Realtime query completion via deep language models

Po-Wei Wang[*]
Machine Learning Dept
Carnegie Mellon University
Pittsburgh, PA, United States
poweiw@cs.cmu.edu

Huan Zhang[*]
Electrical and Computer Engineering
UC Davis
Davis, CA, United States
ecezhang@ucdavis.edu

Vijai Mohan
Amazon
Palo Alto, CA, United States
vijaim@amazon.com

Inderjit S. Dhillon[*]
Amazon & UT Austin
Palo Alto, CA, United States
isd@amazon.com

J. Zico Kolter
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, United States
zkolter@cs.cmu.edu

ABSTRACT

Search engine users nowadays heavily depend on query completion and correction to shape their queries. Typically, the completion is done by database lookup which does not understand the context and cannot generalize to prefixes not in the database. In this paper, we propose to use unsupervised deep language models to complete and correct the queries given an arbitrary prefix. We address two main challenges that renders this method practical for large-scale deployment: 1) we propose a modified beam search process which integrates with a *completion distance* based error correction model, combining the error correction process (as a potential function) together with the language model; and 2) we show how to efficiently perform our modified beam search process on CPU to complete the queries with error correction in real time, by exploiting the greatly overlapped forward propagation process and conducting amortized dynamic programming on the search tree, along with both SIMD-level and thread level parallelism. We outperform the off-the-shelf Keras implementation by a factor of 50, thus allowing us to generate query suggestions in real time (generating top 16 completions within 16 ms). Experiments on two large scale datasets from AOL and Amazon.com show that the method substantially increases hit rate over standard approaches, reduces the memory footprint of database lookup based approach by over two orders of magnitude, and is capable of handling tail queries.

KEYWORDS

Query completion, query correction, deep learning, realtime

ACM Reference Format:
Po-Wei Wang, Huan Zhang, Vijai Mohan, Inderjit S. Dhillon, and J. Zico Kolter. 2018. Realtime query completion via deep language models. In *Proceedings of SIGIR Workshop On eCommerce (SIGIR eCom'18).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.475/123_4

[*]Work performed while at A9.com, an Amazon subsidiary

Copyright © 2018 by the paper's authors. Copying permitted for private and academic purposes. In: J. Degenhardt, G. Di Fabbrizio, S. Kallumadi, M. Kumar, Y.-C. Lin, A. Trotman, H. Zhao (eds.): Proceedings of the SIGIR 2018 eCom workshop, 12 July, 2018, Ann Arbor, Michigan, USA, published at http://ceur-ws.org

CEUR-WS.org/Vol-2319/paper24.pdf

1 INTRODUCTION

Search completion is the problem of taking the prefix of a search query from a user and generating several candidate completions. This problem has enormous potential utility and monetary value to any search provider: the more accurately an engine can find the desired completions for a user (or indeed, potentially steer the user towards high-value completions), the more quickly it can lead the user to their desired goal.

This paper proposes a realtime search completion architecture based upon deep character-level language models. The basic idea is that instead of looking up possible completions from a generic database, we perform search under a deep-network-based language model to find the most likely completions of the user's current input. This allows us to integrate the power of deep language models, that have been shown to perform extremely well on complex language modeling and prediction tasks, with the desired goal of finding a good completion. Although this is a conceptually simple strategy (and one which has been considered before, as we highlight below in the literature survey), there are two key elements required to make this of practical use for a search engine provider, which together make up the primary technical contributions of the paper: 1) The completion must be *error correcting*, able to handle small errors in the user's initial input and provide completions for the most likely "correct" input. We propose such an approach that combines a character-level language model with an edit-distance-based potential function, combining the two using a tree-based beam search algorithm; 2) The completion must be *realtime*, able to produce high-quality potential completions in time that is not even perceivable to the user. We achieve this by developing an efficient tree-based version of beam search and an amortized dynamic programming algorithm for error correction based on *completion distance* (our proposed editing distance variant) along the search tree, exploiting thread-level and SIMD-level CPU-based computation for a single query, and through numerous optimizations to the implementation that we discuss below.

We evaluate the method on the AOL search dataset, a dataset consisting of over 36 million total search queries, as well as on an Amazon product search dataset containing over 100 million user search queries. Our proposed method substantially outperforms highly optimized standard search completion algorithms in terms of its hit rate (the benefit of the deep language model and the

error correction), while being fast enough to execute in real time for search engines. Our approach is also very memory efficient and reduces the memory usage of database lookup based query completion system by at least two orders of magnitude; in addition, we can handle *tail queries*, which are the queries that are rarely seen and for which database lookup based approach cannot give any query completions. The experiments on AOL search dataset and code are publicly available online [1].

2 RELATED WORK

2.1 Background on search completion

Here we review existing approaches to search query completion and error correction. Broadly speaking, two types of query completions are most relevant to our work, *database lookup* methods and *learning-based* approaches.

Database Lookup. One of the most intuitive ways to do query completion is to do a database lookup. That is, given a prefix, we can fetch all the known queries matching the prefix and return the most frequent candidates. This is called the "most popular completion" (MPC) [1], which corresponds to the maximum likelihood estimator for P(completion | prefix). The database lookup can be efficiently implemented by a trie [9]. For instance, it takes only 15μs to give 16 suggestions for a query in our own trie-based implementation. However, due to the long-tail nature [20] of the search queries, many prefixes might not exist in the database; for example, in the AOL search data, 28% of the queries are unique. An excellent survey of these current "classical" approaches is given in Cai et al. [3].

Learning-based. In addition to database lookup approaches, in recent years there have been a number of approaches that use learning-based methods for query completion. Sordoni et al. [19] use a translation model at the word level to output single-word search query suggestions, and also model consecutive sessions of the same user. Liu et al. [12] proposed a word-based method for code completion, but focused solely on greedy stochastic sampling for the prediction. Mitra and Craswell [14] also used neural networks combined with a database-based model to handle tail queries, but focused on CNN approaches that just output the single most likely word-level completion. Shokouhi [18] used logistic regression to learn a personalized query ranking model, specific to individual users. All these approaches are relevant but fairly orthogonal to our own, as we focus here on character-level modeling, beam search, and realtime completion. Finally, Park and Chiba [15] very recently published an approach similar to ours, which uses a character-level language model for completion. But their approach focuses on the use of embeddings (such as word2vec) to produce "intelligent" completions that make use of additional context, and the approach does not handle error correction; they also do not report the prediction time of their completions, which is a key driver for our work.

2.2 Error correction for queries

Our work also relates to methods on error and spelling correction approaches, which again are roughly divided into heuristic models and learning-based approaches.

[1] https://github.com/xflash96/query_completion

Heuristic models. Whitelaw et al. [22] proposed generating candidate sets that contain common errors for given prefixes, then searching these based upon the current query. Similarly, Martins and Silva [13] use a ternary search tree to accelerate the search within candidate sets for spelling correction in general. The approaches are nice in that they are easily parallelizable at runtime, but are relatively "brute force", and cannot handle previously unseen permutations.

Learning-based Model. On the learning side, Duan and Hsu [6] train an n-gram Markov model combined with A* search to determine candidate misspelling; this is similar to our approach except with a much richer language model replacing the simple n-gram model, which creates several challenges in the search procedure itself. Likewise, Xie et al. [23] use a similar character-level model with attention, but do so in the context of error correcting an entire paragraph of text, and don't focus on the same realtime aspects that we do.

3 BACKGROUND ON QUERY COMPLETION

When a user types any prefix string s in the search engine, the query completion function will start to recommend the best r completions, each denoted $\hat{s}$, according to certain metrics. For example, one might want to maximize the probability that a recommendation is clicked. The conditional probability can be formulated as

$$P(\hat{s} \mid s) := P(\text{completion} \mid \text{prefix}), \tag{1}$$

and the goal of query completion in the setting is to find the top r most probable strings $\hat{s}$ which potentially also maximize some additional metric, such as the click-through rate.

Denote $s_{1:m}$ as the first m characters in string s. We first discuss the query completion in a simplified setting, in which all completions must contain the prefix exactly, that is: $\hat{s}_{1:m} = s_{1:m}$, and

$$P(\hat{s}_{1:n} \mid s_{1:m}) = P(\hat{s}_{m+1:n} \mid s_{1:m}) = P(\hat{s}_{m+1:n} \mid \hat{s}_{1:m}), \tag{2}$$

where n is the total length of a completion. Note that the probability is defined in the sequence domain, which contains exponentially many candidate strings. To simplify the model, we can apply the conditional probability formula recursively and have

$$P(\hat{s}_{m+1:n} \mid \hat{s}_{1:m}) = \prod_{t=m}^{n-1} P(\hat{s}_{t+1} \mid \hat{s}_{1:t}). \tag{3}$$

This way, we only need to model $P(\hat{s}_{t+1} \mid \hat{s}_{1:t})$, that is, the probability of the next character under the current prefix. This is precisely a character-level language model, and we can learn it in an unsupervised manner using a variety of methods, though here we focus on the popular approach of using recurrent neural networks (RNNs) for this character-level language model. Character-level models are the right fidelity for the search completion task, because they satisfy the customer's expectation from the user interface, and additionally, word-level models or sequence-to-sequence probabilities would not be able to model probabilities under all partial strings.

3.1 The Unsupervised Language Model

We first focus on the language model term $P(\hat{s}_{t+1} \mid \hat{s}_{1:t})$, the probability of next character under the current prefix. RNNs in general, and variants like long short term memory networks (LSTMs) [8], are

extremely popular for high-fidelity character level modeling, and achieve state-of-the-art performance for a number of datasets [5]. Since they can be trained from unsupervised data (e.g., just datasets of many unannotated search queries), we can easily adapt the model to whatever terms users are *actually* searching for in the dataset, with the potential to adapt to new searches, products, etc, simply by occasionally retraining the model on all data collected up to the current point.

Although character-level language modeling is a fairly standard approach, we briefly highlight the model we use for completeness. Consider a recurrent neural network with hidden state $\boldsymbol{h}_t$ at time t. We want to encode the prefix $\hat{s}_{1:t}$ and predict the next character using $\boldsymbol{h}_t$. We follow fairly standard approaches here and use an LSTM model, in particular the specific implementation from the Keras library [4][2], which is defined by the recurrences

$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right), \tag{4}$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right), \tag{5}$$

$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right), \tag{6}$$

$$c_t = i_t \odot \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) + f_t \odot c_{t-1}, \tag{7}$$

$$h_t = o_t \odot \tanh\left(c_t\right), \tag{8}$$

in which $h_t, b \in \mathbb{R}^d$, $x_t \in \mathbb{R}^{|C|}$, $\forall t$, $W_{xi}, W_{xf}, W_{xo}, W_{xc}$ and $W_{hi}, W_{hf}, W_{ho}, W_{hc}$ are the forward kernel and recurrent kernel with corresponding dimensions, σ is the sigmoid activation function, and $\odot$ is the element-wise product. We use a one-hot encoding of characters as input, a two-layer LSTM with 256 to 1024 hidden units (more discussion on these choices below), and for prediction of character $\hat{s}_{t+1}$, we feed the hidden layer $\boldsymbol{h}_t$ to a softmax function

$$P\left(\hat{s}_{t+1} = i \mid \hat{s}_{1:t}\right) = \text{softmax}\left(i;\ W_{\text{softmax}}\boldsymbol{h}_t\right) = \frac{\exp(w_i^T\boldsymbol{h}_t)}{\sum_{j=1}^{|C|}\exp(w_j^T\boldsymbol{h}_t)}, \tag{9}$$

for all i in the character set C and train the language model to maximize the log likelihood (minimize the categorical cross-entropy loss),

$$\underset{W}{\text{minimize}}\ -\sum_{s\in S}\frac{n_s}{|s|}\sum_{t=1}^{|s|}\log P(s_{t+1} \mid s_{1:t}), \tag{10}$$

where S denotes the set of queries, $|s|$ is the length of query s and n_s is the number of times query s appears in the dataset. Further, we pad all queries with an end-of-sequence symbol to predict whether the query is complete.

3.2 Stochastic Search and Beam Search

Once we have the language model, we can evaluate the probability $P\left(\hat{s}_{m+1:n} \mid \hat{s}_{1:m}\right)$ for any prefix $\hat{s}_{1:m}$, but would ideally like to find the completion with the highest probability. Enumerating all the possible strings is not an option because we have exponentially many candidates. Indeed, finding the best sequence probability, which is called the "decoding problem", is NP-hard [7], so we have to rely on approximations.

[2]Note that, as we describe below, we won't actually use the Keras library at prediction time, but we do use it for training

The most naive way to do so is simply via sampling: we sample the next character (according to its probability of occurrence) given the current prefix, until we hit an end-of-sequence (EOS) symbol:

For $t = m$**;** **;** t++ **:**
  $\hat{s}_{t+1} \sim P(\hat{s}_{t+1} \mid \hat{s}_{1:t})$;
  If $\hat{s}_{t+1}$== EOS **: break**;

This method produces output that looks intuitively reasonable. However, it is biased toward longer sequences (as we can possibly miss the EOS symbol even if it has a relatively large probability) with short-term dependencies and clearly does not generate the most probable sequences, because sampling in a greedy fashion is clearly not the same as sampling from the sequence space.

That is, we really need to do a better approximate search to get better results. One classic way to do this is to perform beam search, that is, perform breadth-first search while keeping the top-r candidates. We illustrate the algorithm as follows:

cand := $\{s_{1:m} : 0\}$, result := { }
For $t = m$**;** cand is not empty**;** t++**:**

$$\text{cand}_{\text{new}} := \left\{\begin{array}{l} s_{1:t+1} : \log P\left(s_{1:t+1} \mid s_{1:m}\right) \\ \qquad \text{for all } s_{t+1} \in C,\ \text{for all } s_{1:t} \in \text{cand} \end{array}\right\};$$

  cand := the most probable ($r - |\text{result}|$) candidates in cand_{new};
  Move $s_{1:t+1}$ from cand to result if s_{t+1} is EOS symbol;

By performing beam search we can consistently obtain a more probable set of completions compared to stochastic search.

However, there are two issues with the above method. First, it does not handle error correction (which is necessary for any practical type of completion) since the completion always attempts to find sequences that fit the current prefix exactly.[3] Second, as we show below, a naive implementation of this model is extremely slow, often taking on the order of one second to produce 16 completions for a given prefix. Thus, in the next two sections, we present our primary technical contributions, which address both these issues.

4 COMPLETION WITH ERROR CORRECTION

Most of the time, query completion is more than completing over a fixed prefix. The input prefix might contain mistakes and sometimes we would also like to insert keywords in the prefix. Traditionally, the database community handles the two features by first doing a pass of error correction by matching the input to a typo database generated by permuting characters, then matching the database again on the permuted terms for insertion completion [17, chap. 14]. Our observation is that with a language-model-based approach, we can handle the spelling correction and insertion completion all in one model.

4.1 Error correction via the noisy channel model

Following the convention in the previous section, define $s_{1:m}$ and $\hat{s}_{1:n}$ to be the prefix and completion string of length m and n, respectively. Different from the previous section, we no longer constrain

[3]A character-level LSTM alone only gives the probability of the input/completion sequence. It could not control the trade-off between the probability of user typos and the likelihood of a completion, thus an error model is necessary.

the beginning of completions to be identical to the prefix so that we can "correct" the user input. Thus, the problem of finding the most probable completion becomes

$$\arg\max_{\hat{s}_{1:n}} P\left(\hat{s}_{1:n} \mid s_{1:m}\right) \tag{11}$$

Now let us derive the maximum-a-posteriori (MAP) estimate of the above problem. Using Bayes' theorem, (11) can be rewritten as

$$\arg\max_{\hat{s}_{1:n}} \frac{P(s_{1:m} \mid \hat{s}_{1:n})P(\hat{s}_{1:n})}{P(s_{1:m})}. \tag{12}$$

Because $s_{1:m}$ never changes, $P(s_{1:m})$ can be considered as a constant. Thus, solving (12) is equivalent to maximizing the following:

$$\arg\max_{\hat{s}_{1:n}} \ \log P(s_{1:m} \mid \hat{s}_{1:n}) + \log P(\hat{s}_{1:n}). \tag{13}$$

This MAP estimate is called the noisy channel model [2, 10] in NLP, in which the first part $\log P(s_{1:m} \mid \hat{s}_{1:n})$ models the noisy channel of user inputs, and the second part $\log P(\hat{s}_{1:n})$ models the prior. For example, when using the noisy channel model for error correction in a paragraph, we can assume that users have a constant probability to make a typo for each letter. Under such assumption, the noisy channel $\log P(s_{1:m} \mid \hat{s}_{1:n})$ is proportional to the edit distance (Levenshtein distance). For the prior part, we can plug in whatever $P(\hat{s}_{1:n})$ we have for the paragraph, like the n-gram transitional probability or the language model. However, error correction for queries is essentially different from that for paragraphs; in query completion the user inputs are always incomplete. Thus, we must perform the completion and error correction at the same time. One consequence of such a constraint is that we can no longer use the edit distance function directly.

4.2 Edit Distance v.s. Completion Distance

The edit distance function, which returns the minimum changes (add/substitute/delete) to transform one string into another, is a natural candidate to measure the number of corrections between user inputs and completions. Assume that the probability by which users make an error is constant, like 2%. As we mentioned before, the noisy channel under such an assumption can be written as

$$\log P\left(s_{1:m} \mid \hat{s}_{1:n}\right) = -\alpha \cdot \text{edit distance}(s_{1:m},\ \hat{s}_{1:n}), \tag{14}$$

where $\alpha = -\log 2\%$. Note that to handle incomplete prefix and insertion completion, we should not incur penalties for the completions. That is, we should not count the edit distance for adding words after the last character (of terms) from the user input. This can be done by modifying the transition function in the edit distance algorithm. To be specific, we change the penalty to an indicator when dealing with the "add" operation in the edit distance algorithm; we define the new transition function to be

$$\text{dist}_{\text{new}}(\text{j}) = \min \begin{cases} \text{dist}_{\text{new}}(\text{j-1}) + \mathcal{I}\left(s_{j-1} \neq \text{last char}\right) & \text{add;} \\ \text{dist}_{\text{compl}}(\text{j-1}) + 1 & \text{substitute;} \\ \text{dist}_{\text{compl}}(\text{j}) + 1 & \text{delete;} \end{cases} \tag{15}$$

We called the new edit distance function a "completion distance", in a way that the completion "pokemon go plus" for the prefix "poke go" would not incur unwanted penalties, because the added characters are proper completions (only append character after terms).

To perform error correction under the noisy channel model, we still need to integrate the noisy channel (distance function) with our LSTM-based language model (the prior), which can only be evaluated once in the forward direction because of the beam search procedure. Recall that the dynamic programming algorithm [21] of edit (completion) distance costs $O(m \cdot t)$ to compare two strings of length m and t. If we apply the algorithm to every candidate in the beam search for the incremental length t which ranges from 1 to n, it would add $O(|C|rm \cdot n^2)$ overhead to the beam search procedure, where $|C|$ is the size of character set, r is the number of candidates we keep, and n is the length of the final completion. This overhead is not affordable, and we need to modify the dynamic programming algorithm for completion distance to amortize it on the search tree.

4.3 Amortized Dynamic Programming On the Search Tree

We can exploit the fact that every new candidate in the beam search procedure originates incrementally from a previous candidate. That is, only one character is changed. Thus, if we can maintain the last column in the completion distance algorithm, that is $\text{dist}_{\text{compl}}, \forall j$, for every candidate, we can save the repeated effort in building the edit distance table. The resulting algorithm is summarized below:

cand := {empty string " ": 0}, result := { }

For $t = 0$**;** cand is not empty**;** t++**:**

$$\text{cand}_{\text{new}} := \left\{ \begin{array}{c} \hat{s}_{1:t+1} : \ \log P\left(s_{1:m} \mid \hat{s}_{1:t+1}\right) + \log P\left(\hat{s}_{1:t+1}\right) \\ \text{for all } \ \hat{s}_{t+1} \in C, \ \text{for all } \ \hat{s}_{1:t} \in \text{cand} \end{array} \right\};$$

cand := the most probable ($r - |\text{result}|$) candidates in cand_{new};

Move $\hat{s}_{1:t+1}$ from cand to result if s_{t+1} is EOS symbol;

Maintain the last col of dist_{new} for $P(s_{1:m} \mid \hat{s}_{1:t})\ \forall \hat{s}_{1:t} \in$ cand;

By such bookkeeping, we are able to amortize the completion distance algorithm over the beam search procedure, making it n times faster (from $O(|C|rm \cdot n^2)$ to $O(|C|rm \cdot n)$).

4.4 Extensions

While we are using the simple assumption (2% user error rate) in this paper, the error correction algorithm can be generalized in various ways [10, chap. 5]. For example, we can plug in the frequency statistics in the transition function of edit distance [11] or learn it directly from the corpus [22]. Finally, we note that this idea of inserting a noisy channel model naturally generalizes to contexts other than edit distance. For example, many product search engines wish to drive the user not simply to a high-probability completion, but to a completion that is likely to lead to an actual sale. By modifying the prior probability to more heavily weight high-value completions, we can effectively optimize metrics other than simple completion probability using this approach.

5 REALTIME COMPLETION

Starting with the system as proposed previously, the key challenge that remains now is to perform such completions in real time. Response time is crucial for query completion because unless the user can see completions as they type the query, the results will likely

have very little value. The bar we set for ourselves in this work is to provide 16 candidate completions in about 20 milliseconds on current hardware.[4] The 20 milliseconds budget, combined with typical network latency, is similar to the pace that a user types in the query; we need to complete faster than typing to make our realtime completion usable in practice. Unfortunately, a naive implementation of beam search with the model trained above (using off-the-shelf implementations), requires more than one second to complete forward propagation through the network and beam search.

In this section, we now provide a detailed breakdown of how we have empirically improved this performance by a factor of over 50x in order to achieve sub-20-ms completion times.

5.1 LSTM over a Tree

First, we observe that all new candidates in the beam search process are extensions from the old candidates because of the BFS property. In this case, the forward propagations would greatly overlap. If we can maintain $\boldsymbol{h}_t$ for every old candidate, extending one character for new candidates would require only one forward propagation step. That is, we amortize the LSTM forward propagation over the search tree. The algorithm is illustrated below.

cand := $\{s_{1:m} : (\boldsymbol{h}_m,\ 0)\}$, result := { };

For $t = m$**;** cand is not empty**;** t++**:**

$$\text{cand}_{\text{new}} := \left\{ \begin{matrix} s_{1:t+1} : (\boldsymbol{h}_t, \log P\,(s_{1:t} \mid s_{1:m}) + \log P\,(s_{t+1} \mid s_{1:t})) \\ \text{for every } s_{t+1} \in C, \text{ for every } s_{1:t} \in \textit{cand} \end{matrix} \right\}$$

cand := the most probable $r - |\text{result}|$ candidates in cand_{new}

Move $s_{1:t+1}$ from cand to result if s_{t+1} is EOS symbol

Bump $\boldsymbol{h}_t$ to $\boldsymbol{h}_{t+1}$ by one step of LSTM on s_{t+1}, $\forall s_{1:t+1} \in$cand

Note that the initialization takes $O(md^2)$, and the four lines in the loop cost $O(r|C|d)$, $O(r|C|)$, $O(r)$, and $O(rd^2)$, where m is the length of $s_{1:m}$, d is the hidden dimension of LSTM, $|C|$ is the length of character set C, and r is the number of completions required. Using this approach, the complexity for computing r completions for d-dimensional LSTM reduces from $O(n^2rd(d+|C|))$ to $O(nrd(d+|C|))$ for sequence with maximum length n. A naive C implementation shows that the running time for such search drops to 250 ms from over 1 sec.

5.2 CPU implementation and LSTM tweaks

Although GPUs appear to be most suitable for computation in deep learning, for this particular application we found that the CPU is actually better suited to the task. This is due to the need for branching and maintaining relatively complex data structures in the beam search process, along with the integration of the edit distance computation. Thus, implementation on a GPU requires a process that frequently shuffles very small amounts of data (each new character) between the CPU and GPU and can be very inefficient. We thus implemented the entire beam search, error correction and forward propagation in C on the CPU.

However, after moving to a pure CPU implementation, it is the case that initially about 90% of the time is spent on computing the matrix-vector product in the LSTM. By properly moving to batch matrix-matrix operations with a minibatch that contains all r candidates maintained by beam search, we can substantially speed this up; By grouping together the product between the W matrices and $\boldsymbol{h}_t$ for all r candidates maintained by the beam search procedure, we can use matrix-matrix products, which have significantly better cache efficiency even on the CPU. We use the Intel MKL BLAS, and the total of these optimizations further reduces the running time to 75ms. By further parallelizing the updates via 8 OpenMP threads brings completion time down to 25 ms.

Finally, one of the most subtle but surprising speedups we attained was through a slightly tweaked LSTM implementation. With the optimizations above, computing the sigmoid terms in the LSTM actually took a surprisingly large 30% of the total computation time. This is due to the fact that 1) our LSTM implementation uses a hard sigmoid activation, which as a clipping operation requires branch prediction; and 2) the fact that the activations we need to apply the sigmoid to are not consecutive in the hidden state vector means we cannot perform fast vectorized operations. By simply grouping together the terms i_t, f_t, o_t in the hidden state, and by using Intel SSE-based operations for the hard sigmoid, we further reduce the completion time down to 13.3ms, or 16.3ms if we include the error correction procedure.

6 EXPERIMENTAL RESULTS

We evaluate our method on the AOL search dataset [16], a public dataset of real-world searches from 2006, as well as an internal dataset of product search queries from Amazon.com. The AOL dataset contains 36 million total queries, with 10 million of these being unique, illustrating the long tail in these search domains. We set a maximum sequence length for the queries at 60 characters, as this contained 99.5% of all queries. The Amazon dataset contains a random sample of about 110 million product search queries that users typed on the Amazon.com web site during 2017 (we excluded sexually explicit and culturally insensitive or inappropriate queries).

Training and testing splits. For each example in the dataset, we choose a random cutting point (always after two characters in the string), and treat all characters beforehand as the prefix and all characters afterwards as the completion. For examples in the validation and test set, we use these prefixes and actual completions to evaluate the completions that our method predicts. In the training set, we discard the cutting points and just train on the entire queries.

For the AOL dataset, we use a test set size of 330K queries, and use the rest for training. For the Amazon dataset we use a test set size of 1 million queries. We create training and testing splits to evaluate our method using two different strategies:

- Prefix splitting: sort the queries according to the MD5 hash of the prefix, and then split. This ensures that data in the test set does not contain an exact prefix match in the training set.
- Time splitting: For both the AOL and Amazon datasets, we sort the queries by timestamp and split. This mimics making predictions online as new data comes in.

[4] Experiments are carried on an Intel Xeon E5-2670 machine. We use up to 8 threads, and test it on the error-corrected query completion model with 512 hidden units. For each input, 16 suggestions are generated.

Table 1: Character-level language model cross-entropy loss (see (10)) for the LSTM on AOL search dataset and Amazon Product Search dataset. We explored both LSTM and GRU as the recurrent units and varied their dimensions from 256 to 1,024.

<table>
<tr><th rowspan="3">Dataset</th><th rowspan="3">Split</th><th colspan="6">Training Loss</th><th colspan="6">Validation loss</th></tr>
<tr><td colspan="3">LSTM</td><td colspan="3">GRU</td><td colspan="3">LSTM</td><td colspan="3">GRU</td></tr>
<tr><td>256</td><td>512</td><td>1024</td><td>256</td><td>512</td><td>1024</td><td>256</td><td>512</td><td>1024</td><td>256</td><td>512</td><td>1024</td></tr>
<tr><td rowspan="2">AOL</td><td>Prefix split</td><td>.07929</td><td>.0691</td><td>.06282</td><td>.07745</td><td>.06920</td><td>.06385</td><td>.07192</td><td>.06405</td><td>.05866</td><td>.07236</td><td>.06528</td><td>.06073</td></tr>
<tr><td>Time split</td><td>.07928</td><td>.0691</td><td>.06279</td><td>.07739</td><td>.06918</td><td>.06373</td><td>.07241</td><td>.06416</td><td>.05904</td><td>.07279</td><td>.06562</td><td>.06108</td></tr>
<tr><td rowspan="2">Amazon</td><td>Prefix split</td><td>.06635</td><td>.0583</td><td>.05275</td><td>.06463</td><td>.05819</td><td>.05356</td><td>.06099</td><td>.05463</td><td>.04997</td><td>.06116</td><td>.05562</td><td>.05175</td></tr>
<tr><td>Time split</td><td>.06624</td><td>.0580</td><td>.05308</td><td>.06454</td><td>.05832</td><td>.05428</td><td>.06076</td><td>.05424</td><td>.05008</td><td>.06058</td><td>.05497</td><td>.05133</td></tr>
</table>

6.1 Training language model

We trained our character-level language model on the characters of all the queries in the training set. We trained each model for 3-4 epochs over the entire dataset, and applied early stopping if the validation loss did not improve for more than 50,000 mini-batches. We used a 2-layer LSTM with 256, 512, 1024 hidden dimensions with dropout of 0.5 between the two LSTM layers (no dropout within a single layer), and used Adam optimizer to train with a mini-batch size of 256. We use cross-entropy loss weighted by query length for training. For each model, we select the learning rate from $\{10^{-2}, 10^{-3}, 10^{-4}\}$, weight decay from $\{10^{-7}, 10^{-8}, 10^{-9}, 0\}$ and gradient norm clipping from $\{0.01, 0.001, 0.0001, 0.00001\}$. We also conduct experiments on replacing LSTM with Gated Recurrent Unit (GRU), as GRU has lower computation cost compared to LSTM. Training and validation losses for each datasets, under the two different splittings and varying LSTM/GRU dimensions, are shown in Table 1. We observe that with the same model size, GRU usually shows slightly worse performance than LSTM, and increasing the hidden dimension does help in improving model performance.

Our training time on a NVIDIA V100 GPU (on AWS P3 instance) for the AOL dataset is 17, 18 and 24 hours for LSTM with 256, 512, and 1024 neurons, respectively. For the Amazon dataset, we remove all duplicate queries and apply per instance weight as the number of occurrences of each query to reduce the number of training examples to iterate over. The training time for the Amazon dataset is roughly three times longer than the AOL dataset using a batch size of 256. However, we found that if we increase the batch size to 1024, we can speed up the training by a factor of 2.2 because of increased GPU utilization, without noticeable performance loss. As a result, we can run one epoch of the Amazon dataset in approximately 6 hours, and training on the entire dataset can be done within one day.

We evaluated relatively few other architectures for this model, as the goal here is to use the character-level language model for completion rather than attain state-of-the-art results on language modeling in general. It is worth noting that the validation loss is lower than the training loss in Table 1, and the two losses becomes closer when the LSTM/GRU size is increased, indicating that our models are still in the regime of under-fitting, and even larger LSTM sizes may be used for improving performance. However, the requirement of completing the queries in real-time forbids us from using a larger model, as we will show shortly in the next section.

6.2 Runtime evaluation

Compared with the traditional database lookup based query completion system, one challenge of our deep learning based approach is its prediction time. Here we summarize the speedups achieved by the different optimizations discussed in Section 5 in Table 2, and report the time to give 16 suggestions for a prefix. A naive implementation in Keras would result in a prediction time of over one second per query, which is intolerable in the case of completing the user's query in real time, where a completion time close to typing speed is desired. With all the optimization techniques applied, we observe over 50X speedup comparing with a naive beam search implementation. These optimizations are crucial to make our deep learning based query completion practical as an online service.

One interesting point to note is that stochastic search in this setting actually takes *three times longer* with all the same optimization techniques applied than beam search, to generate the same number of completion candidates. This is due to the fact that stochastic search tends to generate completions that are much longer than those of beam search, interestingly making the "simpler" method here actually substantially slower while giving worse completions (which we will evaluate shortly).

Table 2: The speedups from different optimizations, with an LSTM of dimension 256. Results were produced on a Xeon E5-2670 machine, running 8 threads.

Optimization	Resulting runtime
Naive beam search implementation	>1sec
Tree-based beam search	250ms
Adding MKL BLAS	75ms
OpenMP parallelization	25ms
Custom LSTM implementation	13.3ms
Adding prefix edit distance	16.3 ms
Stochastic search	40 ms

As an online service, it is important to pay attention to the worst-case performance, especially because the nature of user query prefixes have a long-tail nature. In Table 3, we measure the prediction time using 100,000 real user query prefixes from Amazon.com, and report the Top-Percentiles (TPS) performance. A TP99 of 12 ms indicates that 99% of user requests can be served under 12 milliseconds. In Figure 1, we plot the cumulative distribution function (CDF) of prediction time. We observe that the distribution of prediction time given real user query prefixes does have a very long tail. We

desire that the response time of our query completion service is close to user typing speed, thus LSTM with 1024 hidden neurons is unsuitable for our use case despite showing the best prediction performance.

Table 3: Top-Percentile of completion time on the Amazon dataset with varying LSTM dimension. TPx is the minimum time (in milliseconds) under which $x\%$ of requests have been served. Results were produced on an Intel Xeon E5-2686 v4 machine, running with 8 threads.

Top-Percentile	LSTM Dimension		
	256	512	1024
TP50	5.388 ms	15.74 ms	67.19 ms
TP90	8.067 ms	24.17 ms	96.08 ms
TP99	11.14 ms	28.86 ms	114.74 ms
TP99.9	12.02 ms	30.93 ms	125.63 ms
TP99.99	12.30 ms	31.35 ms	131.84 ms

6.3 Performance evaluation

Finally, we evaluate the actual performance of the completion approaches, both comparing the performance of our beam search method to stochastic search (evaluated by log likelihood under the model), and comparing our completion method to a heavily optimized in-memory trie-base completion model, the standard data structure for completion given string prefixes.

Stochastic Search vs. Beam Search. In Table 5 we highlight the performance of beam search versus stochastic search for query completion, evaluated in terms of log likelihood under the model. Over all models, splitting methods and LSTM sizes, beam search produces substantially better results in terms of log likelihood; in addition, it is 3x faster as mentioned above. Thus we believe that beam search is necessary in our real-time query completion task, justifying our efforts on optimizing its runtime. Note that in this case we are not including any error correction, as it is not trivial to integrate this into the stochastic search setting, and we wanted a direct comparison on sample likelihood.

Our approach vs. database lookup. Finally, we compare our total approach (beam search with error correction) to a trie-based (i.e., prefix lookup) completion model. We compare the approach using a combination of two metrics: 1) probabilistic coverage, which is simply the empirical conditional probability of the predicted completion given the prefix:

$$\sum_i \hat{P}(\text{completion } i \mid \text{prefix}), \tag{16}$$

where $\hat{P}$ is the empirical probability for the whole dataset (counts of completion i over all other queries with the same prefix in the whole dataset); and 2) hit rate, which simply lists the number of times a completion appears in the entire dataset. Because the error correction model adjusts the prefix, it is not possible to compute probabilistic coverage exactly, but we can still get a sense of how likely the completions are based upon how often they occur using the hit rate metric. Table 4 shows the performance of the trie-based approach, beam search, and beam search with error correction under these metrics. Our models generally outperform trie-based approaches in all settings, the one exception being probabilistic coverage on the time-based training/testing split. This is possibly due to some amount of shift over time in the search query terms. And although we cannot generate coverage numbers for the error-correction method, the significantly larger hit rate suggests that it is indeed giving better completions than all the alternative approaches.

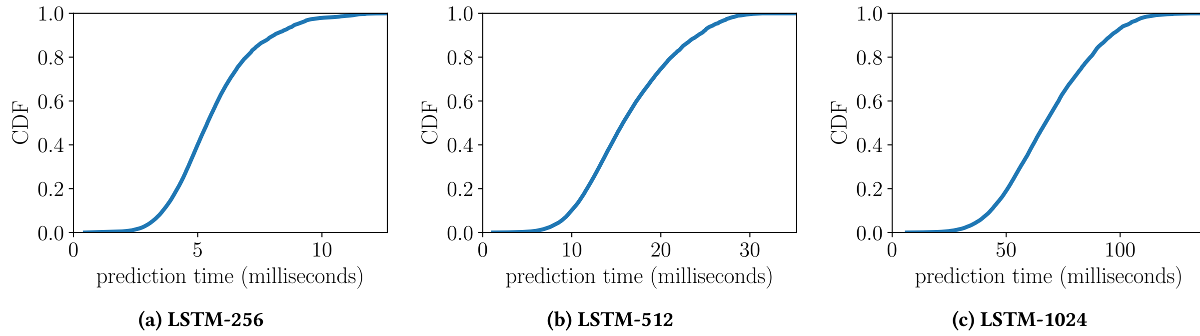
Further, we note that in addition to these numbers, there are a few notable disadvantages with trie-based lookup. The trie data structure we compare to is very memory intensive (requires keeping prefixes for all relevant queries in memory), and takes a minimum of 11 GB of RAM for the entire AOL search data set, and over 50 GB of RAM for the Amazon dataset. Our deep learning based language model approach uses over two magnitudes less memory, even at its largest configuration (LSTM-1024), as shown in Table 6. It is worth mentioning that the Amazon dataset we used in experiments contains user queries that *are sampled from* one month's data on amazon.com; if we want to utilize complete data from the month, it can be memory intensive to use the trie-based approach, while our deep learning based approach does not have this limitation and in general, a learning based approach can benefit more from having a bigger dataset.

Additionally, if a prefix has not been seen before in the dataset, the trie-based approach will offer no completions. In our test set of the Amazon dataset, which contains user queries from a time period subsequent to the training data, we observe that there are a substantial fraction of queries which do not appear in the training data. A trie-based query approach cannot give these queries as suggestions, whereas our deep learning based approach can still make suggestions using its language model. Furthermore, the trie-based approach is not amenable to error correction in isolation, as candidate corrections need to be proposed prior to lookup in the database; the process of repeatedly generating these candidates and performing the lookups will work for at most 2 edits, whereas our approach empirically easily handles completions that include 4-5 edits; this is reflected in the significantly higher hit rate in Table 4.

7 CONCLUSIONS

In this paper, we have presented a search query completion approach based upon character-level deep language models. We proposed a method for integrating the approach with an error correction framework and showed that candidate completions with error correction can be efficiently generated using beam search. We further described several optimizations that enabled the system to deliver results in real time, including a CPU-based custom LSTM implementation. We demonstrated the effectiveness of our method on two large-scale datasets from AOL and Amazon, and showed that our proposed deep learning based query completion model is able to jointly produce better completions than simple prefix lookup, while simultaneously being able to generate the candidates in real time.

Acknowledgment. The authors thank Juzer Arsiwala for his help on preparing Amazon training dataset.



(a) LSTM-256

(b) LSTM-512

(c) LSTM-1024

Figure 1: The cumulative distribution function (CDF) of completion time for different LSTM sizes. Time is measured by predicting on 100,000 real user query prefixes from Amazon.com

Table 4: Performance of our language model based methods (LSTM with 256 and 512 dimensions) versus trie-based prefix lookup on AOL dataset.

Train/test Split	Completion Method	Probabilistic Coverage	Hit Rate
Prefix Splitting	Trie-based	0.2754	1482
	Beam Search (LSTM-256)	0.4023	1679
	Beam Search (LSTM-512)	**0.4476**	1730
	Beam Search w/ error correction (LSTM-256)	-	**3864**
	Beam Search w/ error correction (LSTM-512)	-	3860
Time Splitting	Trie-based	**0.4869**	1273
	Beam Search (LSTM-256)	0.3089	1065
	Beam Search (LSTM-512)	0.3578	1080
	Beam Search w/ error correction (LSTM-256)	-	1534
	Beam Search w/ error correction (LSTM-512)	-	**1581**

Table 5: Completion negative log likelihood for stochastic search vs. beam search (lower is better)

Dataset	Split	LSTM Dimension			
		256		512	
		Beam	Stochastic	Beam	Stochastic
AOL	Prefix	0.984	1.058	**0.798**	0.840
	Time	1.569	1.726	**1.229**	1.321
Amazon	Prefix	2.708	3.159	**2.449**	2.780
	Time	**3.063**	3.646	3.259	3.863

Table 6: Number of model parameters and memory consumption of each method. Measurements were done on a 64-bit Linux machine and we record resident set size (RSS) for each program.

Dataset	Model	Number of parameters	Memory Requirement
AOL	Trie-based	0 M	11 GB
	LSTM-256	0.8 M	12 MB
	LSTM-512	3.3 M	30 MB
	LSTM-1024	12.8 M	103 MB
Amazon	Trie-based	0 M	50 GB
	LSTM-256	1.1 M	14 MB
	LSTM-512	3.8 M	35 MB
	LSTM-1024	13.9 M	112 MB

REFERENCES

[1] Ziv Bar-Yossef and Naama Kraus. 2011. Context-sensitive query auto-completion. In *Proceedings of the 20th international conference on World wide web*. ACM, 107–116.

[2] Eric Brill and Robert C Moore. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 286–293.

[3] Fei Cai, Maarten De Rijke, et al. 2016. A survey of query auto completion in information retrieval. *Foundations and Trends® in Information Retrieval* 10, 4 (2016), 273–363.

[4] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras. (2015).

[5] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. 2016. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704* (2016).

[6] Huizhong Duan and Bo-June Paul Hsu. 2011. Online spelling correction for query completion. In *Proceedings of the 20th international conference on World wide web*. ACM, 117–126.

[7] G David Forney. 1973. The Viterbi algorithm. *Proc. IEEE* 61, 3 (1973), 268–278.

[8] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[9] Bo-June Paul Hsu and Giuseppe Ottaviano. 2013. Space-efficient data structures for top-k completion. In *Proceedings of the 22nd international conference on World Wide Web*. ACM, 583–594.

[10] Dan Jurafsky and James H Martin. [n. d.]. *Speech and language processing*. Vol. 3.

[11] Mark D Kernighan, Kenneth W Church, and William A Gale. 1990. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th conference on Computational linguistics-Volume 2*. Association for Computational Linguistics, 205–210.

[12] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural Code Completion. (2016).

[13] Bruno Martins and Mário J Silva. 2004. Spelling correction for search engine queries. In *Advances in Natural Language Processing*. Springer, 372–383.

[14] Bhaskar Mitra and Nick Craswell. 2015. Query auto-completion for rare prefixes. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 1755–1758.

[15] Dae Hoon Park and Rikio Chiba. 2017. A Neural Language Model for Query Auto-Completion. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 1189–1192.

[16] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *Proceedings of the 1st international conference on Scalable information systems*. ACM, 1.

[17] Toby Segaran and Jeff Hammerbacher. 2009. *Beautiful data: the stories behind elegant data solutions*. " O'Reilly Media, Inc.".

[18] Milad Shokouhi. 2013. Learning to personalize query auto-completion. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 103–112.

[19] Alessandro Sordoni, Yoshua Bengio, Hossein Vahabi, Christina Lioma, Jakob Grue Simonsen, and Jian-Yun Nie. 2015. 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*. ACM, 553–562.

[20] Idan Szpektor, Aristides Gionis, and Yoelle Maarek. 2011. Improving recommendation for long-tail queries via templates. In *Proceedings of the 20th international conference on World wide web*. ACM, 47–56.

[21] Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)* 21, 1 (1974), 168–173.

[22] Casey Whitelaw, Ben Hutchinson, Grace Y Chung, and Gerard Ellis. 2009. Using the web for language independent spellchecking and autocorrection. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2-Volume 2*. Association for Computational Linguistics, 890–899.

[23] Ziang Xie, Anand Avati, Naveen Arivazhagan, Dan Jurafsky, and Andrew Y Ng. 2016. Neural language correction with character-based attention. *arXiv preprint arXiv:1603.09727* (2016).