An Exploration of Approaches to Integrating Neural Reranking Models in Multi-Stage Ranking Architectures

Zhucheng Tu, Matt Crane, Royal Sequiera, Junchen Zhang, and Jimmy Lin

David R. Cheriton School of Computer Science
University of Waterloo, Ontario, Canada
{michael.tu, matt.crane, rdsequie, j345zhan, jimmylin}@uwaterloo.ca

ABSTRACT
We explore different approaches to integrating a simple convolutional neural network (CNN) with the Lucene search engine in a multi-stage ranking architecture. Our models are trained using the PyTorch deep learning toolkit, which is implemented in C/C++ with a Python frontend. One obvious integration strategy is to expose the neural network directly as a service. For this, we use Apache Thrift, a software framework for building scalable cross-language services. In exploring alternative architectures, we observe that once trained, the feedforward evaluation of neural networks is quite straightforward. Therefore, we can extract the parameters of a trained CNN from PyTorch and import the model into Java, taking advantage of the Java Deeplearning4J library for feedforward evaluation. This has the advantage that the entire end-to-end system can be implemented in Java. As a third approach, we can extract the neural network from PyTorch and “compile” it into a C++ program that exposes a Thrift service. We evaluate these alternatives in terms of performance (latency and throughput) as well as ease of integration. Experiments show that feedforward evaluation of the convolutional neural network is significantly slower in Java, while the performance of the compiled C++ network does not consistently beat the PyTorch implementation.

1 INTRODUCTION
As an empirical discipline, information retrieval research requires substantial software infrastructure to index and search large collections. To address this challenge, many academic research groups have built and shared open-source search engines with the broader community—prominent examples include Lemur/Indri [9, 10] and Terrier [7, 13]. These systems, however, are relatively unknown beyond academic circles. With the exception of a small number of companies (e.g., commercial web search engines), the open-source Lucene search engine (and its derivatives such as Solr and Elasticsearch) have become the de facto platform for deploying search applications in industry.

There is a recent push in the information retrieval community to adopt Lucene as the research toolkit of choice. This would lead to a better alignment of information retrieval research with the practice of building search applications, hopefully leading to richer academic–industrial collaborations, more efficient knowledge transfer of research innovations, and greater reproducibility of research results. Recent efforts include the Lucene4IR1 workshop organized by Azzopardi et al. [3], the Lucene for Information Access and Retrieval Research (LIARR) Workshop [2] at SIGIR 2017, and the Anserini IR toolkit built on top of Lucene [24].

Given the already substantial and growing interest in applying deep learning to information retrieval [11], it would make sense to integrate existing deep learning toolkits with open-source search engines. In this paper, we explore different approaches to integrating a simple convolutional neural network (CNN) with the Lucene search engine, evaluating alternatives in terms of performance (latency and throughput) as well as ease of integration.

Fundamentally, neural network models (and even more broadly, learning-to-rank approaches) for a variety of information retrieval tasks behave as rerankers in a multi-stage ranking architecture [1, 5, 8, 11, 14, 18, 22]. Given a user query, systems typically begin with a document retrieval stage, where a standard ranking function such as BM25 is used to generate a set of candidates. These are then passed to one or more rerankers to generate the final results. In this paper, we focus on the question answering task, although the architecture is exactly the same: we consider a standard pipeline architecture [17] where the natural language question is first used as a query to retrieve a set of candidate documents, which are then segmented into sentences and rescored with a convolutional neural network (CNN). Viewed in isolation, this final stage is commonly referred to as answer selection.

We explored three different approaches to integrating a CNN for answer selection with Lucene in the context of an end-to-end question answering system. The fundamental challenge we tackle is that Lucene is implemented in Java, whereas many deep learning toolkits (including PyTorch, the one we use) are written in C/C++ with a Python frontend. We desire a seamless yet high-performance way to interface between components in different programming languages; in this respect, our work shares similar goals as Pyndri [19] and Luandri [12].

Overall, our integration efforts proceeded along the following line of reasoning:

• Since we are using the PyTorch deep learning toolkit to train our convolutional neural networks, it makes sense to simply expose the network as a service. For this, we use Apache Thrift, a software framework for building scalable cross-language services.2

• The complexities of deep learning toolkits lie mostly in training neural networks. Once trained, the feedforward evaluation of neural networks is quite straightforward and can be completely decoupled from backpropagation training. Therefore, we

1https://sites.google.com/site/lucene4ir/home
2https://thrift.apache.org/
explored an approach where we extract the parameters of the trained CNN from PyTorch and imported the model into Java, using the Deeplearning4J library for feedforward evaluation. This has the advantage of language uniformity—the entire end-to-end system can be implemented in Java.

- The ability to decouple neural network training from deployment introduces additional options for integration. We explored an approach where we directly “compile” our convolutional neural network into a C++ program that also exposes a Thrift service. Experiments show that feedforward evaluation of the convolutional neural network is significantly slower in Java, which suggests that Deeplearning4J is not yet as mature and optimized as other toolkits, at least when used directly “out-of-the-box.” Note that PyTorch presents a Python frontend, but the backend takes advantage of highly-optimized C libraries. The relative performance of our C++ implementation and the PyTorch implementation is not consistent across different processors and operating systems, and thus it remains to be seen if our “network compilation” approach can yield performance gains.

2 BACKGROUND

In this paper, we explore a very simple multi-stage architecture for question answering that consists of a document retrieval and answer selection stage. An input natural language question is used as a bag-of-words query to retrieve h documents from the collection. These documents are segmented into sentences, which are treated as candidates that feed into the answer selection module.

Given a question \( q \) and a candidate set of sentences \( \{c_1, c_2, \ldots, c_n\} \), the answer selection task is to identify sentences that contain the answer. For this task, we implemented the convolutional neural network (CNN) shown in Figure 1, which is a slightly simplified version of the model proposed by Severny and Moschitti [16].

We chose to work with this particular CNN for several reasons. It is a simple model that delivers reproducible results with multiple implementations [15]. It is quick to train (even on CPUs), which supports fast experimental iteration. Although its effectiveness in answer selection is no longer the state of the art, the model still provides a reasonable baseline.

Our model adopts a general “Siamese” structure [4] with two subnetworks processing the question and candidate answers in parallel. The input to each “arm” in the neural network is a sequence of words \( \{w_1, w_2, \ldots, w|S|\} \), each of which is translated into its corresponding distributional vector (i.e., from a word embedding), yielding a sentence matrix. Convolutional feature maps are applied to this sentence matrix, followed by ReLU activation and simple max-pooling, to arrive at a representation vector \( x_d \) for the question and \( x_q \) for the “document” (i.e., candidate answer sentence).

At the join layer (see Figure 1), all intermediate representations are concatenated into a single vector:

\[
x_{\text{join}} = [x_q^T; x_d^T; x_{\text{feat}}^T]
\]

The final component of the input vector at the join layer consists of “extra features” \( x_{\text{feat}} \) derived from four word overlap measures between the question and the candidate sentence: word overlap and \( idf \)-weighted word overlap between all words and only non-stopwords.

Our model is implemented using the PyTorch deep learning toolkit based on a reproducibility study of Severny and Moschitti’s model [16] by Rao et al. [15] using the Torch deep learning toolkit (in Lua). Our network configuration uses the best setting, as determined by Rao et al. via ablation analyses. Specifically, they found that the bilinear similarity component actually decreases effectiveness, and therefore is not included in our model.

For our experiments, the CNN was trained using the popular TrecQA dataset, first introduced by Wang et al. [23] and further elaborated by Yao et al. [25]. Ultimately, the data derive from the Question Answering Tracks from TREC 8–13 [20, 21].

3 METHODOLOGY

3.1 PyTorch Thrift

Given that our convolutional neural network for answer selection (as described in the previous section) is implemented and trained in PyTorch, the most obvious architecture for integrating disparate components is to expose the neural network model as a service. This answer selection service can then be called, for example, from a Java client that integrates directly with Lucene. For building this service, we take advantage of Apache Thrift.

Apache Thrift is a widely-deployed framework in industry for building cross-language services. From an interface definition language (IDL), Thrift can automatically generate server and client stubs for remote procedure calls (RPC). In our case, the server stub is in Python (to connect to PyTorch) while the client stub is in Java (to connect to Lucene). These stubs handle marshalling and unmarshalling parameters in a language-agnostic manner, message transport, server protocols, connection management, load balancing, etc.

Our Thrift IDL for answer selection is shown in Figure 2. The namespace specifies the module that the definition is declared in, as well as the language. The syntax for the function declaration closely
resembles a function declaration in C/C++ or Java, containing the return type of the function as well as the name and type of each parameter. The main difference is the addition of an integer parameter id, which is used for identifying fields for schema evolution.

Thrift automatically generates “service boilerplate”, and the programmer is left to implement the service and client handler. The service handler is essentially a wrapper that feeds the input received by the server stub into the neural network model already implemented in PyTorch, as shown in Figure 3. The client directly calls the corresponding method of the client stub generated by Thrift like an ordinary function invocation—in our case, from Java.

### 3.2 Deeplearning4J

In our end-to-end question answering pipeline, only the answer selection stage involves deep learning and thus depends on PyTorch. However, the training of neural networks can be easily decoupled from its deployment. Since the feedforward pass of neural networks is relatively straightforward, we can use an existing deep learning toolkit for training, and once the model is learned, we can extract the model parameters and import the model into a different toolkit (in a different programming language), or reimplement the feedforward evaluation directly in a language of our choice.

---

In our case, if we implement the feedforward evaluation in Java, then our entire question answering pipeline can be captured in a single programming language. We take advantage of a Java deep learning toolkit called Deeplearning4J, built on top of a Java numeric computation library called ND4J, to explore this idea (see example code snippet in Figure 4). Note that although Deeplearning4J is a complete deep learning toolkit with support for training neural networks, we only use it for forward evaluation. Since deep learning today remains dominated by Python-based toolkits such as PyTorch, TensorFlow, and Keras, this captures the workflow of most researchers and practitioners.

To export model parameters, we need an interoperable serialization format for exchanging data between Python and Java. For this, we use Apache Avro, a widely-adopted data serialization framework. Although Thrift can also be used for data serialization, Avro’s main advantage is its dynamic typing feature that avoids code generation. A script written in Python loads the model trained by PyTorch, extracts the parameters, parses the schema, and then writes the model parameters to an Avro file. The Avro schema we use, expressed in JSON, is shown in Figure 5.

---

\[https://deeplearning4j.org/\]

\[https://avro.apache.org/\]
Note that weights for different types of parameters could have different dimensions. For example, since multiple convolution filters are used to extract different features from the sentence matrices, and each filter is a 2-dimensional tensor (matrix), the convolution filters together form a 3-dimensional tensor. However, the edge weights for the fully-connected layer is only a 1-dimension tensor (vector). All weights are reshaped to one dimension in the serialized storage, but are restored to their original dimensions using saved dimension metadata. A Java program can read from the Avro file, parse the schema, read the dimensions, reshape each weight matrix, and finally convert them into the correct datatype for Deeplearning4J.

The advantage of the Deeplearning4J approach for reranking is language uniformity, which allows the entire question-answering pipeline to be tightly integrated into a single monolithic codebase—in contrast to the microservices architecture that is fashionable today. There are advantages and disadvantages to tightly- vs. loosely-coupled architectures, but a broader discussion is beyond the scope of our work.

### 3.3 C++ Thrift

Once we separate the training of a neural network from its deployment, we can in principle reimplement the feedforward pass in any language we choose. A Java implementation (as described in the previous section) yields language uniformity, which is a worthy goal from an integration perspective. As an alternative, we explored the potential for increased performance by “compiling” the neural network itself into a standalone binary that exposes a Thrift service.

In more detail: We have implemented a C++ conversion program that reads the same serialized model from Avro (as described in the previous section). However, rather than construct the network after reading the weights, the program instead generates another program with the network already encoded—see representative snippet of code in Figure 6. This gives the maximum potential to benefit from compiler optimizations as a number of the matrices are constants. Since the conversion program is under our control, this technique can adapt to arbitrary network architectures (although this is not the case in our current proof-of-concept implementation). Additional, our compilation approach means that any potential deployment requires only a single binary.

The generated program uses the Blaze math library\(^5\) to perform all vector/matrix manipulations, which makes extensive use of BLAS/LAPACK functions and SIMD instructions for maximum performance. Like the PyTorch implementation, the generated program exposes a Thrift service. The generation program and a simple Thrift client are publicly available on GitHub.\(^6\)

```cpp
class QuestionAnsweringHandler
{
  virtual public QuestionAnsweringIf {
    public:
      ...
    // Static matrix values
    double getScore(const std::string &question,
                    const std::string &answer) {
      // Convolution
      this->conv_result.resize(question_words.size());
      for (int k=0; k<question_words.size(); k++) {
        auto sub = submatrix(
          this->question_input, 0,
          k + COLUMN_PADDING,
          this->q_conv_filters[i].rows(),
          this->q_conv_filters[i].columns());
        auto sum = std::accumulate(cc.begin(j), cc.end(j), 0.0);
      }
      this->conv_result[k] = sum;
    } this->question_conv_map[i] =
    max(this->conv_result);
  }
  this->question_conv_map = tanh(
    this->question_conv_map +
    this->q_conv_biases);
  ...
  return score;
};
```

Figure 6: Snippet of the C++ generated program deployed behind the Thrift service, showing the convolution computation for the question.

3.4 Alternative Approaches

For completeness, it is worthwhile to discuss other obvious integration approaches that we did not examine in this study.

In the quest for language uniformity, it would also be possible to implement our entire CNN (including training) in Java using the Deeplearning4J toolkit. We decided not to pursue this route for two reasons: First, Deeplearning4J does not appear to be as mature and widely used as other deep learning toolkits such as Torch and TensorFlow. Second, we did not have an implementation of our CNN in Deeplearning4J and lacked the resources to port the model. We concede, however, that a thorough evaluation should include this experimental condition.

There are alternative approaches to integrating Python and Java code that we did not explore: PyLucene is a Python extension for accessing Java Lucene, and Py4J is a bridge that allows a Python program to dynamically access Java objects in a Java Virtual Machine. In such an architecture, the entire question-answering pipeline would be implemented in Python, calling Java for functionalities that involve Lucene. Again, we did not have sufficient resources to explore this approach to integration, but agree that such a condition should be evaluated in future work.

### 4 RESULTS

We compared the performance of our three proposed approaches in terms of latency and throughput. All the experiments in this paper were performed on two machines:

- A desktop with an Intel Core i7-6800K CPU (6 physical cores, 12 logical cores with hyperthreading) running Ubuntu 16.04.
- An Apple MacBook Air laptop with an Intel Core i5-5250U CPU (2 physical cores, 4 logical cores with hyperthreading) running macOS Sierra 10.12.6.

---

\(^5\)https://bitbucket.org/blaze-lib/blaze
\(^6\)https://github.com/snapbug/coconut
For the Python implementation, we used Python 3.6 and PyTorch 0.1.12. For the Java implementation, we used Java 8 and Deeplearning4j 0.8.0 on Oracle JVM 1.8.0. Deeplearning4j (which uses the ND4J matrix library) uses OpenBLAS by default. Our C++ implementation uses Blaze 3.1 and was compiled with `-O3 -DNDEBUG` options using gcc 5.4 on the desktop and Apple LLVM version 8.1.0 (clang-802.0.42) on the laptop.

Both PyTorch and Deeplearning4j use the OMP_NUM_THREADS environment variable to decide how many threads to use. We set OMP_NUM_THREADS=1 for all experiments. The C++ implementation is also single-threaded. In the relevant configurations, Avro 1.8.2 and Thrift 0.10.0 are used. All implementations ran on the CPU only. Performance measurements do not include deserialization time of the model parameters and other startup costs.

### 4.1 Feedforward Evaluation

We first examined the performance of the three approaches without Thrift. For each implementation, after we loaded the model parameters into memory, we iterated through the question–answer pairs from the TrecQA raw-dev and raw-test splits and invoked the appropriate function that computes the similarity score for each pair. We divided the total number of pairs scored by the total elapsed time to arrive at performance in terms of queries per second (QPS). Results are summarized in Table 1. The calling program ran on a single thread.

We observe that the performance of the Java Deeplearning4j implementation is approximately 2–6x slower than the PyTorch and C++ implementations, depending on the environment. In our initial Java implementation, we used the ND4J matrix library directly to implement convolutions in the simplest way possible—looping over filters and convolving each filter with the sentence embedding separately. The performance of this naïve implementation was two orders of magnitude worse than that of Deeplearning4j, because internally Deeplearning4j implements convolutions using the im2col approach, which turns the problem into a matrix multiplication and takes advantage of highly-optimized functions such as GEMM.\(^7\) We spent a considerable amount of time investigating the performance of our Java implementations (both using ND4j directly and using Deeplearning4j). We were unable to further improve the performance of the Deeplearning4j implementation; it seems that the toolkit already exploits all the “obvious” optimization tricks.

Interestingly, the relative performance of the C++ and PyTorch implementations is not consistent across our different test environments, which is likely attributable to some combination of hardware, operating system, and the software toolchain (e.g., gcc vs. LLVM). We also note that the performance gap between the Java implementation and the other two vary across the different environments. Although performance differences are to be expected, we currently lack a clear understanding of their sources.

Overall, these results suggest that Deeplearning4j and ND4j are less mature than their counterparts in Python/C++ (i.e., PyTorch and numpy) and “out-of-the-box” configurations may need further fine-tuning to compete with the other approaches. The lack of a consistent performance advantage in our C++ implementation suggests that the Python frontend to PyTorch adds minimal performance overhead (since the PyTorch backend also uses optimized C libraries). It remains to be seen if further optimizations in our C++ implementation can translate into consistently better performance.

### 4.2 Thrift Service

Finally, we compared the performance of the PyTorch and C++ implementations behind Thrift. We did not examine wrapping Deeplearning4j in a Thrift service since the advantage of the Java implementation is the ability to directly integrate with Lucene. We expect Thrift to introduce some overhead due to data serialization/deserialization and network protocols, but such a design enables components in an end-to-end question answering pipeline to be built in different languages.

Typically, in a microservices architecture, the Thrift server and clients would run on separate machines, but for the purposes of this experiment, we run them both on the same machine. In this setup, we use a Python Thrift client to make requests to both the Python Thrift server as well as the C++ Thrift server. The Python client uses a single thread to send requests—keeping the configuration as close to the non-Thrift setting as possible. Both Thrift servers use TSimpleServer, which runs a single thread, accepts one connection at a time, and repeatedly processes requests from the connection. Results are summarized in Table 2, with p50 and p99 denoting median and 99th percentile latency, respectively.

Comparing the results in Table 1 with those in Table 2, we are able to quantify the overhead introduced by the Thrift service. On both machines, wrapping PyTorch with Thrift added about 6–7% overhead. Wrapping C++ with Thrift added around 10% and 24% overhead on the laptop and on the desktop, respectively. In this particular case, it seems that the Python Thrift implementation is more efficient than the C++ one.

---

\(^7\)https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

---

### Table 1: Feedforward evaluation performance of the CNN (without the Thrift service wrapper).

| Machine | Approach | Throughput (QPS) | Latency (ms) |
|---------|----------|------------------|--------------|
| Laptop | PyTorch | 828.57 | 2.22 |
| Desktop | PyTorch | 1226.49 | 1.01 |
| Laptop | Deeplearning4j | 165.75 | 2.01 |
| Desktop | Deeplearning4j | 530.4 | 1.00 |
| Laptop | C++ Generator | 1025.91 | 2.10 |
| Desktop | C++ Generator | 1235.50 | 1.00 |

### Table 2: End-to-end performance of the Thrift service.

| Machine | Approach | Throughput (QPS) | p50 Latency (ms) | p99 Latency (ms) |
|---------|----------|------------------|------------------|------------------|
| Laptop | PyTorch Thrift | 774.28 | 1.20 | 2.51 |
| Desktop | PyTorch Thrift | 1150.86 | 0.83 | 1.72 |
| Laptop | C++ Thrift | 924.39 | 1.08 | 1.71 |
| Desktop | C++ Thrift | 1000.40 | 1.00 | 1.59 |
5 CONCLUSIONS

In light of the recent push in the information retrieval community to adopt Lucene as the research toolkit of choice and the growing interest in applying deep learning to information retrieval problems, this paper explores three ways to integrate a simple CNN with Lucene. We can directly expose the PyTorch model with a Thrift service, extract the model parameters and import the model into the Java Deeplearning4j toolkit for direct Lucene integration, or “compile” the network into a standalone C++ program that exposes a Thrift service.

All considered, the simplest approach of wrapping PyTorch in Thrift seems at present to be the best option that combines performance and ease of integration. The other two approaches have potential advantages that remain currently unrealized. For Java, the performance gap might shrink over time as Deeplearning4j becomes more mature. For our compiled C++ approach, additional optimizations might yield consistent performance gains to justify the additional complexity. Finally, there are other approaches discussed in Section 3.4 that we have not yet explored. The best approach to integrating neural networks with the Lucene search engine for information retrieval applications remains an open question, but hopefully our preliminary explorations start to shed some light on the relevant issues.

6 ACKNOWLEDGMENTS

We’d like to thank the DeepLearning4j team for help with their toolkit. This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada, with additional contributions from the U.S. National Science Foundation under CNS-1405688. Any findings, conclusions, or recommendations expressed do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Nima Asadi and Jimmy Lin. 2013. Effectiveness/Efficiency Tradeoffs for Candidate Generation in Multi-Stage Retrieval Architectures. In SIGIR 997–1000.
[2] Leif Azizpardi, Matt Craner, Hui Fang, Grant Ingersoll, Jimmy Lin, Yashar Moshfeghi, Harrison Scoll, Peilin Yang, and Guido Zuccon. 2017. The Lucene for Information Access and Retrieval Research (LIARR) Workshop at SIGIR 2017. In SIGIR.
[3] Leif Azizpardi, Yashar Moshfeghi, Martin Halvey, Rami S. Alkhawaldeh, Kristian Balog, Emanuele Di Bucchio, Diego Cecarelli, Juan M. Fernandez-Luna, Charlie Hull, Jake Mannix, and Sauparna Pal Chowdhury. 2016. Lucene4IR: Developing Information Retrieval Evaluation Resources using Lucene. SIGIR Forum 50, 2 (2016), 58–75.
[4] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Sackinger, and Roopak Shah. 1993. Signature Verification Using a “Siamese” Time Delay Neural Network. In NIPS. 737–744.
[5] J. Shane Culpepper, Charles L. A. Clarke, and Jimmy Lin. 2016. Dynamic Cutoff Prediction in Multi-Stage Retrieval Systems. In ADCCS 17–24.
[6] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. High Performance Smart Expression Template Math Libraries. In HPCS. 367–373.
[7] Craig Macdonald, Richard McCreadie, Rodnyys L. T. Santos, and Iadh Ounis. 2012. From Puppy to Maturity: Experiences in Developing Terrier. In SIGIR 2012 Workshop on Open Source IR.
[8] Irina Matveeva, Chris Burges, Timo Burkard, Andy Laucius, and Leon Wong. 2006. High Accuracy Retrieval with Multiple Nested Ranker. In SIGIR. 437–444.
[9] Donald Metzler and W. Bruce Croft. 2004. Combining the Language Model and Inference Network Approaches to Retrieval. IPAM 40, 5 (2004), 735–750.
[10] Donald Metzler, Trevor Strohm, Howard Turtle, and W. Bruce Croft. 2004. Indri at TREC 2004. Terabyte Track. In TREC.
[11] Bhaskar Mitra and Nick Craswell. 2017. Neural Models for Information Retrieval. arXiv:1701.09749.
[12] Bhaskar Mitra, Fernando Diaz, and Nick Craswell. 2017. Luandri: A Clean Lua Interface to the Indri Search Engine. arXiv:1702.05042.
[13] Iadh Ounis, Giannis Amati, Vassilis Plachouras, Ben He, Craig Macdonald, and Christina Lioma. 2006. Terrier: A High Performance and Scalable Information Retrieval Platform. In SIGIR 2006 Workshop on Open Source IR.
[14] Jan Pederesen. 2010. Query Understanding at Bing. In Invited Talk at SIGIR.
[15] Jinfeng Rao, Hua He, and Jimmy Lin. 2017. Experiments with Convolutional Neural Network Models for Answer Selection. In SIGIR.
[16] Aliaksei Severyn and Alessandro Moschitti. 2015. Learning to Rank Short Text Pairs with Convolutional Deep Neural Networks. In SIGIR. 373–382.
[17] Stefanie Tellex, Boris Katz, Jimmy Lin, Gregory Marton, and Aaron Fernandez. 2003. Quantitative Evaluation of Passage Retrieval Algorithms for Question Answering. In SIGIR. 41–47.
[18] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. 2013. Efficient and Effective Retrieval Using Selective Pruning. In WSDM. 63–72.
[19] Christophe Van Gysel, Evangelos Kanoulas, and Maarten de Rijke. 2017. Pyndri: a Python Interface to the Indri Search Engine. arXiv:1701.09749.
[20] Ellen M. Voorhees and Hoa Trang Dang. 2005. Overview of the TREC 2005 Question Answering Track. In TREC.
[21] Ellen M. Voorhees and Dawn M. Tice. 1999. The TREC-8 Question Answering Track Evaluation. In TREC.
[22] Lidan Wang, Jimmy Lin, and Donald Metzler. 2011. A Cascade Ranking Model for Efficient Ranked Retrieval. In SIGIR. 105–114.
[23] Mengqiu Wang, Noah A. Smith, and Teruko Mitamura. 2007. What is the Jeopardy Model? A Quasi-Synchronous Grammar for QA. In EMNLP-CoNLL. 22–32.
[24] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In SIGIR.
[25] Xuchen Yao, Benjamin Van Durme, Chris Callison-Burch, and Peter Clark. 2013. Answer Extraction as Sequence Tagging with Tree Edit Distance. InHLT-NAACL. 858–867.