line </q>
	ebay usa usa </q>	

Table 5: Next-query prediction evaluation results for the vanilla HRED model.

Method	MRR	Δ %	MRR'[7]
ADJ	0.3191		0.5334
Baseline Ranker	0.5371	+68%	0.5563
BR+ HRED	**0.5397**	+69% / +0.0%	**0.5749**

Column four shows the MRR values of the original paper we are replicating

Table 6: Robust prediction results. next-query prediction evaluation results for the vanilla HRED model.

Method	MRR	Δ %	MRR'
ADJ	0.3205		0.4507
Baseline Ranker	0.5324	+66%	0.4831
BR+HRED	**0.5478**	+67% / +0.1%	**0.5309**

Column four shows the MRR values of the original paper we are replicating

Table 7: Long-tail prediction results. next-query prediction evaluation results for the vanilla HRED model.

Method	MRR	Δ %	MRR'
ADJ	0.1542		0.3830
Baseline Ranker	0.4426	+187.0%	0.6788
BR+HRED	**0.4310**	+179.0% / -2.0%	**0.7112**

Column four shows the MRR values of the original paper we are replicating

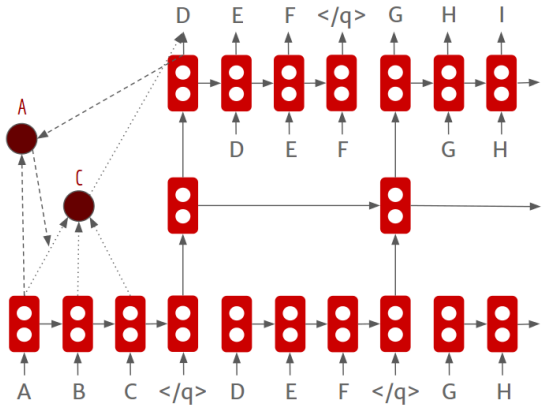

Figure 6: Schematic structure of the attention HRED that is used for query suggestion. For clarity reasons only one arrow from the attention module to a query encoder state is drawn. In reality the attention module computes a score for every hidden state (A, B and C in this example).

3.3 Results

We use the same dataset as described in section 2. We train our model for 36 hours for 250,000 iterations. Im this section we first present the quantitative results, followed by some example samples that were produced by the attention model.

3.3.1 Quantitative results

In table 8 we can see the quantitative results for the LambdaMART evaluation for the attention model. In the table the scores of the vanilla model are present as well so that we can compare them to our attention extension. We can observe that the HRED score of the attention model does not add any value when ranking the query suggestions (we followed the same procedure as for the vanilla model). We observe almost the same results for the vanilla model and for the model with attention.

Table 8: Model with attention prediction results. next-query prediction evaluation results for the vanilla HRED model.

Method	MRR	Δ %	MRR'
ADJ	0.3171		0.3191
Baseline Ranker	0.5400	+70.0%	0.5371
BR+HRED	**0.5381**	+69.7% / +0.0%	**0.5397**

Column four shows the MRR values of the vanilla model

3.3.2 Qualitative Results

Introspection of the output shows that for some queries attention seems to have the expected effect. Yet, not on all queries this is as clear. In table 9 we present the same queries as presented in table 4, i.e. samples that were produced under the vanilla model and we compare the two. The samples in the table clearly show some differences between the vanilla samples and the samples that were derived from the attention model. Whereas sample 1 produces many query suggestions that contained the words *cruise* and *lines* in the vanilla HRED model, the type of words that are sampled are shifted in the attention HRED module. In the latter words like *tickets*, *airline* and *flight* appear. These are words that could be related to cruises, yet not necessarily as much as the words sampled in the vanilla model. Adding attention to the second sample looks more promising. The sampled queries are longer and the query word *school* seems to have received more attention during the computation of the next query, as more words like *school* and *university* appear in table 9 than in 4. As for the third sample, whereas the suggestions in the vanilla model often contained the word *lake*, that also appeared in the query, in the attention model this word has disappeared. Instead, the word *florida* receives most of the attention. The fourth example is an example where the query clearly does not seem to benefit from the addition of attention. Not only are very different words sampled than in the vanilla model, also the semantic relation of the words to the query seems questionable.

4. HRED WITH CHARACTER N-GRAMS AS INPUT

In the vanilla HRED model, the input data was provided as word tokens. To limit the output dimensionality of the model we used a fixed vocabulary of the 50,000 most frequent words in the background data set, instead of all the unique words in the background set. As a consequence all words in the train, test and validation set that are not present in the vocabulary are replaced with the <unk> token. The extension we describe in this section will use character n-gram tokens instead of words as the smallest units that our model input and output is composed of. These character n-grams are built without explicit word segmentation[3] so we incorporate the "space" character in out n-grams as a separate character. As a result the vocabulary reduces from 50,000 words to 540 character n-grams.

4.1 Key Idea

The distribution of the words in the background data set follows Zipf's law [18]. This means that there are a few words that often occur in the data set and many words that rarely occur. To make the distribution of words in our vocabulary closer to uniform and to make it possible to predict query words that are not present in the background set, we decide to use character n-gram representations instead of words.

Machine translation gained a significant improvement by using a character-based unlimited vocabulary instead of a fixed word vocabulary [8]. The character n-gram approach has a few advantages. First of all, by using a vocabulary consisting of character n-grams only we can use a considerably smaller vocabulary. This is beneficial for the computation cost during learning and inference. When predicting the next query, the beam search algorithm has to calculate a softmax score for every word in our vocabulary. By reducing the vocabulary from 50,000 words to 540 character n-grams (see implementation details in section 4.2), this computation will be less expensive.

As a second advantage, using character n-grams instead of words excludes the possibility to predict the <unk> token. That is, as we include all possible unigrams in our n-gram vocabulary, we can always build our query word with single unigrams. In contrast to this when the vanilla HRED model receives a misspelled word as input, this word will most likely be translated into the <unk> token, because it is not present in our 50,000 word vocabulary. As a result, the model will produce incorrect next query candidates for the misspelled input. Using character n-grams instead of words will partially solve this problem, as we can still make a representation for the misspelled word.

The use of character n-grams instead of words also has a few disadvantages. First of all, as character n-grams are smaller chunks of text than regular words, the unrolled RNN will be much deeper than in the original approach. We raise the maximum sequence length to 256 (as opposed to 50 in the vanilla HRED) in order to be able to compose full queries. Therefor, the vanishing gradient problem can negatively effect the learning procedure of the lower level layers in the network. Another disadvantage is that when predicting a new query we need to reformulate our beam search algorithm. Because a query-string could be composed of different character n-gram combinations, we need to recombine suggestions when similar query-strings are composed out of different n-gram combinations.

4.2 Implementation

In order to construct our character n-gram distribution we first go over the entire background set and extract all tri-, bi- and unigrams. The result is a frequency distribution over all character n-grams. Second, we extract the 200 most common trigrams, 300 most common bigrams and all the unigrams. Then, we combine these n-grams into a new vocabulary. This way, we ensure that our vocabulary follows a more uniform distribution, instead of the previously mentioned Zipf's law distribution.

Next, we translate all the queries in our train, test and validation set into character n-gram representations. For every query string of length k $Q_{1:k}$, we start with a substring $Q_{i:n}$ of length 3. This is the first n-gram of length $n = 3$ (trigram) and $i = 1$. If this trigram is in our character n-gram vocabulary of most common tri-, bi- and unigrams, we translate this trigram to an index token. If not, we continue with the bigram $Q_{i:n}$ where $n = 2$ and $i = 1$. If this bigram is in the distribution we use this bigram, otherwise we use the unigram $Q_{i:n}$, where $n = 1$ and $i = 1$. Next, we raise i by one and continue with the same procedure until $i = k$. We do this for all the queries in the data set. Spaces are included in the n-grams.

When taking the 200 most common trigrams, 300 bigrams and all the unigrams, the final vocabulary contains 540 n-grams. Finally we train the vanilla HRED model with this n-gram query representation. Instead of predicting words to create next query suggestions, we now predict character n-grams.

4.3 Results

We only adapt the vanilla model with the new character n-gram representation. That is, we do not use the attention extension here. The model has been trained for 24 hours.

4.3.1 Qualitative Results

In table 10 we provide a few output samples for the model trained on character n-grams. We can see that the model does not generate useful and accurate query suggestions. Due to these results we decide to stop the character n-gram model training before it had run as long as our previously trained models. Therefor we only examine the qualitative results of our model here and we do not conduct all the quantitative experiments that we performed for the vanilla model.

Table 10: Some samples of query suggestions produced by the n-gram HRED model

Input Query	Suggestion Query
labbusas orglabusas org	wwwy com
	www com
	wwwerm
coin worldconecac	wwwy com
	www com
	wwwerm
yahoo mail	wwwy com
	www com
	wwwerm

There are multiple explanations why our n-gram model is not performing as well as the vanilla model. Within the NLP community there is broad consensus on that n-gram

models should train much longer than models that use a fixed word vocabulary. Due to the poor intermediate results mentioned before and time limitations, we have decided to prioritize the training of the other two models. Note that it is not certain that an extended training procedure would have resulted in a better performance of the model.

It has been recently shown that attention can be very beneficial to character-based models [3], and even for hierarchical models [6]. A logical step for future work would be to extend the character-based HRED model with the attention architecture that we outline in section 3.

5. DISCUSSION AND FUTURE WORK

In this section we will discuss various problems we encountered, some of which we solved. We will also give several recommendations for future work.

5.1 Encountered problems

During the replication of [13] and implementation of HRED using Tensorflow we encountered various problems.

The first of these problems had to do with batch creation. Before grouping batches, as described in section 2.2.5, our model showed unexpected cost increase on the test set. Randomized batches should not have given this cost increase since we masked the padding tokens, however this still happened. After switching to the grouped batches, our results improved drastically. However one would expect that grouped batches are more correlated and should thus perform slightly worse, if you discount the unneeded computation over the extra padding symbols. This was not the case, and this might indicate an error in our implementation.

As described in the quantitative results 2.3, we were not able to replicate the difference in MRR values for the BR and BRplusHRED mode. This could be due to an error in the implementation of the HRED model, or in our experimental pipeline. We also trained the original implementation of [13] and repeated the experiments. These give similar results, which are different from the original results. One explanation is a slight error somewhere in our experimental pipeline, or the difference could be attributed to the different preprocessing used.

Another problem we encountered often during training has to do with unexpected floating point calculations. During the Adam optimization procedure, weights became NaN (Not a number) or ∞ after an iteration and the loss spiked to enormous heights for such a specific iteration. This happened even though we clipped the gradients to prevent them from exploding. During training we ignored iterations where the loss became NaN, ∞ or an unusually high number. Despite this check, we still think these errors might have compromised the training procedure. We repeated some small training experiments on a NVIDIA Tesla K40 GPU instead of our NVIDIA GeForce GTX 980Ti GPU, and these types of errors almost stopped occurring. NVIDIA's consumer GPUs might be more vulnerable for these types of miscalculations. We were unable to run all experiments on a more resilient NVIDIA Tesla K40 GPU due to our limited supercomputer quota.

Our character n-gram extension does produce very similar nonsensical results each time we try to generate query suggestions from a given sentence—see table 10. This is not what we expected. The cost of the model has converged. We think this might have to do with a miscalculation during training, or it might be due a slight error in the processing of the n-gram data. Another theory is that we gave the model not enough flexibility and breathing room to model the complexer structure of words with their character n-grams. It might have been better to train such a model with larger dimensions for the hidden layers. The problem with such an architecture is that the difficulty of fitting it into memory which was limited by the 6GB the NVIDIA GeForce GTX 980Ti GPU provided.

5.2 Future Work

In future work we could explore different architectures and different data processing techniques.

Attention is a technique to shorten the distance between the encoded and decoded tokens. It has been shown that reversing the encoder input could be beneficial in NMT [15]. Moreover, state-of-the-art NMT models [12] use a bi-directional encoder where separate layers encode the default and reversed input sequences. We could apply such an optimization to our HRED model as well, by introducing a bi-directional query encoder. The same principle could sadly not be applied to the session encoder. This is because it would loose the recurrent aspect of our HRED model, i.e. it could not train multiple decoded queries in one session anymore.

It has also been shown that multiple LSTM or GRU-layers could be beneficial for each component of the network[15]. This would mean that for each of our components (query encoder, session encoder, decoder) we could stack multiple GRU-layers instead of just using one layer.

The current model is trained without any form of regularization. L_2-regularization could be applied to the output layer. Dropout[14] has been shown to be a great regularizer as well, for example for language modeling tasks [17]. This is a relatively small modification to our architecture but could yield large performance improvements.

Another strategy to improve our final model's performance is applying early stopping. By checking our model against a validation set at certain intervals, we could prevent overfitting when our model has converged.

Because the HRED model's session-encoder has similar recurrent properties as a language model we could try to exploit these properties even further. One way to do this is by using very long sessions, and sliding over them during training. This approach is similar to sliding a window over words in a language model.

Now that we have implemented a character-based HRED model and a HRED model with attention a logical next step is to combine those. It has been shown that attention can be very beneficial to character-based models [3], and recently even for hierarchical models [6]. In the latter, an NMT system was built with a word-encoder layer on top of a character-encoder layer. In this work the attention was placed on the second RNN layer, where in our work it is placed on the lowest RNN layer. Another possible modification we could make is attending over both layers, and see if the results differ.

During preprocessing we could have applied a lot more data cleansing techniques. Some of the sessions are highly repetitive, and a lot of queries are on keywords such as: "ebay", "google", "yahoo" which are not very informative. One approach to improve our model is to start with cleaner input data, manually cleaning it even further could probably

be beneficial. The model should learn not to suggest a query that has already been tried.

6. CONCLUSION

In this study we have replicated the work of [13]. The results sampled from our model look very satisfying. The query suggestions are very semantically related to the user query. The quantitative results are not so satisfying yet. We expect a small flaw in our implementation there. We also presented two extensions to the original model. First we implemented attention, that has been shown to be very beneficial in Neural Machine Translation [1, 9]. The quantitative results show no improvement, but again, we expect a small flaw in our implementation there. On the other hand, the samples show that attention shifts the focus to specific parts of the input query. It is to be said that the attention extension still needs to be refined, to obtain better results. As a second extension we switched to character n-grams instead of words as the building blocks of our queries, to reduce the sparsity of the input data. This change has not obtained better results yet.

7. REFERENCES

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

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

[3] J. Chung, K. Cho, and Y. Bengio. A character-level decoder without explicit segmentation for neural machine translation. *arXiv preprint arXiv:1603.06147*, 2016.

[4] Q. He, D. Jiang, Z. Liao, S. C. Hoi, K. Chang, E.-P. Lim, and H. Li. Web query recommendation via sequential query prediction. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1443–1454. IEEE, 2009.

[5] J.-Y. Jiang, Y.-Y. Ke, P.-Y. Chien, and P.-J. Cheng. Learning user reformulation behavior for query auto-completion. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 445–454. ACM, 2014.

[6] A. R. Johansen, J. M. Hansen, E. K. Obeid, C. K. Sønderby, and O. Winther. Neural machine translation with characters and hierarchical encoding. *arXiv preprint arXiv:1610.06550*, 2016.

[7] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] W. Ling, I. Trancoso, C. Dyer, and A. W. Black. Character-based neural machine translation. *arXiv preprint arXiv:1511.04586*, 2015.

[9] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[10] B. Mitra. Exploring session context using distributed representations of queries and reformulations. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3–12. ACM, 2015.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[12] R. Sennrich, B. Haddow, and A. Birch. Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*, 2016.

[13] A. Sordoni, Y. Bengio, H. Vahabi, C. Lioma, J. Grue Simonsen, and J.-Y. Nie. A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 553–562. ACM, 2015.

[14] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[15] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[16] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[17] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

[18] G. K. Zipf. Selected studies of the principle of relative frequency in language. 1932.

Table 9: Some samples of query suggestions produced by the attention HRED model.

Sample 1		
Input	</q> princess cruise lines </q> princess cruise lines </q>	
Output	ebay </q>	orbitz </q>
	craigslist </q>	travelocity </q>
	expedia </q>	directions </q>
	google </q>	espn </q>
	jetblue </q>	yahoo </q>
	american airlines </q>	yellow pages </q>
	american idol </q>	southwest airlines </q>
	airline tickets </q>	jet blue book </q>
	cheap airline tickets </q>	american airline tickets </q>
	southwest airline tickets </q>	american airlines tickets </q>
	cheap tickets tickets </q>	cheap flights tickets </q>
	american express tickets </q>	cheap tickets airlines </q>
	american idol tickets </q>	
Sample 2		
Input	</q> st charles north high school </q>	
	st charles north high school </q>	
Output	google </q>	mapquest </q>
	ebay </q>	yahoo </q>
	google com </q>	google maps </q>
	mapquest com </q>	google mail </q>
	google in </q>	the city </q>
	st louis florida </q>	st george florida </q>
	st john school </q>	st george county florida </q>
	st george county america </q>	google new york florida </q>
	google new york university </q>	st louis county florida </q>
	st st louis florida </q>	st george county school </q>
	st george school school </q>	st john school school </q>
	st louis florida state university </q>	st george county state university </q>
	google new york state university </q>	
Sample 3		
Input	</q> lake <unk> florida </q> lake <unk> florida </q>	
Output	google </q>	mapquest </q>
	google com </q>	google ca </q>
	www google </q>	google in </q>
	new york florida </q>	new york university </q>
	www google com </q>	orange county florida </q>
	florida county florida </q>	lake county florida </q>
	florida of florida </q>	houston county florida </q>
	florida state florida </q>	washington county florida </q>
	new jersey florida </q>	new jersey university </q>
	kentucky county florida </q>	new york college </q>
	georgia county florida </q>	miami county florida </q>
	google in florida </q>	san diego university </q>
	new york state </q>	
Sample 4		
Input	</q> pornstar dynamite </q> pornstar dynamite </q>	
	pornstar dynamite </q> pornstar dynamite	
	</q> pornstar dynamite </q>	
Output	mapquest </q>	google </q>
	jeeves </q>	ebay </q>
	drudge </q>	switzerland </q>
	vitamin </q>	www </q>
	mapquest com </q>	google com </q>
	ebay com </q>	kathy noble </q>
	peace and </q>	kathy things </q>
	switzerland and </q>	beer and </q>
	vitamin and </q>	drudge and </q>
	ebay for </q>	turkey and </q>
	pope and </q>	bio and </q>
	mammoth and </q>	blues and </q>
	drudge of </q>	