# DeQA: On-Device Question Answering

Qingqing Cao
qicao@cs.stonybrook.edu
Stony Brook University

Noah Weber
nwweber@cs.stonybrook.edu
Stony Brook University

Niranjan Balasubramanian
niranjan@cs.stonybrook.edu
Stony Brook University

Aruna Balasubramanian
arunab@cs.stonybrook.edu
Stony Brook University

## ABSTRACT

Today there is no effective support for device-wide question answering on mobile devices. State-of-the-art QA models are deep learning behemoths designed for the cloud which run extremely slow and require more memory than available on phones. We present DeQA, a suite of latency- and memory- optimizations that adapts existing QA systems to run completely locally on mobile phones. Specifically, we design two *latency optimizations* that (1) stops processing documents if further processing cannot improve answer quality, and (2) identifies computation that does not depend on the question and moves it offline. These optimizations do not depend on the QA model internals and can be applied to several existing QA models. DeQA also implements a set of *memory optimizations* by (i) loading partial indexes in memory, (ii) working with smaller units of data, and (iii) replacing in-memory lookups with a key-value database. We use DeQA to port three state-of-the-art QA systems to the mobile device and evaluate over three datasets. The first is a large scale SQuAD dataset defined over Wikipedia collection. We also create two on-device QA datasets, one over a publicly available email data collection and the other using a cross-app data collection we obtain from two users. Our evaluations show that DeQA can run QA models with only a few hundred MBs of memory and provides at least 13x speedup on average on the mobile phone across all three datasets.

## CCS CONCEPTS

• **Information systems → Question answering**; • **Human-centered computing → Mobile computing**; **Mobile devices**.

## KEYWORDS

Question Answering; Mobile Devices; Mobile Systems

**ACM Reference Format:**
Qingqing Cao, Noah Weber, Niranjan Balasubramanian, and Aruna Balasubramanian. 2019. DeQA: On-Device Question Answering. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307334.3326071


*MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326071

## 1 INTRODUCTION

Mobile users today access content via multiple applications – browsers, email, social media, and many more. A device-wide Question Answering (QA) over all data accessed on a device can help users find information efficiently.

Consider Mateo who is out looking for an off-the-counter medication for his diabetic son. He has browsed vendor sites, discussed options with his son's physician over email, and has gathered suggestions from his social media friends. To decide on an alternative at the store, Mateo needs to know: (a) the price of the product he found earlier on the Web, (b) alternatives his friends suggest? and (c) alternatives his physician suggested? The answers to these questions are in his email conversations, Web searches, and social media feeds. A device-wide QA system can help him find these answers.

Unfortunately, such a device-wide QA system is difficult to design today. To do so, the QA system needs to work across device-wide content rather than within a single application ecosystem, such as email. This means the entire device content needs to be shipped to the cloud. This is untenable for privacy, data protection, and performance reasons (more discussions in §2).

We design DeQA (pronounced de-ka), which stands for *on-device QA*, that runs completely locally on a mobile device. DeQA stores data accessed by the user locally and then runs QA over this data. Storing and indexing a user's digital footprint is feasible on mobile devices, given the large storage capacity on phones [16]. The central challenge, however, is in running end-to-end QA systems on mobile devices. State-of-the-art systems use deep learning based QA models [35, 54, 57], which are unusably slow on mobile devices.

DeQA is a set of compute- and memory-optimizations to significantly improve QA on mobile. We design DeQA to be broadly applicable to existing QA systems by studying the common patterns and bottlenecks of state-of-the-art QA models. To this end, we first study three top-ranked QA models—RNet [54], MReader [35], and QANet [57]—from the SQuAD 1.1 Leaderboard [50], a widely used benchmark in the NLP community.

Even on a relatively new mobile phone (running Android 8 with Snapdragon 820 CPU, 6GB RAM and 64GB storage) the three QA systems take over 80 seconds to answer a single question, making them unusable. The problem is, existing mobile deep learning optimizations [32, 33, 37, 59] for vision/CNN applications do not apply to QA, as we discuss in §4. Instead, we design three optimizations based on our study:

**(i) Dynamic Early Stopping** Our first observation is that in many cases the QA system does not need to process all documents to answer a question. For easy questions the answer is likely found

immediately, in the top few documents, and for hard ones processing more documents is unlikely to yield better answers. DeQA formalizes this idea by designing a *Dynamic Early Stopping* algorithm that predicts when further processing is going to be unhelpful. The stopping algorithm uses a separate classifier for the prediction, and the features used to train the classifier do not rely on the internal states of the model or require model modifications.

**(ii) Offloading Neural Encoding** Our second observation is that a large part of the computation performed by the QA model does not depend on the question. If any processing does not depend on the question, then it does not have to be computed at run time. In fact, on all three models—RNet, MReader, and QANet—neural encoding of the documents (§3) take over 50% of the time, but this step is independent of the question. In DeQA, we move this neural encoding out of the critical path and process them offline.

**(iii) Memory Optimizations** Even with the latency optimizations, existing QA systems cannot run *as-is* on mobile devices because of their memory requirements. QA systems trade memory usage for latency benefits by processing documents as a whole to locate answers, using an in-memory index for easy document retrieval, and performing in-memory lookups. DeQA reduces memory requirements by (a) working with partial indexes loaded iteratively in memory, (b) breaking down the job into smaller units of paragraphs instead of documents, and (c) replacing in-memory lookups with a key-value database.

We implement DeQA on two mobile platforms: Nvidia TX2 board [8] and OnePlus 3 Android phone [9]. We adapt RNet [54], MReader [35], and QANet [57] to mobile devices using DeQA requiring minimal changes to the QA models themselves.

We first evaluate the performance of DeQA on the Stanford Question Answering Dataset (SQuAD) [50] and CuratedTrec [17] dataset with over 10K question/answer pairs run over Wikipedia collection stored locally. Using DeQA optimizations, the end-to-end QA system only requires a few hundred megabytes of memory. DeQA also reduces QA latencies by an average of 16x over the phone and 6x over the board, with less than 1% drop in accuracy for all 3 models.

There are no standard QA datasets for question answering over on-device user data. Instead, we create two smaller datasets: (a) a 120 question-answer dataset created over publicly available email collection from Enron [38], and (b) a 100 question-answer dataset over data collected from two users across 5 different apps. The cross-app data is created by recording content as users interact with their apps over a 1 week period. For the email dataset, DeQA improves latency by an average of 14x on the phone and 6x on the board with a little more than 1% drop in accuracy for all three models. For the cross-device dataset, DeQA improves latency by an average of 13x on the phone with 1.5% accuracy drop.

## 2 MOTIVATION

Device-wide question answering is an important capability for intelligent assistants. They have the potential to provide a single-entry point for finding answers over content that is already available on the user's device. Here we motivate the need for such a device-wide QA service that resides entirely on-device.

Consider Alice, a New Yorker, who is on a business trip to Seoul. She booked flights and hotel rooms, and she looked up information about the local transportation, landmarks and tourist attractions of Seoul before arrival. In Seoul, if she has questions about which hotel she is staying in, her return flight schedule or the location of a particular tourist location, she has two choices. Either re-search the information or use a QA service that is provided by each individual application, if available. Both of these options are burdensome because Alice needs to remember where she got the information from, or search in each app to see if the information is available. A device-wide QA service, on the other hand, can provide a single entry point for meeting all of Alice's questions during her travel.

One possible solution for such a device-wide QA service is to allow a single cloud-based provider to integrate all of a user's content and provide a single QA service over the entire content. However, this forces an unreasonable privacy tradeoff on the user where all of the user data is now available to a single third-party service. Indeed, by integrating all content, the third-party can now learn even more information about the user than any single application provider. Further, relying on a cloud service means Alice needs to use a possibly expensive and unreliable international roaming plan. Recently, secure enclaves in the cloud, such as SGX [5], have becoming popular, where computation is performed within the enclave without revealing sensitive user data. However, SGX enclaves are extremely restricted in terms of compute and memory capacity [36].

An on-device QA service, however, integrates content from different apps but keeps the integrated content entirely local, which means users do not have to trust a single entity to store all of their data. Further, being on-device allows the QA service to operate even under unstable or expensive network conditions.

## 3 ARCHITECTURE OF QA MODELS

We envision a device-level QA solution that returns answers from text locally available on the user device. This is similar to desktop search [28] that supports search over local content, but our goal is to go beyond search and support question answering which is a more complicated task.

Previous studies have shown that a user's local digital footprint can be stored and indexed locally on a mobile device [16]. The key challenge is in running QA over this data, which is the focus of the paper. A complementary problem is in aggregating data accessed by users across different apps for on-device QA. We leave this problem for future work.

### 3.1 Question Answering Background

QA systems search through documents to locate candidate answers. To avoid having to analyze a large number of documents, end-to-end QA systems work in two stages. In the first search engine stage, the system obtains a subset of documents relevant to the question, and in the second stage the system runs complex QA models on this subset. Figure 1 shows a typical end-to-end QA pipeline which consists of the search engine and the QA model.

(i) **Search engine** A search engine uses an index, an efficient lookup data structure, to find documents that match the question keywords. It then returns a ranked set of documents according

| Rank | Name | Model Type | Cite |
|---|---|---|---|
| 1 | QANet | Transformer | [57] |
| 2 | Reinforced Mnemonic Reader (MReader) | BiLSTM | [35] |
| 3 | RNet | BiGRU | [54] |
| 4 | SLQA+ | BiLSTM | [53] |
| 5 | Hybrid AoA Reader | BiGRU | [26] |
| 6 | BiDAF + SelfAttn + ELMo + A2D | BiGRU | [25] |
| 7 | MAMCN+ | BiGRU | [58] |
| 8 | MEMEN | BiLSTM | [48] |
| 9 | RaSoR + TR + LM | BiLSTM | [52] |
| 10 | SAN | BiLSTM | [43] |

**Table 1: The top 10 QA models on the SQuAD dataset v1.1 as of September 18, 2018. We only rank models for which technical details are available, and variations of the same model are merged.**

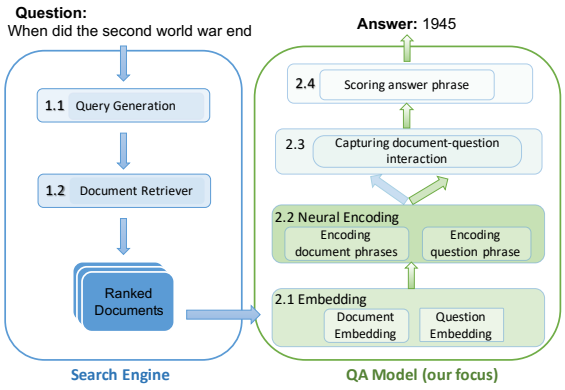



**Figure 1: Question Answering Pipeline: There are two main components. The search engine returns a sorted list of relevant documents in response to the question. The QA model uses a neural representation of the documents and the question to score the potential answer phrases in the relevant documents.**

to the relevance of the document to the question, using a scoring function. As users access new content, the search engine needs to store the data and index it.

(ii) **QA Model** A QA model analyzes the top-n documents returned by the search engine to score the phrases in the document as possible answers. The core task that QA models perform is to verify if the information contained in the potential answer phrase and its surrounding context supports a claim (i.e., is the answer to the question). This is a hard problem because the information can be expressed through different words, split across multiple sentences, and often might require significant amount of inference.

## 3.2 State-of-the-art QA models

Most state-of-the-art QA models today use deep learning to perform the complex QA task. Table 1 lists top QA models (as of September 18, 2018) from the SQuADv1.1 leaderboard, a large scale benchmark dataset that is widely-used in the NLP field.

Most models use BiLSTM or BiGRU as encoding layers with Recurrent Neural Network (RNN) [29] blocks, which compose representations of inputs in a sequential manner. One model, the QANet [57], uses self-attention encoding layers with transformer blocks, which mimic RNN capabilities for modeling context with fewer sequential dependencies. Transformer-based models have gained popularity for QA over the last year (for example, BERT [27]) in part because they allow for deeper and larger models that can be trained faster [27, 49], compared to the RNN models.

## 3.3 Structure of state-of-the-art QA models

The state-of-the-art QA models share a common underlying architecture and differ only in how they implement the architecture [55]. The right block in Figure 1 (QA Model) shows the processing steps in the different layers of a QA model. We describe each step below:

**Embeddings (2.1 in Figure 1)** The first step in a QA model is to map the words in the documents (and the question) to their corresponding word embeddings. Embeddings are used to represent each word as a $k$-dimensional real-valued vector. These vectors are said to capture word meaning in the sense that words with similar usages and meaning have similar vectors i.e., the vector distance between similar words is shorter than those between dissimilar words.

**Neural Encoding (2.2 in Figure 1)** The neural encoding layer combines the word embeddings to produce a representation of longer pieces of text. For instance, the meaning of a sentence can be represented as the meaning of each word in the context of the entire sentence. The encoding layers in QA models use RNNs such as BiLSTMs [54] or transformers [57].

**Question document interaction: (2.3 in Figure 1)** The encoding layers usually produce independent representations of the question and passage. The interaction layer creates a representation of the passage that is related to the question at hand. A passage might contain many different pieces of information, and the interaction layer allows the model to emphasize the relevant portions of the passage.

**Scoring (2.4 in Figure 1)** The scoring layer uses the neural representation and the interaction information to locate answers by looking for the most likely starting and ending positions for the answer. The models evaluates each start and end position and provides a score. The answer phrase with the highest score is returned as the final answer.

# 4 BOTTLENECKS IN RUNNING QA MODELS ON MOBILE DEVICES

We study the bottlenecks of three end-to-end QA systems instantiated with the top-ranked QA models from Table 1–QANet [57], MReader [35], and RNet [54]. The QA systems all use a search engine to retrieve relevant documents and then run one of the three models over the documents to answer the question.

One challenge is that existing QA systems simply will not run on mobile devices due to their memory requirement. DeQA's memory optimization reduces the memory requirements of the QA systems

to fit on mobile. We conduct the latency benchmarks over this memory-optimized version. We discuss the memory optimizations in §6 and focus on the latency bottlenecks in this section.

## 4.1 Measurement setup

**QA Dataset** We study the performance of QA systems using SQuADv1.1 [50], a large question-answer dataset that is widely used within the NLP community to evaluate QA models. The dataset is built using Wikipedia text as its knowledge source containing 5.5 million articles amounting to 12 GB of data stored on disk. The recently introduced SQuADv2.0 is a harder dataset that includes questions that may not have an answer in the given document context. But in the full Wikipedia setting, those no-answer questions can have correct answers potentially, therefore, for our purposes we find SQuADv1.1 is more suitable.

We use SQuAD because there are no standard datasets for QA over personalized on-device user data. In our evaluation (§9) we create two new datasets over on-device user data and show that the bottlenecks are similar on these datasets.

**Methodology** The models are trained on a collection of 240k question answer pairs from SQuAD [50], CuratedTrec [17], WikiMovies [45], and WebQuestions [18]. For training purposes the models not only need the answers to questions, but also need to know where the correct answers are found in text. To this end, given the questions and their answers, we use the distant supervision procedure described in DrQA [23] to automatically locate the answers within the text and generate labeled answer spans. We tune the model's hyperparameters on the dev set in the SQuAD dataset.

For testing we use the dev split (10k questions) from SQuAD and the test split (694 questions) of the CuratedTrec dataset.

**Device specification:** We benchmark the QA systems on four devices: (i) **Cloud:** A machine from the Google Cloud Platform which has 8 vCPUs, 32 GB memory, 2 x NVIDIA Tesla T4 GPUs, 100GB SSD disk, (ii) **PC:** an Intel PC running Ubuntu with a 3.4 GHz CPU, GTX 1070 GPU and 32GB memory, (iii) **Mobile Board:** A next generation NVidia Jetson TX2 [8] development board with quad core ARM 64bit CPU, a 256-core CUDA GPU, 8GB memory, and 128GB storage. The TX2 board is a high performance platform for next generation applications such as autonomous driving [44] and VR headsets [4, 6], (iv) **Mobile Phone:** A OnePlus 3 Android Phone [9] running Android 8.0. The phone runs a Snapdragon 820 CPU, with 6GB RAM and 64GB storage.

**Metrics** The accuracy of the QA system is measured as the percentage of questions for which the answer is within the top 5 phrases returned by the QA model. All experiments on the TX2 board use CUDA GPU [3] by default.

## 4.2 Latency of existing QA systems on mobile

Table 2 shows the latencies for running the three top-ranked QA systems on the cloud platform, the desktop PC, and the two mobile platforms. First, on the cloud platform, answering a question only takes a few hundred milliseconds while the desktop PC takes a few seconds to answer a question. This is not surprising given that the cloud GPU is more powerful than that of the desktop PC. With two GPUs, the cloud platform can process all the documents in parallel.

On the mobile platforms, it takes much longer to answer a question even compared to the desktop PC. For instance, on the mobile phone, answering a single question takes over a minute, making it effectively unusable.

| | MReader | RNet | QANet |
|---|---|---|---|
| Cloud | 314 ms | 235 ms | 209 ms |
| Desktop PC | 3.63 s | 5.11 s | 3.13 s |
| Mobile platforms | | | |
| TX2 Board | 24.13 s | 28.48 s | 23.48 s |
| Mobile Phone | 81.68 s | 88.55 s | 80.66 s |

**Table 2: Latencies of running QA systems on the cloud, desktop PC, and the two mobile platforms. Answering a question only takes a few hundred milliseconds on the cloud and a few seconds on the PC. QA takes much longer on the mobile platforms, requiring more than a minute on the phone.**

## 4.3 Bottlenecks on the mobile device

A QA system has several components as shown in Figure 1. We breakdown the QA latency to study where the main bottleneck is for two models MReader and RNet. The bottlenecks were similar for QANet. This study was done on the TX2 board.

Table 3 shows the breakdown in terms of percentage time for the different components. The key bottleneck is in neural representation of the paragraphs. About half of the processing time is spent in encoding paragraphs i.e., in processing the retrieved paragraphs through the RNN layers in the case of MReader and RNet, and the transformer blocks in the case of QANet. None of the other individual steps are significant bottlenecks by themselves.

The question we answer in DeQA is, how can we address the bottleneck in the deep learning layers to significantly reduce QA latency on mobile devices?

## 4.4 Using existing deep learning optimizations for QA

A natural idea to improve QA performance, given that the deep learning component is the bottleneck, is to use existing deep learning optimizations designed for mobile phones. Most existing optimizations have focused on running general deep neural networks (DNNs) and Convolutional Neural Networks (CNNs). While these work well for vision and sensing applications [19, 41, 42], they are ill-suited for optimizing the QA models.

Existing QA models process a large amount of data for every question (138 documents, 120k words). Even though the time taken to process each document through the RNN model is small, the latency increases because of the number of documents that need to be processed. In contrast, vision models themselves are large but the models do not process large amounts of data. As a result, several CNN optimizations focus on model compression and pruning [32, 33, 59]. In contrast, the QA model sizes are already small.

Second, the encoding layers in these models often have more dependencies and therefore fewer opportunities for parallelism

| QA Stage | MReader | RNet |
|---|---|---|
| 1. Search Engine | 15% | 10.4% |
| 2. Embeddings and preprocessing | 8.53% | 9.01% |
| 2.2 Encoding Question | 8.47% | 8.96% |
| **2.2 Encoding neural Rep of Paragraphs** | **48.54%** | **52.26%** |
| 2.3 Capturing interactions | 16.38% | 13.17% |
| 2.4 Classifying answer spans | 2.24% | 2.76% |
| Total time | 24.127 s | 28.477 s |

**Table 3: The latency breakdown for each component of the QA pipeline for MReader [35] and RNet [54]. The results show the percentage time spent in each component when answering questions from the CuratedTrec-test dataset (694 questions) on the TX2 board. Encoding the Neural Representation of Paragraphs is the main bottleneck, accounting for nearly 50% of the total time.**

compared to the CNN models, where filters are extremely parallelizable. While QA models can be parallelized at the document level, one cannot easily parallelize the processing within a document.

## 5 DeQA LATENCY OPTIMIZATIONS

In DeQA we design latency optimizations that significantly reduce the time taken to run end-to-end QA on a constrained mobile device. We do not want the optimizations to rely on the internals of the QA models because even if they provide benefits for one model, they may not be effective for others. Instead, the DeQA optimizations are designed by leveraging common patterns of existing QA models (Figure 1) and a bottleneck analysis (Table 3).

The QA model (including all the optimizations) are trained offline, once for a given model. The trained model runs on the user device to answer questions. Before running end-to-end QA for the first time, we index all documents accessed by the user across all apps and store them locally. The question answering system is run over this index. Of course, as new documents arrive, we need to update the index incrementally. We show in §8.2 that this incremental indexing incurs negligible energy and latency.

### 5.1 Dynamic Early Stopping

Our preliminary study shows that the QA model pipeline is the key latency bottleneck. The one reason for this bottleneck is the amount of data that is processed by the encoding layers. A document contains a large number of words, and encoding these words increases QA latency significantly. Instead, we propose an early stopping algorithm where the QA models do not process all the documents. The algorithm indicates when processing further documents will not improve answer accuracy.

*5.1.1 Early Stopping: Answer Distribution.* The effectiveness of a stopping strategy depends on how the correct answers are distributed. Recall that the search engine returns a ranked list of documents according to the relevance of the document to the question, and the QA model runs on this ranked list of document. Naturally, a common premise in QA systems is that the answers are more likely to be found in the top ranked documents.

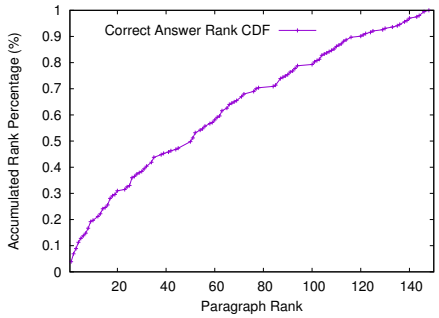



**Figure 2: CDF of the ranks of the paragraphs that contain the correct answer. The correct answers are evenly distributed across all ranks. Therefore, using a fixed stopping algorithm will result is not finding the answer in many cases.**

Given this, a reasonable approach would be to process only a small *fixed* number of top documents (or paragraphs[1]) for every question. However it turns out that the answers are not always found in the top ranks. Figure 2 shows that the correct answers are evenly distributed across all ranks. Specifically, if you process the top 150 paragraphs returned by the search engine, 50% of the time the paragraph that contains the correct answer is within the top 60 documents according to the search engine ranking.

Suppose we had a perfect QA model. It will correctly answer a quarter of the questions when processing top 20 paragraphs for each question, answer about half with top 60 paragraphs, and answer three quarters with top 100 paragraphs. This means that while some questions can be answered with a small number of paragraphs, others require more. Therefore, no *fixed* choice is optimal for both accuracy and latency (as also corroborated in our evaluation in §8).

*5.1.2 Dynamic Early Stopping Classifier.* Our approach is based on the observation that stopping early makes sense for questions that are too *easy* or too *hard*. For easy questions the answer is presumably found in the early documents, and the QA model is able to provide a significantly high score to the correct answer compared to the other candidates. For hard questions the QA model will find no candidate answer to be satisfactory and will provide low scores to all candidate answers seen so far. In either case it is better to stop as further processing is unlikely to change the QA model's final answer.

Figure 3 illustrates the main idea of our *early stopping* algorithm[2]. As noted before, the QA model processes the paragraphs in the order ranked by the search engine. After each paragraph (or batch of paragraphs), the classifier is asked to predict whether to continue processing more paragraphs or stop and return its current best answer. We model this as a supervised classification task using the following features:

*1. Retrieval scores* – The retrieval score, which is used to rank the paragraphs by the search engine, is an indicator of the relevance of the paragraph to the question.

[1]We use paragraphs for the rest of the early stopping description because DeQA uses paragraph retrieval instead of document retrieval, as we discuss in §6

[2]This is unrelated to the machine learning term used to denote the early stopping of training using performance on a development set.

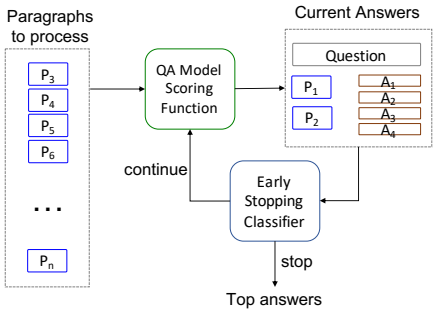



**Figure 3: Early stopping model: At each time step the current outputs from the DeQA model are evaluated by the early stopping classifier to decide to stop or continue processing.**

*2. QA Model Scores* – The QA scores indicate the confidence of the QA model with respect to the answers it has seen thus far.
*3. Z-Scores* – When the correct answer is found we'd expect to see a large increase in the answer score compared to the previous scores. To model this, we measure how much the answer score deviates from the previous scores by using the *log-transformed Z-score [12]* of the current answer score. The Z-score [12] measures the distance between the data point and the mean in terms of the number of standard deviations. The intuition is that for hard questions or cases when the model is confused, one would expect similar (possibly low) scores to all answer candidates. In these cases, the largest deviation is likely to be small. To capture this we also track the *largest Z-score* thus far.
*4. Duplicate Answers* – The number of *repeated answers* from different paragraphs is a good indicator that the answer is likely to be correct.
*5. Paragraphs Processed* – The more the number of documents processed, the less likely that processing further will be useful. To incorporate *the number of paragraphs processed*, we use rank indicator features $f_{s:e}$ such that $f_{s:e}$ is equal to 1 if we have processed between $s$ and $e$ documents, and 0 otherwise.

By relying only on the output scores, the early stopping optimization remains generic and model agnostic – it neither changes the architecture of the QA models nor assumes expert knowledge about the internals of the model. We discuss the classifier training and implementation details of our early stopping classifier in §7.

## 5.2 Offline Neural Encoding

For the second optimization, we analyze the QA pipeline to identify computational units that do not depend on the question. The idea is that if the computation is independent of the question, they do not have to be computed at runtime.

For many QA models, though not all, the neural encoding of documents and the question encoding are performed independently. The QA models do perform a question-guided representation of the documents, but this is done after the encoding step. The three QA models in our study QANet [57], RNet [54], and MReader [35], all perform document and question encoding independently.

Recall that the neural encoding of documents is the key bottleneck in the QA models we study, accounting for about 50% of the time (§4). Our basic optimization, therefore, is to pre-process the documents and turn them into encodings and store them in a file-backed key-value database. At run-time, for a given question, when we iteratively process each document given by the search engine, we use the document id to fetch the pre-computed document encoding and feed it to the subsequent layers in the QA model. Fetching the pre-processed neural encoding is not expensive, requiring 10ms for a paragraph. Our evaluation shows that this offline encoding reduces latency by 2x across all the models we evaluated (§8).

Of course, pre-processing the documents is not free as we trade-off storage for compute. Pre-computing encoding reduces compute latency during runtime, but the pre-computed encodings needs to be stored in disk requiring storage. For the Wikipedia dataset, we require 5GB of disk space to store all the encodings. In our current implementation, we store all the neural encoding in disk. As part of future work, we will investigate the use of memoization techniques to store the neural encodings of documents at runtime and re-use them, especially for documents that are accessed often.

Note that the two DeQA optimizations can also be applied to QA systems deployed on the cloud, but they are less relevant. First, cloud platforms have powerful GPUs that can process several tens of documents in parallel, reducing the effectiveness of early stopping. The offline neural encoding can reduce compute, but our experiments show that pre-computing document encoding reduces QA latency by less than 10% when run on the cloud, suggesting that neural encoding is likely not the bottleneck on cloud GPUs.

## 5.3 Other optimizations

We also explore the use of GPU offloading and batch processing to further optimize QA latency. Since QA models are built using RNNs and self-attention, computation within these models is not easily parallelizable, unlike CNN models. However, we can parallelize paragraph processing over the GPU.

On the mobile board, offloading to GPU and processing multiple paragraphs in parallel is trivial. However, offloading to the phone GPU is harder. This is because existing deep learning frameworks, including TensorFlow, Keras, CNTK, MXNet, PyTorch, Caffe2, do not support GPU offloading on the phone, as of the writing of this paper. Instead we use a data-parallel computation language called RenderScript [14] to rewrite the QA models. Renderscript provides support for GPU offloading and batch processing.

While batch processing on the GPU can significantly reduce QA latency, it also increases power consumption. We analyze this trade-off between latency and power in §8. We find that batch processing by itself significantly increases power consumption, but coupled with the two DeQA optimizations, the power consumption is significantly lowered. This is because, the DeQA optimizations reduce the amount of computation required, making batch processing more energy efficient.

# 6 DeQA MEMORY OPTIMIZATIONS

We design a set of memory optimizations so that existing end-to-end QA systems can be loaded on memory-constrained devices.

## 6.1 Memory requirements of existing QA systems

The QA models—RNet, QANet, and MReader, are reading comprehension models that work on documents returned by a search engine. To make these QA models work end-to-end, we use the TF-IDF search engine used by a popular end-to-end QA system known as DrQA [23]. DrQA also has a deep-learning based QA model, but this model performs poorly compared to our top-ranked QA models, so we don't include it in our study. The study set up and the QA dataset is the same as described in §4.1.

Table 4 shows the memory requirements of the three end-to-end QA systems. The memory is dominated by the search engine. The QA models themselves are roughly comparable to each other.

The search engine requires that the index of the document collection be loaded in-memory. The size of this index is a function of the document collection. For the Wikipedia document collection (12 GB), the size of the index is 13 GB. Even the next generation TX2 board only has 8 GB of memory. Of course, if the document collection is small, the size of the index would be smaller and can fit on the board, but our goal is to not restrict the local data collection over which the user wishes to ask a question.

| Storage Type | Index | MReader | RNet | QANet |
|---|---|---|---|---|
| Storage Size | 13GB | 272MB | 279MB | 270MB |

**Table 4: Memory requirement for running the end-to-end QA service, including the size of the model and the size of the in-memory search engine**

In addition Android sets a hard limit on the heap size for each app [1]. Even high-end mobile devices such as OnePlus 3 have a per-application memory limit of 256MB, even though the phone itself has a larger memory capacity. With this limited app memory, the phones cannot even load the QA models let alone the search engine index.

The result is that existing QA services simply cannot run on either of our mobile devices *as-is*.

## 6.2 Loading partial index using Lucene

One reason for the large memory requirement is the need to load search engine indexes in-memory. One obvious optimization is to only load partial indexes in-memory, which is shown to work for mobile devices [16]. To this end, we use Lucene [2], a standard search engine. We modified a recent version (7.5.0) of Lucene Core to run on Android. Lucene uses a file-backed index, parts of which can be loaded on demand. Lucene performs search on the loaded portions, and combines the results to produce a final ranking.

## 6.3 Moving from document to paragraph retrieval

The problem is that the partial index approach does not completely solve the memory issues of running QA on mobiles. The QA models, by default, take as input a set of relevant documents from the search engine. A document contains a substantial amount of words, and analyzing a large number of documents to score answer phrases can increase memory requirement. In our experiments the QA model was only able to process 50 retrieved documents (roughly 120,000 words) before the memory ran out.

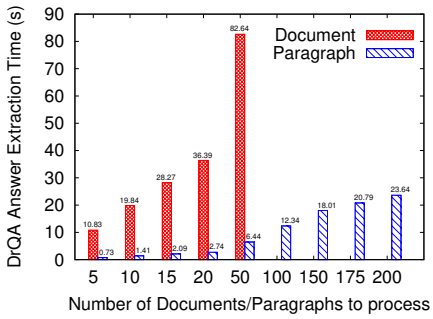



**Figure 4: The average time to retrieve answers when processing paragraphs versus documents.**

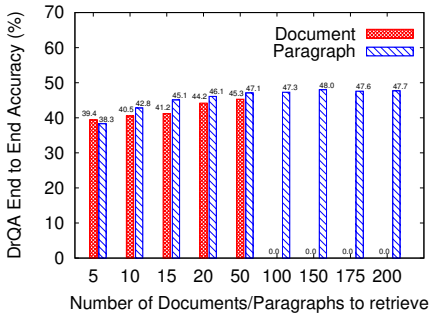



**Figure 5: The accuracy when retrieving answers when processing paragraphs versus documents.**

While it is possible to iteratively process fewer documents in batches, we instead move the unit of analysis to paragraphs. This strategy not only reduces the memory requirements, but also helps improve latency and overall accuracy.

Figures 4 and 5 shows the time to extract the answers, without retrieval time, and the accuracy respectively when the QA model runs over document versus paragraph retrieval. These experiments ran on the mobile board [8]. The figure shows that the QA model can process a far greater number of paragraphs without memory issues, since the number of words that are being processed is much smaller (less than 25,000 words in 200 paragraphs).

However, processing a smaller number of total words does not affect accuracy; in fact, the accuracy of QA when working with paragraphs is much higher compared to documents. This is because, a paragraph that matches many of the question terms is more likely to contain the answer than a document that matches the question terms, potentially in different sections. In our evaluation, we use 150 paragraphs as input to the QA model, because increasing the number of paragraphs further did not improve accuracy commensurately.

## 6.4 Replacing in-memory lookup with a key-value database

The first two approaches bring down the memory requirement of QA systems to run on the board. However, end-to-end QA still cannot run on the phone because each application on the phone cannot use more than 256 MB. The QA models themselves are larger than 256MB (Table 4).

When we break down the model size, we find that the large size of the model is due to the size of the word embeddings. For example, the RNet model size is 279 MB, out of which the word embeddings account for 257.3MB and model weights account for 23.9MB.

Recall that the word embeddings is a mapping of words to a k-dimensional vector. Each phrase is processed by mapping the words in the phrase to its corresponding embedding. The embeddings are loaded in-memory for quick lookup. To process a sentence, the model only needs a small set of embeddings; those corresponding to the words contained in the sentence. Further, the embeddings are static and do not change during the QA process. To reduce memory requirement, we replace the in-memory lookup with an efficient lightweight key-value database called PalDB [10].

The QA models now only contain the model weights, whose size is less than 30 MB for all three models. We also find that the key-value database lookup does not increase latency; look up for 200 random words takes less than 15ms.

## 7 DeQA IMPLEMENTATION

We implement DeQA on the Nvidia Jetson TX2 board and OnePlus 3 Phone (§4.1). We also experiment with the Pixel 2 phone and find that the results are qualitatively similar to the OnePlus 3 phone. The implementation follows the DeQA design (§5 and §6).

**Porting QA models to phones using DeQA**

We port the three state-of-the-art QA models—RNet [54], MReader [35], QANet [57] to use DeQA. Figure 6 shows how the existing QA models are ported to the mobile board [8]. The only change to the QA models is to separate out the embeddings from the model weights using a database (§6). DeQA interfaces with the existing QA model to obtain the features necessary to train the early stopping classifier (§5). These changes do not require modifications to the existing QA models.

There is one engineering challenge in porting QA models to the mobile phone. Many of these models are written in PyTorch [11] which is not compatible with Android devices. We rewrote the original QA models in TensorFlow and Renderscript (for GPU offloading). We verified that the implementations achieve similar performance as reported in the original papers. To each of these models, we add the Lucene-based search engine (§6), to make them end-to-end QA systems.

**Training the early stopping classifier**

We implement early stopping using a binary classifier which outputs stop or continue decisions after processing each paragraph. We create multiple instances for each question, one for each position in the ranked list of passages provided by the search engine. The task is to decide if the model should *stop* processing at this position or *continue* to process more. An instance is labeled 'stop' if the correct answer (or the eventual answer) returned by the system doesn't change after this position. Otherwise it is labeled 'continue'.

We use the XGBoost [24] library to train a logistic gradient boosting decision tree classifier over the features described in §5.1. We set the hyper-parameters for classifier as follows: max tree depth is 6, learning rate is set to 0.3, max delta step is 3. We trained the classifier for 10 epochs. The classifier outputs a score that reflects its confidence on whether the processing should continue or stop. Rather than use the classification decision directly, we learn a threshold that gives a suitable trade-off for accuracy versus latency on a development set. To reduce the inference overhead of the classifier, we use the Treelite [13] library to deploy the classifier on both the board and the phone.

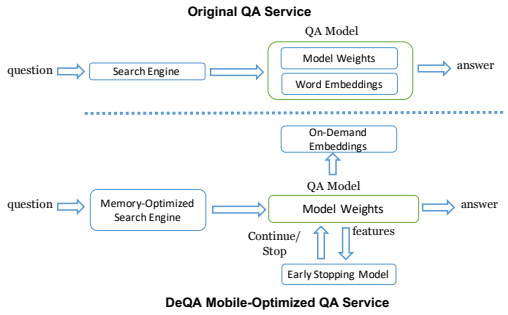



**Figure 6: Porting an existing QA system to the mobile device using DeQA. The original QA model is not changed beyond separating the model weights from the word embeddings.**

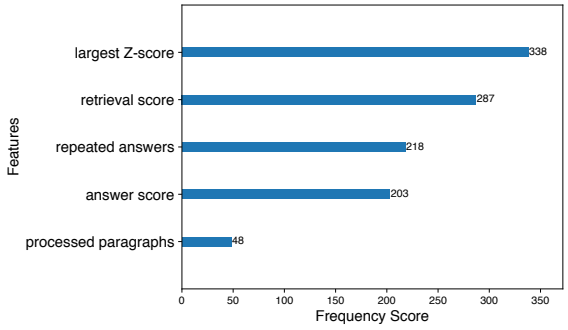



**Figure 7: The frequency of occurrence of different features used to train the early stopping classifier. The frequency of occurence is a proxy for feature importance, with higher values indicating more importance. The Z-score and the document retrieval score are the most important features.**

We use the decision tree classifier to find the importance of the different features. The classifier provides the frequency with which each feature occurs in the decision tree, which is a proxy for feature importance. Figure 7 shows the importance of each feature in terms of frequency of occurrence. Z-score and document retrieval score are most important compared to the other features.

## 8 EVALUATIONS

DeQA's optimizations reduce the memory requirements of the end-to-end QA systems to under 256 MB. This allows the QA systems to run on both the TX2 board and the phone. Our evaluation of three QA systems show that

- DeQA reduces end-to-end QA time on the phone by an average of 16x on the SQuAD dataset with less than 1% loss in accuracy. On an average, DeQA takes less than 5 seconds to answer a question.

- DeQA reduces the energy required to answer a question by 9x. The result is that it takes less energy to answer a question using DeQA than to load a Web page from the Internet.
- On two QA datasets over on-device user content—one dataset over email data and another dataset over cross-app data—DeQA improves QA latency by an average of 14x and 13x on the email and cross-app datasets, respectively (§9).

## 8.1 Experimental Setup and Methodology

**QA datasets** We use the same setup as described in §4.1. In this section, we show the results of evaluating DeQA using the SQuAD dataset over the Wikipedia collection. We also evaluate DeQA over a QA dataset created over two on-device user data collections, which we discuss in §9.

**Comparisons:** We evaluate three end-to-end QA systems that all use a Lucene-based search engine as described in §6. The three systems use RNet [54], QANet [57], and MReader [35], the top-ranked QA models. We compare the performance of the following versions of the QA systems:

- **ExistingQA** – This is the original version of the QA system without any optimizations. This version does not run on mobile devices.
- **DeQA (mem)** – This is the memory-optimized version that can run on mobile devices, but is not optimized for latency.
- **DeQA (ES)** – This refers to the version that uses dynamic early stopping (ES) on top of the memory-optimized version.
- **DeQA (ES + OE)** – This version uses both the optimizations, dynamic early stopping and Offline Neural Encoding (OE).
- **DeQA** – This is the fully optimized version with all of the memory and latency optimizations. The latency optimizations include ES, OE, and Batch Processing.

**Devices** All experiments are conducted on the OnePlus 3 phone and the TX2 board as described in §4. We measure energy using Monsoon Power Monitor [7].

## 8.2 DeQA Latency, Energy, and Accuracy

Figure 8a and Figure 8b shows the performance of the three QA systems on the phone and the board, respectively. The existing QA systems cannot run on the device, and is shown using the infinity bar. DeQA (mem), the memory-optimized version that can run on the phone but is not optimized for latency, takes over a minute to answer a single question on the phone.

**Latency:** DeQA reduces the time to answer a question by an average of 16x on the phone and an average of 6x on the board. Recall that on a desktop, QA takes an average of 4 seconds (Table 2). In effect, with DeQA, the QA latency on the mobile is comparable to that of a desktop.

As a further point of comparison, according to a 2017 study, the average time to load a mobile Web page over LTE is 5 seconds [51]; loading a heavy page takes over 10 seconds. In other words, DeQA can support local question answering with speeds comparable to (or even less than) searching over the Internet.

**Accuracy:** Accuracy is the fraction of questions for which we get at least one correct answer in the top 5 ranked candidate answers returned by the system. This evaluation is focused on factoid questions, where typically a question has only one correct answer. Figure 8c shows that the DeQA optimization does not come at a cost of QA accuracy. The adapted systems work within 1% of the accuracy of the original QA system. The accuracy results are same for both the board and the phone, since they both run the same QA model.

**Energy:** Figure 9 shows that DeQA reduces energy consumption of answering a question to 6J on an average. Without DeQA, the energy consumption is over 50J across all three QA systems. For these experiments, we averaged the energy consumption over 10 questions chosen randomly.

As a point of comparison, the median energy to load the top 100 Web page on mobile is 12J [22]. The energy consumption with DeQA includes the end-to-end energy, including retrieving paragraphs from the search engine, running the QA model, using the early stopping classifier, retrieving neural encodings from disk, and batching.

**Incremental Indexing:** We also separately evaluated the energy (and the latency) to incrementally index content. This is required because users will continue accessing new content that need to be indexed locally. Incremental indexing only required 166mJ per document and took about 13ms per document.

## 8.3 DeQA components

DeQA has two key optimizations–the early stopping (ES) and Offline Neural Encoding (OE). DeQA couples these with a standard batch processing technique. Figures 10 shows the improvements (on the phone) of each technique in DeQA optimization when applied iteratively – early stopping (ES) by itself, combining ES with OE, and three optimizations together. The two optimizations combined provide a 4x benefit and the batch processing provides a further 4x benefit.

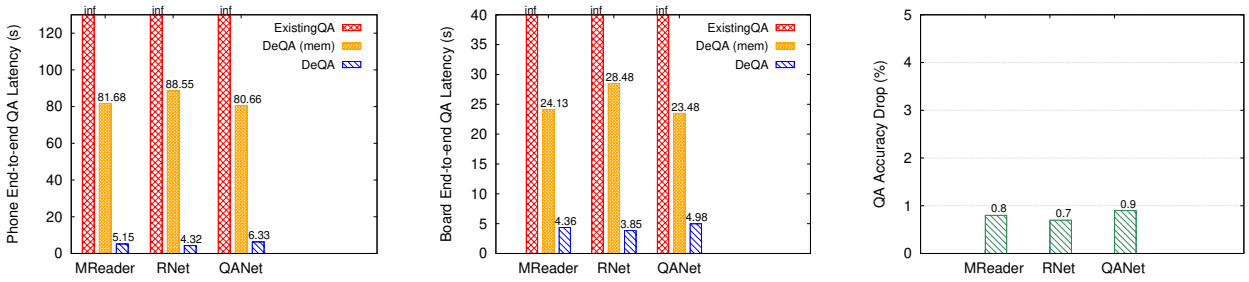
We applied the optimizations in other combinations and found that applying early stopping and offline encoding followed by batch processing provides the most benefits both in terms of performance and power. Although it appears that batch processing is powerful, we show in §8.4 that batch processing applied by itself without the other DeQA optimizations require 3 times as much energy. This is because DeQA optimizations reduce the amount of processing required, which reduces the burden on batch processing.

## 8.4 Digging deeper into the optimizations

We explore the optimizations further to understand their trade-offs.

**Why dynamic early stopping?** Early stopping predicts when further processing is not useful. A simpler strategy is to always stop processing paragraphs after a fixed number of paragraphs. The problem is that if you process too few paragraphs, the answer extraction time reduces but the accuracy drops (and vice-versa if you process too many).

In Figure 11 we show that this simple strategy of a fixed stopping threshold is not enough. We vary the stopping threshold and show results for RNet and MReader, but the results are similar for QANet. The RNet model takes 18 seconds to extract answers when processing 100 paragraphs but incurs a 2% drop in accuracy compared to using 150 paragraphs. If RNet is forced to use only 20 paragraphs




**(a) End-to-end QA latency across the three QA systems on the phone.**

**(b) End-to-end QA latency across the three QA systems on the board**

**(c) DeQA accuracy drop compared to the original QA systems.**

**Figure 8: Latency benefits and accuracy of DeQA: End-to-end latency reduces on an average of 16x on the phone, and on average of 6x on the board with less than a 1% drop in overall accuracy. In (a) and (b) *inf* indicates that the QA system could not be loaded on the device. In (c) the accuracies are the same on the phone and the board, so we only report on one set of accuracies.**

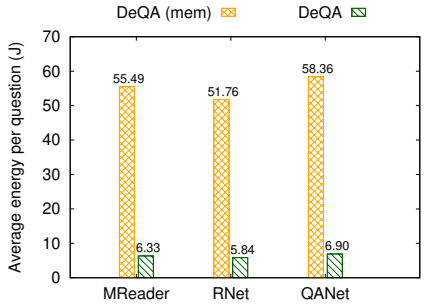



**Figure 9: Energy Consumption (on the phone) with and without DeQA across the three QA systems.**

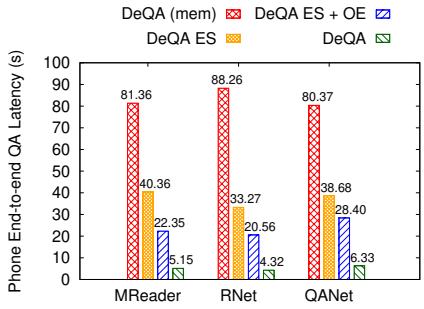



**Figure 10: Latency improvements with DeQA when optimizations are applied iteratively.**

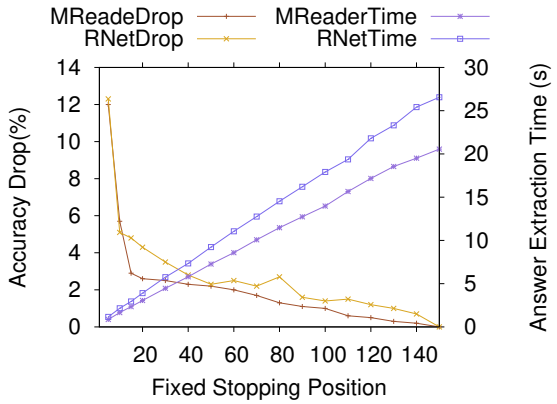



**Figure 11: QA accuracy drop and answer extraction time when using a fixed stopping algorithm rather than DeQA's dynamic early stopping algorithm. DeQA has a better answer extraction time/accuracy trade-off compared to using any of the fixed stopping threshold.**

then it finishes faster, within 3.2 seconds but incurs a higher drop in accuracy, by 4%.

DeQA's early stopping can balance the accuracy and time trade-off by stopping early if further processing is unlikely to be useful. As a result, the answer extraction time with Early Stopping is around 3.5 seconds (similar to processing only 20 paragraphs), but the accuracy drops less than 1%. The figure only shows the answer extraction time, excluding retrieval time.

**Energy versus performance trade-off with batch processing** Figure 12 shows the energy consumption when using batching alone (without the other optimizations) for different batch sizes. For comparison, the average DeQA energy consumption was 6J (as shown in Figure 9), which is 3x smaller than the energy consumption of only batching, even if both schemes use a batch size of 16. Further, batching alone only provides a 2x benefit over the default DeQA (mem) in terms of latency. DeQA optimizations reduce the amount of processing, making batching useful in terms of latency and energy.

## 9 EVALUATING DeQA OVER ON-DEVICE USER DATA

In the previous section, we evaluate DeQA over a Wikipedia collection. We next evaluate DeQA in the context of user data collected on their devices. Unfortunately, there are no publicly available QA datasets over on-device data because it is difficult to obtain large scale data from users due to privacy concerns.

To circumvent the privacy issues, we create our own QA dataset over two data sources. The first data source is a publicly available

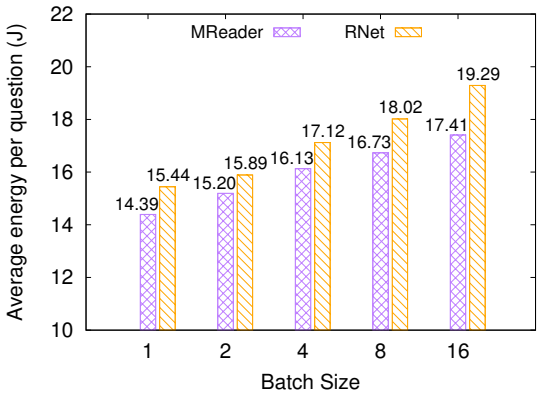



**Figure 12: Energy consumption vs batching size on Phone GPU for MReader and RNet systems on the phone**

collection of emails from the employees of Enron [39]. The second is a cross-app data source that we collect from two users. We collect on-device content as the users interact with five different apps on their mobile phones for 1 week. The data however remains within their devices (not available to anyone else) and the users themselves create question and answer pairs for evaluation. Because the on-device QA datasets are much smaller, we do not retrain the QA model. The model is trained over the SQuAD dataset as before and is applied to the new datasets.

## 9.1 On-device QA: Single App

**Creating a QA dataset over a email collection** The Enron Email Dataset is a collection of emails from employees of an organization that has been publicly used for many types of language related problems such as text classification [39]. We randomly sample a single user from this collection, and create a set of questions that can be answered from this user's emails. The collection consisted of 3034 emails.

Our procedure to create the QA dataset is similar to the one used to create the standard SQuAD dataset from the Wikipedia collection. Specifically, we hired six annotators who each created twenty question answer pairs yielding a set of 120 questions covering 60 different emails. Each annotator was given ten different emails. For each email they generated two questions, whose answers would be an exact phrase within the email. Similar to the SQuAD data construction, this procedure ensures that the answers are spans within the emails and that each question is guaranteed to have a precise answer. The annotators were not given any additional instruction. To conduct the end-to-end QA experiments, we index the 3034 emails consisting of 8825 paragraphs for this single user.

**Characteristics of the email QA dataset** Without any optimizations, the time to answer a question over this new dataset is not much smaller compared to that over the Wikipedia collection. Across the three systems, answering a question takes an average of 68 seconds on this dataset on the phone. This result is surprising given that the email collection is orders-of-magnitude smaller compared to the Wikipedia collection.

The takeaway is that the bottleneck in QA is not the size of the collection, but the number of paragraphs that need to be processed to get the correct answer. We conducted a similar analysis as Figure 2 to analyze how the answers are distributed across the top ranked paragraphs. If the answers can always be found in the top few paragraphs, then the QA model need not process a lot of documents. However, similar to the SQuAD dataset, the answers are distributed across the paragraphs in this new dataset. We find that the accuracy does not improve significantly beyond processing 100 paragraphs. So we restrict the number of paragraphs processed by the QA model to 100.

| board | DeQA (mem) | DeQA |
|---|---|---|
| RNet | 25.497 s | 4.1 s |
| MReader | 21.395 s | 3.2 s |
| QANet | 22.651 s | 4.6 s |

**Table 5: QA latency with and without DeQA over the QA dataset defined over the email collection on the TX2 board.**

| phone | DeQA (mem) | DeQA |
|---|---|---|
| RNet | 72.374 s | 4.917 s |
| MReader | 68.932 s | 4.285 s |
| QANet | 65.583 s | 5.461 s |

**Table 6: QA latency with and without DeQA over the QA dataset defined over the email collection on the phone.**

**Evaluating DeQA over the email dataset** Table 5 and Table 6 show the latency benefits of applying DeQA on this email QA collection. As before, DeQA reduces latency to less than 5 seconds in most cases on both the board and the phone. On the phone, DeQA provides an average speedup of 14x and on the board, DeQA provides an average speedup of 6x with an accuracy drop of 1.3%. Overall, these results demonstrate that the DeQA optimizations are effective for question answering over user content.

## 9.2 On-Device QA: Cross Apps

**Collecting data across apps** We collect on-device data from two users as they interact with five different apps for 1 week. We chose the five apps (a) such that the apps are used by the users frequently, and (b) the apps contain a good mix of social media, personal, and informational content. To this end, the five apps we choose are: BBC News, LinkedIn, Twitter, TripAdvisor, and Notes.

For each app, the users interacted with the apps as they normally would. For example, scrolling the page to read news/tweets feed, entering texts for note taking, or simple browsing. We install an app crawler service to extract the textual content of the screen. We store the crawled content and index it for use by the search engine in the QA system.

**Creating a QA dataset over the cross-app collection** As before, we ask the two users to randomly sample crawled documents from this data collection to create 100 questions (10 per app per user) that can be answered from these documents.

We ask the same two users to also perform answer annotation because we did not want a third-party looking at the user data. The answers would be an exact phrase within the document. To conduct the end-to-end QA experiments, we index the content from the five apps on each user's mobile device. In total, we indexed 14524 paragraphs across both the users.

**Evaluating DeQA over the cross-app dataset** Table 7 shows the latency benefits of applying DeQA on the cross-app data collection. Compared to the Wikipedia articles and email dataset, the cross-app

| phone | DeQA (mem) | DeQA |
|---|---|---|
| RNet | 54.783 s | 3.639 s |
| MReader | 49.401 s | 3.476 s |
| QANet | 51.132 s | 4.842 s |

**Table 7: QA latency with and without DeQA over the QA dataset defined over the cross-app data.**

| Indexing Type | time per doc | total docs | total time |
|---|---|---|---|
| scratch index | 2.78 ms | 8024 docs | 22.307 s |
| incremental index | 2.61 ms | 1000 docs | 2.612 s |

**Table 8: The time to index the cross-app data for the first time and then incrementally index data on the phone for one of the two users.**

data have much shorter paragraphs. The result is that even without any latency optimizations, answering a question takes an average of 51 seconds, compared to an average of 68 and 88 seconds on the email and Wikipedia datasets, respectively. However, DeQA reduces latency to less than 5 seconds for an average speedup of 13x with an accuracy drop of 1.5%.

**Indexing cross-app data:** Table 8 shows the time taken to index the entire cross-app data on one of the users device. It took about 22 seconds to index all the cross-app data from scratch. Subsequently, each document takes only 2.61 ms to be indexed incrementally. In summary, indexing on-device content on the phone incurs negligible overheads.

# 10 RELATED WORK

## 10.1 Question Answering Systems

Question answering has a rich history in the NLP, information retrieval, and AI communities and has been studied in various domains (c.f. [20, 30, 40]). Their focus has largely been to improve the accuracy of various aspects of the QA systems. Recent work in the systems community has studied the performance of QA systems (e.g. OpenEphyra) in the context of data center deployments [34]. Qme! [46] is a speech-driven QA system for mobile devices. The focus of Qme! is to reduce the noise in speech recognition and perform tight integration between speech recognition and search. However, Qme! does not run a QA model and instead retrieves answers from a database. Running a QA model on the phone is more computationally intensive and is the focus of our work.

Recent advances in deep learning have led to development of many neural network based architectures for question answering [23, 25, 55, 56] which have outperformed feature-based methods. Most of these work focus on extracting an answer, assuming the answer containing passage is given as input. DrQA [23] is one of the few end-to-end QA systems that uses deep learning. In this work, we show how an end-to-end QA system with a deep learning QA model can be optimized to run on mobile platforms.

We have deliberately designed DeQA to not depend on model specifics or internals. This allows DeQA optimizations to be compatible with new QA models variants. For instance, a recent development in NLP is to use large models that use many layers of transformer blocks (e.g., OpenAI GPT [49] and BERT [27]). Recall that QANet, a QA model we optimize using DeQA also uses similar transformer blocks but does not use several layers of the blocks. The DeQA optimizations are still relevant with these large models, even though these models introduce other significant memory challenges that require further optimizations. We leave the optimizations of these large NLP models for future work.

## 10.2 Optimizing deep learning for Mobile

Deep learning models, in most cases, cannot run *as is* on mobile devices. Existing optimizations focus on reducing the model sizes (compression) and parallelizing model computations (decomposition). These optimizations were designed mainly for DNN and CNN architectures used in vision and other sensing applications [19, 31, 41, 42], with large model sizes and compute patterns that are amenable for parallelization.

Mobile GPUs provide another avenue to optimize the compute involved in these models. DeepMon [37] and CNNDroid [47] both show that a mobile GPU can be used to improve the CNN/DNN execution time, in some cases getting more than a ten-fold speedup for AlexNet, an image recognition model.

However, RNN optimizations, used by most QA models, are not well studied in the context of adapting to mobile devices as CNNs. Some recent studies focus on optimizing RNNs for desktop GPUs [15]. MobiRNN[21] optimizes small RNN models used for activity recognition. In this work, we build on these ideas and deliver QA specific optimizations.

# 11 CONCLUSIONS

Building a question answering system that operates over local device content requires running deep learning based QA models on resource constrained environments. End-to-end QA systems today have prohibitively large memory requirements and high latencies making them unusable. Our benchmark study showed that bulk of processing is spent in encoding documents and the overall latency is mostly due to processing a large number of documents for each question. Based on these findings, we designed set of latency and memory optimizations that address these inefficiencies in a way that does not rely on the model structure or the internals. Our predictive early stopping model automatically identifies when further processing is unlikely to improve the model's answer, thereby reducing the amount of data that is processed by the model. A further latency optimization pre-computes partial neural representations of documents offline and uses them on-demand. Our memory optimizations reduce the overall in-memory data using on-demand loading of various components of the QA model backed by appropriate storage mechanisms. Our evaluation of the resulting optimized QA system shows that it is both efficient and effective for resource-constrained mobile devices.

# ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Matthai Philipose, for their insightful comments and their help towards improving the paper. We thank Sruti Kumari and Mohit Marwari for their contributions in implementing the on-device cross-app data collection and evaluating DeQA on this collection. We thank Heeyoung Kwon for his help with understanding the QA pipeline. This work was supported in part by NSF grant CNS-1717973, IIS-1815358, and VMware.

# REFERENCES


[1] Android app memory restriction. https://developer.android.com/topic/performance/memory-overview.
[2] Apache lucene. https://lucene.apache.org/.
[3] Cuda. https://developer.nvidia.com/cuda-downloads.
[4] Gameface labs hmc. https://www.tomshardware.com/news/gameface-labs-standalone-steamvr-headset,37112.html.
[5] Intel sgx. https://software.intel.com/en-us/sgx.
[6] Magic leap vr. https://www.tomshardware.co.uk/magic-leap-tegra-specs-release,news-58814.html.
[7] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.
[8] Nvidia jetson tx2 board. https://developer.nvidia.com/embedded/buy/jetson-tx2.
[9] Oneplus 3. https://www.oneplus.com/3.
[10] Paldb. https://github.com/linkedin/PalDB.
[11] Pytorch. http://pytorch.org/.
[12] Standard score. https://en.wikipedia.org/wiki/Standard_score.
[13] Treelite : model compiler for decision tree ensembles. https://github.com/dmlc/treelite.
[14] Android. Renderscript. https://developer.android.com/guide/topics/renderscript/compute.html. accessed 8-April-2017.
[15] J. Appleyard, T. Kocisky, and P. Blunsom. Optimizing Performance of Recurrent Neural Networks on GPUs. *ArXiv e-prints*, Apr. 2016.
[16] A. Balasubramanian, N. Balasubramanian, S. J. Huston, D. Metzler, and D. J. Wetherall. Findall: a local search engine for mobile phones. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 277–288. ACM, 2012.
[17] P. Baudis and J. Sedivý. Modeling of the question answering task in the yodaqa system. In *CLEF*, 2015.
[18] J. Berant, A. Chou, R. Frostig, and P. Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, 2013.
[19] S. Bhattacharya and N. D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, SenSys '16, pages 176–189, New York, NY, USA, 2016. ACM.
[20] E. Brill, S. Dumais, and M. Banko. An analysis of the askmsr question-answering system. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 257–264. Association for Computational Linguistics, 2002.
[21] Q. Cao, N. Balasubramanian, and A. Balasubramanian. Mobirnn: Efficient recurrent neural network execution on mobile gpu. In *EMDL@MobiSys*, 2017.
[22] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi. Deconstructing the energy consumption of the mobile page load. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):6:1–6:25, June 2017.
[23] D. Chen, A. Fisch, J. Weston, and A. Bordes. Reading wikipedia to answer open-domain questions. In *ACL*, 2017.
[24] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.
[25] C. Clark and M. Gardner. Simple and effective multi-paragraph reading comprehension. *arXiv preprint arXiv:1710.10723*, 2017.
[26] Y. Cui, Z. Chen, S. Wei, S. Wang, T. Liu, and G. Hu. Attention-over-attention neural networks for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 593–602, 2017.
[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.
[28] P. Dmitriev, P. Serdyukov, and S. Chernov. Enterprise and desktop search. In *WWW*, 2010.
[29] J. L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.
[30] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, et al. Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79, 2010.
[31] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *ArXiv e-prints*, Oct. 2015.
[32] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *ICLR*, 2016.
[33] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. MobiSys '16, pages 123–136, 2016.
[34] J. Hauswald, M. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. N. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.
[35] M. Hu, Y. Peng, Z. Huang, X. Qiu, F. Wei, and M. Zhou. Reinforced mnemonic reader for machine reading comprehension. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4099–4106. International Joint Conferences on Artificial Intelligence Organization, 7 2018.
[36] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel. Chiron: Privacy-preserving machine learning as a service. *CoRR*, abs/1803.05961, 2018.
[37] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. MobiSys, 2017.
[38] B. Klimt and Y. Yang. The enron corpus: A new dataset for email classification research. In *Proceedings of the 15th European Conference on Machine Learning*, ECML'04, pages 217–226, Berlin, Heidelberg, 2004. Springer-Verlag.
[39] B. Klimt and Y. Yang. The enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pages 217–226. Springer, 2004.
[40] C. C. T. Kwok, O. Etzioni, and D. S. Weld. Scaling question answering to the web. In *ACM Trans. Inf. Syst.*, 2001.
[41] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*, IPSN '16, 2016.
[42] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 283–294. ACM, 2015.
[43] X. Liu, Y. Shen, K. Duh, and J. Gao. Stochastic answer networks for machine reading comprehension. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1694–1704. Association for Computational Linguistics, 2018.
[44] M. Martinez, A. Roitberg, D. Koester, R. Stiefelhagen, and B. Schauerte. Using technology developed for autonomous cars to help navigate blind people. In *Computer Vision Workshop (ICCVW), 2017 IEEE International Conference on*, pages 1424–1432. IEEE, 2017.
[45] A. Miller, A. Fisch, J. Dodge, A.-H. Karimi, A. Bordes, and J. Weston. Key-value memory networks for directly reading documents. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1400–1409. Association for Computational Linguistics, 2016.
[46] T. Mishra and S. Bangalore. Qme!: A speech-based question-answering system on mobile devices. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 55–63, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.
[47] S. S. L. Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *ACM Multimedia*, 2016.
[48] B. Pan, H. Li, Z. Zhao, B. Cao, D. Cai, and X. He. Memen: multi-layer embedding with memory networks for machine comprehension. *arXiv preprint arXiv:1707.09098*, 2017.
[49] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/language-unsupervised/language_ understanding_paper. pdf*, 2018.
[50] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100, 000+ questions for machine comprehension of text. In *EMNLP*, 2016.
[51] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 390–403, New York, NY, USA, 2017. ACM.
[52] S. Salant and J. Berant. Contextualized word representations for reading comprehension. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, volume 2, pages 554–559, 2018.
[53] W. Wang, M. Yan, and C. Wu. Multi-granularity hierarchical attention fusion networks for reading comprehension and question answering. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1705–1714, 2018.
[54] W. Wang, N. Yang, F. Wei, B. Chang, and M. Zhou. Gated self-matching networks for reading comprehension and question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 189–198, 2017.
[55] D. Weissenborn, G. Wiese, and L. Seiffe. Making neural qa as simple as possible but not simpler. In *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017)*, pages 271–280, 2017.
[56] C. Xiong, V. Zhong, and R. Socher. Dynamic coattention networks for question answering. *CoRR*, abs/1611.01604, 2016.
[57] A. W. Yu, D. Dohan, Q. Le, T. Luong, R. Zhao, and K. Chen. Qanet: Combining local convolution with global self-attention for reading comprehension. In *International Conference on Learning Representations*, 2018.

[58] S. Yu, S. R. Indurthi, S. Back, and H. Lee. A multi-stage memory augmented neural network for machine reading comprehension. In *Proceedings of the Workshop on Machine Reading for Question Answering*, pages 21–30, 2018.

[59] X. Zeng, K. Cao, and M. Zhang. Mobiledeeppill: A small-footprint mobile deep learning system for recognizing unconstrained pill images. MobiSys, 2017.