Question Answering Using Regularized Match-LSTM and Answer Pointer

Ellen Blaine*
Department of Computer Science
Stanford University
Stanford, CA 94305
eblaine@stanford.edu

Debnil Sur*
Department of Computer Science
Stanford University
Stanford, CA 94305
debnil@stanford.edu

Abstract

Automated reading comprehension is an important problem in natural language processing. The Stanford Question Answering Dataset (SQuAD) is a convenient set of questions and crowdsourced answers to utilize for evaluation of QA systems. In this paper, we implement a version of an architecture proposed by Wang and Jiang (2016) based on match-LSTM [9], a model used for textual entailment, and answer candidate generation based on Pointer Net (Vinyals et al., 2015) [7]. We extend the original implementation with an investigation of the effect of regularization methods on this task. Specifically, we argue that any amount of regularization via dropout improves test performance because it prevents overfitting; however, we also note that varying the dropout probability does not significantly change performance, provided it is sufficiently far from 1 (< 0.8).

1 Introduction

1.1 Motivation

Machine comprehension is one of the most challenging tasks in natural language processing; it provides one of the best indicators of an intelligent system's holistic "understanding" of documents. While this ability can be assessed in many ways, some recent research has focused on answering questions for evaluation. Here, the machine is first presented a specific piece of content, such as a news article or story. It then answers questions regarding that content.

1.2 Problem Definition

Most work in this vein uses multiple-choice questions in evaluation. In this approach, an agent is presented some content, asked a question, and must choose a correct answer from a set of provided candidate answers [2, 5]. While this provides some assessment of text comprehension, a more open-ended answering method would theoretically be more difficult and thus provide better assessment. The Stanford Question Answering Dataset (SQuAD) is a dataset that follows this approach [4]. A correct answer can be any sequence of tokens from a given text. (This can be compactly represented through the start and end indexes of the answer.) Since its questions and answers were created by humans through crowdsourcing, it is also more realistic than other datasets generated automatically in Cloze style [1, 2]. Thus, this dataset provides a good basis to design a question-answer system and thus study machine comprehension of text.

1.3 Previous Work

Previous traditional solutions utilize complex NLP pipelines that involve many steps of linguistic analysis, such as syntactic and parsing, named entity recognition, and question classification. Advances in neural nets in NLP have led to end-to-end neural architectures for many NLP tasks. Uses of NLP for question answering have typically focused on previous machine comprehension datasets. As a result, they either utilize a multiple-choice setting [2] or assume the answer is a single token [1]. This makes those approaches unsuitable for the task outlined in SQuAD.

1.3.1 Encoding

Implementing a neural net architecture for SQuAD has two major requirements outlined by Wang and Jiang [9]. First, it must learn and combine rich hidden representations of a passage and question. A significant simplifying step is the observation that many questions in SQuAD are paraphrases of sentences in the earlier test. Consequently, we adopt a match-LSTM model developed for textual entailment [8]. In a textual entailment task, a model must predict whether the premise entails the hypothesis. Here, an LSTM processes the context (the possible hypotheses) sequentially and obtains a weighted vector representation of the question (the premise). It then combines this weighted question with a vectorized representation of the current context sequence and inputs it into an LSTM, called the match-LSTM. This LSTM sequentially aggregates these matchings and makes a final prediction over them.

We also considered other effective encoding approaches in the literature. Ultimately, our decision to proceed with Match LSTM was guided by resource constraints and personal preference, rather than marked performance improvements. For one, Xiong et al. utilize a co-attentive encoder to capture the relationship between question and context [10]. Though it had slightly higher F1 and EM than the original Match-LSTM paper, this model concatenated multiple fusings of question and context. We worried that the graph-computation model of Tensor Flow would be less compatible with this approach. Specifically, we prioritized quick train cycles (and lower memory usage) to allow for greater hyper-parameter tuning, embedding optimization, and similar time and memory-intensive operations.

Seo et al. employ a bi-directional attention flow mechanism to achieve a question-aware representations for the passage [6]. Similarly, Match-LSTM had better memory usage and train speed than this model. In addition to the reasons above, some of the most important features in this model (and others) could not be implemented in the given framework. For example, the authors employ a character-level convolutional neural network to pre-process their data and create a richer initial hidden representation. For time and space efficiency, we utilize a vocabulary dictionary, in which words are represented by integer keys, to represent the passage and context. This sacrifices the character-level granularity that provides some of BiDAF's performance improvements. Similarly, the Sherlock computational cluster, which we were forced to utilize, only had TensorFlow 0.12.1; many of the optimizations in BiDAF could only be efficiently implemented in TensorFlow 1.0. These major concerns—memory, time, and efficient pre-processing—led us away from other successful approaches to SQuAD encoding.

1.3.2 Decoding

The second major task is decoding this hidden representation. A model must learn an answering method to find the most probable start and end answer tokens. To do this, Wang and Jiang adopt the Pointer-Net (Ptr-Net) model developed by Vinyals et al. [7], which enables the prediction of tokens from the input sequence rather than the larger vocabulary. They use this in two ways: a sequence model, which generates a sequence of tokens in the paragraph that is the most probable answer, and a boundary model, which generates the start and end tokens of the answer. We utilize the boundary model for several reasons. First, the assignment specifications only require the creation of a probable substring, which can be generated through start and end indexes. Second, the graph-based model of TensorFlow constructs all required parameters, so only requiring two vectors holding probability distributions (as opposed to one for every possible location in a paragraph) significantly reduces memory usage and computational time. Third, the experiments in the original Match-LSTM paper had 6.4% better exact match (EM) and 3% better F1 score (the harmonic mean of precision and recall).

We also considered other approaches to identifying maximal indices in the literature. A concern we had with the answer pointer layer was the ”greediness” of its approach: it simply chooses the most probable answering section in the passage after learning a hidden relationship between question and context. This can easily get stuck in local maxima, rather than returning a more optimal answer. Dynamic decoding approaches have been proposed to resolve this issue. Xiong et al. utilize an iterative technique that alternates between predicting the start and end indices of an answer within a context [10]. The authors combine Highway and Maxout Networks to create a novel Highway Maxout Network that pools across different networks to better predict spans. While well-motivated, the slight improvement in prediction performance did not seem to justify the creation and usage of an extra neural network in decoding. Rather, the much simpler, single call to an LSTM cell in the Answer Pointer saves both train time and memory utilization. Given our two-week development cycle, we prioritize efficient use of resources over all else.

2 Method

Our model implements a modified version of the Match-LSTM with Answer Pointer architecture proposed in Wang & Jiang (2016) [9].

2.1 Data Preprocessing

To decrease training time, we discarded training question and context paragraphs longer than 30 and 300 tokens, respectively. This amounts to 0.07% of questions and 1.6% of context paragraphs from the training set in total. When evaluating on development or test sets, we simply truncate questions and contexts to those lengths. See the histogram in Figure 1 for a sense of the distribution of context and question lengths.

To allow batch-processing of training data, we pad all remaining question and context paragraphs to the lengths specified above with zeros.

Finally, to initialize the model’s vocabulary, we use all available GloVe word embeddings of length 300 (Pennington et al., 2014) [3]. Words that appear in the training set but are absent from GloVe are mapped to random vectors (these are generally infrequent proper nouns).

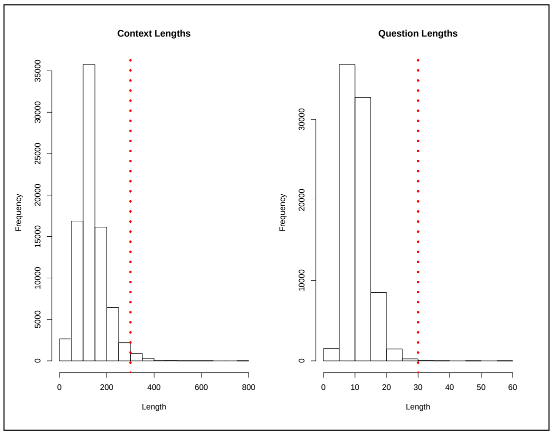

Figure 1: Histogram of context paragraph and question lengths. Red lines indicate threshold of question and context lengths discarded during training.

2.2 LSTM Preprocessing

Before computing the match-LSTM or answer pointer layers, we run a forward LSTM over the question and context paragraph:

$$\mathbf{H}^c(\mathbf{C}) = \overrightarrow{LSTM}(\mathbf{C}) \qquad\qquad \mathbf{H}^q(\mathbf{Q}) = \overrightarrow{LSTM}(\mathbf{Q})$$

The LSTM outputs $\mathbf{H}^c \in \mathbb{R}^{C \times l}$ and $\mathbf{H}^q \in \mathbb{R}^{Q \times l}$ are hidden representations of the question and answer, where C and Q are the maximum context paragraph and question lengths considered, respectively, and l is the size of the hidden state used.

2.3 Match-LSTM

The next layer is a biLSTM run over the context, using the question as a premise. The hidden representations of the question and context, $\mathbf{H}^q$ and $\mathbf{H}^c$, are used to compute an attention vector for each word in the context, conditioned on the question. The motivation for doing so is that the answers in the SQuAD dataset are frequently very syntactically similar to the questions asked. In particular, for every position i in the passage, we compute the following attention weight vector $\overrightarrow{\mathbf{a}}_i$:

$$\begin{aligned}\overrightarrow{\mathbf{G}}_i &= \tanh(\mathbf{H}^q\mathbf{W}^q + (\mathbf{h}_i^c\mathbf{W}^c + \overrightarrow{\mathbf{h}}_{i-1}^r\mathbf{W}^r + \mathbf{b}^c) \otimes \mathbf{e}_Q) \\ \overrightarrow{\mathbf{a}}_i &= \text{softmax}(\overrightarrow{\mathbf{G}}_i\mathbf{w} + b \otimes \mathbf{e}_Q)\end{aligned}$$

where $\mathbf{W}^q, \mathbf{W}^c, \mathbf{W}^r \in \mathbb{R}^{l \times l}$; $\mathbf{b}^c, \mathbf{w} \in \mathbb{R}^{l \times 1}$; and $b \in \mathbb{R}$, and the $\otimes\mathbf{e}_Q$ operator copies (or broadcasts, depending on implementation) a vector or matrix Q times into rows of a new matrix for addition. $\mathbf{h}_i^c$ corresponds to the hidden representation of token i in the context paragraph and $\overrightarrow{\mathbf{h}}_{i-1}^r$ is the previous hidden state in the match-LSTM.

This attention vector is then used to generate a vector $\overrightarrow{\mathbf{z}}_i$, which concatenates both the hidden representation of the current context paragraph token and the hidden question representation, weighted using the attention vector for that token:

$$\overrightarrow{\mathbf{z}}_i = \left[\mathbf{h}_i^p, \mathbf{H}^{q\intercal}\overrightarrow{\mathbf{a}}_i\right]$$

Finally, we feed $\overrightarrow{\mathbf{z}}_i$ as input to the forward LSTM. The reverse direction is defined similarly.

$$\overrightarrow{\mathbf{h}}_i^r = \overrightarrow{LSTM}(\overrightarrow{\mathbf{z}}_i, \overrightarrow{\mathbf{h}}_{i-1}^r)$$

Let $\overrightarrow{\mathbf{H}}^r$ be the stacked hidden states at all forward time-steps of the forward LSTM and $\overleftarrow{\mathbf{H}}^r$ be the backward hidden states. This layer returns these matrices concatenated into $\mathbf{H}^r \in \mathbb{R}^{C \times 2l}$:

$$\mathbf{H}^r = [\overrightarrow{\mathbf{H}}^r, \overleftarrow{\mathbf{H}}^r]$$

To prevent overfitting, we regularize this layer via dropout, removing output layers from $\mathbf{H}^r$ with probability 0.5.

2.4 Answer Pointer

This layer uses a forward LSTM to generate probability distributions, β_{start} and β_{end}, for the answer start and end tokens over the context paragraph from the match-LSTM output $\mathbf{H}^r$. We initialize a zero-state and unroll the network twice to compute these distributions as follows:

$$\begin{aligned}\mathbf{F}_i &= \mathbf{H}^r\mathbf{V} + (\mathbf{h}_{i-1}^a\mathbf{W}^a + \mathbf{b}^a) \otimes \mathbf{e}_C \\ \beta_i &= \text{softmax}(\mathbf{F}_k\mathbf{v} + c \otimes \mathbf{e}_C)\end{aligned}$$

where $i \in \{1, 2\}$; $\mathbf{V} \in \mathbb{R}^{2l \times l}$; $W^a \in \mathbb{R}^{l \times l}$; $\mathbf{b}^a, \mathbf{v} \in \mathbb{R}^{l \times 1}$; and $c \in \mathbb{R}$. We generate each new hidden vector $\mathbf{h}_i^a \in \mathbb{R}^l$ with the above result:

$$\mathbf{h}_i^a = \overrightarrow{LSTM}(\mathbf{H}^r\beta, \mathbf{h}_{i-1}^a)$$

We then minimize the cross-entropy loss between each of the two β_i and two one-hot vectors identifying the true start and end tokens.

There is one caveat to this approach. It is necessary to normalize a β_i using softmax() if it is immediately passed to the next step of the LSTM. However, the TensorFlow softmax cross-entropy loss function assumes *un*normalized distributions. An easy solution would be to return unnormalized β vectors to the loss function. This is not ideal, because the distributions predicted from zero-padded contexts would assign positive probabilities to indices beyond the length of the paragraphs. Instead, we construct a sequence mask, take its log, and add it to the β vectors before the softmax call, such that when any softmax normalization occurs (whether in a loss function or before passing to the LSTM call), the result assigns zero probability to the padded tokens.

2.5 Span Prediction

The simplest prediction approach is to return the indices corresponding to the greatest values in β_{start} and β_{end} as the starting and ending token indices. However, this tends to predict very long answer spans in practice. Instead, we predict the most likely spans within a length of 15 words in the paragraph, using the exponentiated sum of their scores (to weight positive scores more highly and allow for numerical stability). As illustrated in Figure 2, the overwhelming majority of answers are within this length range.

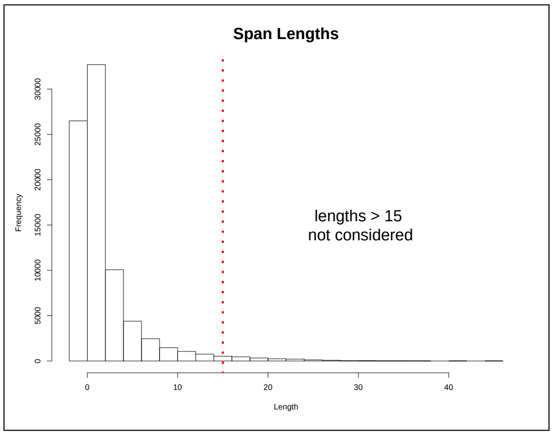

Figure 2: Histogram of answer span lengths in the SQuAD dataset.

3 Experiments

3.1 Data

Our experiments are conducted on the Stanford Question Answering Dataset (SQuAD) v1.1. Passages in SQuAD come from 536 articles from Wikipedia covering a wide range of topics. Each context is a single paragraph from a Wikipedia article, and around 5 questions are associated with each context. In all, there are 23,215 passages and 107,785 questions. The data has been split into a training set (with 87,599 question-answer pairs), a development set (with 10,570 question/answer pairs), and a hidden test set. The first was further split into training and validation sets, where the latter consisted of 5% of the data. The development and hidden test sets were accessed remotely through CodaLab.

3.2 Configuration

We first tokenize all the passages, questions, and answers. The resulting vocabulary contained 117,000 unique words. We use word embeddings in GLoVe with 300-dimensions to initialize the vocabulary. Words not in GLoVe are randomly initialized. This created a total embedding space of 429,170 words, 400,000 of which had proper GLoVe vectors. The word embeddings are not

Table 1: Summary of Coda Lab models' performance

	Development (F1/EM)	**Test (F1/EM)**
Wang and Jiang	72.7/63.0	- / -
Unregularized	45.0/35.0	- / -
Dropout p=0.8	61.6/49.2	63.0/51.0
Dropout p=0.5	61.3/48.6	61.3/49.5

updated during the training of the model. The dimensionality l of the hidden layers is set to be 150. We use ADAM to optimize the model, with a learning rate of 0.001. Each update is computed through a minibatch of 50 instances. We experimented with different dropout rates and placement. In the final model, we utilized a dropout probability of 0.5 and only applied this regularization to the Match-LSTM cell. We train the model for 10 epochs; each epoch takes 45 to 55 minutes.

We measure performance by two metrics: percentage of exact match with the ground truth answers, and the word-level F1 score when comparing the tokens in the predicted answers with the tokens in the ground-truth answers. Note that in the development and test sets, each question has around three ground truth answers. F1 scores with the best matching answers are used to compute the average F1 score.

3.3 Results

The results of our models submitted to CodaLab as well as the most comparable implementation by Wang and Jiang are shown in Table 1. The chosen model from the original Match-LSTM paper has a unidirectional LSTM for pre-processing questions and paragraphs; has a state size $l = 150$; and an intelligent search function similar to ours detailed above. We see that while our models performed well, they were not able to replicate the performance of the models from the paper.

We have several potential explanations for this discrepancy. First, we accidentally overlooked two hidden layers whose equations were not explicit in the original paper. While these served as intermediaries between the attention layer and the decoder, their absence could very well generate some of the observed gap.

Second, the different optimizer adversely impacted our performance. Wang and Jiang utilized the ADAMAX algorithm. This is more suitable for sparsely updated parameters, as its infinite-order norm ignores terms that are close to zero. It is therefore more robust to noise in the gradients. The ADAM optimizer that we use does not do this. The displayed graph of train loss demonstrates that our model lacks this robustness. Though the loss decreases over epochs, it does not converge. Applying a better optimizer or utilizing simulated annealing can help better convergence and create a better model.

Finally, regularization needed more experimentation than we gave it. While we tried dropout values of 0.8 and 0.5, long train cycles prevented us from checking these as thoroughly as we could have. Either using L2 regularization or no regularization at all could have yielded better results than our experiments.

Past research is inconclusive regarding the benefits of dropout on LSTMs. Our results provide no clarity regarding this question, as the model with dropout $p = 0.8$ did just as well as that with dropout $p = 0.5$ on the test set. However, the former model overfit the train set far more than the latter: at the end of training, it had a train F1/EM of 77/55 and val F1/EM of 41/19. The latter had 85/70 and 65/50 in these categories, respectively. The far smaller gap between train and val seemed to suggest better generalization, though this did not occur. Thus, while overfitting is a problem, only part of it (excessive training) is fixed through dropout. Approaches that can maintain performance when applied to LSTMs are certainly the subject of future theoretical and practical work.

3.3.1 Error Analysis

The model generally performs well with proper noun, date, and number answers. These correspond to simpler "what," "who," and "when" questions. The decision to limit answer spans greatly increased performance on these questions. We have found that when dates, proper nouns, etc. are present, they are assigned very high probability. This could be because sets of these simpler question/answer pairs are fairly syntactically similar to each other.

Questions that require more advanced analysis ("why" or "how") questions are not answered as easily. In particular, this system fails to capture certain syntactic subtleties in longer answers, even though it usually selects an answer in the right neighborhood of the ground truth. For example, the system often adds unnecessary words after the end of the true answer until the end of a clause:

```
Q: What do a variety of industries benefit from?
A (predicted): hunting and support hunting on economic grounds
A (true): hunting
```

Penalizing longer answers would correct the above problem, but this system also occasionally removes words from the correct answer. For example, this happens when questions begin with "why"—humans generally answer these with statements starting with "because." However, the model rarely chooses conjunctions as starting words, and the same is true in this instance:

```
Q: Why was the word " national " a cause for alarm to both
Federalists and Anti-Federalists ?
A (predicted): the experience under the british crown
A (true): because of the experience under the british crown
```

The second error type (predicting a shorter answer than the gold standard) is fairly rare, so a mechanism that penalizes longer answers might help performance overall. As for the second error type, extra features such as part of speech in the initial input might help to identify "special" conjunctions, etc. such as "because." We did notice, however, that using larger GloVe vectors alone enriched the model's vocabulary and understanding of these subtleties, and helped performance on these trickier questions.

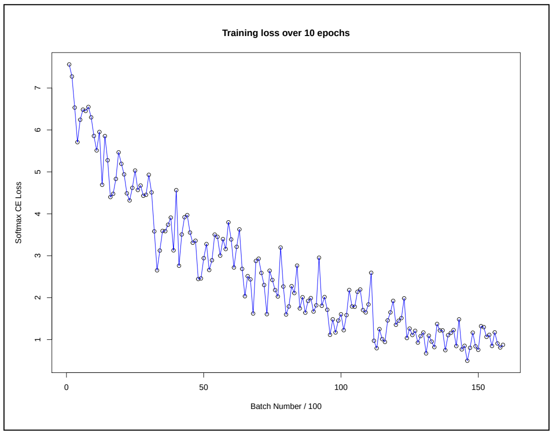

Figure 3: Training loss over 10 epochs. One data point per 100 batches.

4 Conclusions and Future Directions

We implemented a question-answering system originally proposed by Wang and Jiang (2016) [9] and evaluated on the Stanford Question Answering Dataset (SQuAD). After some experimentation with regularization parameter tuning, we achieved an F1 score of 63.0% and exact match score of 51.0% on a hidden test set, indicating the model both performs well and is able to generalize.

Future directions would begin with exploring the hyperparameter optimizations and model infrastructure mentioned above: better regularization, learning rate updates (through simulated annealing or a better optimizer), and implementing the hidden intermediaries in the original paper. In particular, we are interested in comparing the effects of various regularization procedures on LSTMs and understanding how to best avoid overfitting to train data while preserving F1 and EM.

We also have a number of ideas on improving various parts of our model. We would like to implement character-level CNN and bi-LSTM over our paragraph and question in pre-processing. This would provide a richer hidden representation to feed into the Match-LSTM layer. Better preprocessing may, in and of itself, improve performance on tasks by encoding richer syntactic and semantic understanding. To correct for the most common errors noted above, we would like to incorporate layers that better encode syntactic information and penalize longer answers (for example, through a weighted window in the answer search). These and similar improvements could potentially push our model past the performance in the original paper.

Acknowledgments

We'd like to thank the CS 224N teaching staff for the hours of assistance they provided in office hours and Piazza. This project was massive in scope, particularly considering the tight timeframe, and would not have been achievable without their help. We also want to thank the folks running the Sherlock computational cluster at Stanford. While we would have appreciated receiving an Azure GPU, the Sherlock staff was amazingly helpful at helping us properly configure TensorFlow and deal with myriad bugs. Finally, we were amazed that in a project with an explicit leaderboard, so many classmates were willing to discuss concepts and help us debug. We'd like to thank our fellow students for reminding us of the collaborative learning environment that makes Stanford such a unique place.

References

[1] Karl Moritz Hermann et al. "Teaching machines to read and comprehend". In: *Advances in Neural Information Processing Systems*. 2015, pp. 1693–1701.

[2] Felix Hill et al. "The Goldilocks Principle: Reading Children's Books with Explicit Memory Representations". In: *arXiv preprint arXiv:1511.02301* (2015).

[3] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: `http://www.aclweb.org/anthology/D14-1162`.

[4] Pranav Rajpurkar et al. "Squad: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250* (2016).

[5] Matthew Richardson, Christopher JC Burges, and Erin Renshaw. "MCTest: A Challenge Dataset for the Open-Domain Machine Comprehension of Text." In: *EMNLP*. Vol. 3. 2013, p. 4.

[6] Minjoon Seo et al. "Bidirectional Attention Flow for Machine Comprehension". In: *arXiv preprint arXiv:1611.01603* (2016).

[7] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2692–2700.

[8] Shuohang Wang and Jing Jiang. "Learning natural language inference with LSTM". In: *arXiv preprint arXiv:1512.08849* (2015).

[9] Shuohang Wang and Jing Jiang. "Machine comprehension using match-lstm and answer pointer". In: *International Conference on Learning Representations* (2017).

[10] Caiming Xiong, Victor Zhong, and Richard Socher. "Dynamic Coattention Networks For Question Answering". In: *arXiv preprint arXiv:1611.01604* (2016).