An Exploration of Approaches for the Stanford Question Answering Dataset

Charles C. Chen

Abstract

In this paper, we present an exploration of several approaches of varying complexity, novelty and effectiveness applied to solving the reading comprehension problem evaluated by the Stanford Question Answering Dataset (SQuAD), taken from the perspective of a novice practitioner of Deep Neural Networks for Natural Language Processing. Here, we present several models, their mathematical structure, analysis of their implementation and evaluation of their effectiveness in retrieving answers to questions on examples from the SQuAD dataset. In particular, we evaluate (1) a simple sequence attention-mix boundary model, (2) a more complex sequence attention-mix token model and (3) an implementation of a Match-LSTM model for reading comprehension. For each model, we also provide the results of a thorough hyperparameter search and lessons learned from implementation the model in the TensorFlow framework. Our final model achieves an F1 score of 57.7% and an exact match score of 46.3% on the official SQuAD test set.[1] We conclude with notes on the training infrastructure we built to effectively support model training and hyperparameter search, and valuable practical lessons learned from the operation of a physical networked computer cluster for neural network training and evaluation.

1 Introduction

The Stanford Question Answering Dataset (SQuAD), introduced by Rajpurkar et. al. (2016), is a crowdsourced reading comprehension dataset consisting of questions concerning context paragraphs of text and their associated answers as subspans of the associated context paragraph. The SQuAD dataset is a popular metric for evaluating modern reading comprehension models. The dataset offers Natural Language Processing practitioners a large dataset of question-answer pairs that are realistic and diverse in the forms of reasoning needed to arrive at a solution. Previous machine understanding datasets like DeepMind's CNN & Daily Mail news article dataset were in the fill-in-the-blank "cloze" format and were found to be easier to solve than expected (Chen 2016). Others were high-quality but too small for adequate training of deep neural networks.

1.1 Prior work

Since the release of SQuAD, highly competitive approaches to machine question-answering systems have been proposed. All high-scoring approaches on the SQuAD leaderboard involve the usage of recurrent neural networks (RNNs) and LSTMs in particular. As is commonplace in the field, all such approaches use dense vector embedding representation of words as a primary component, and several models have additionally proposed the usage of character-based convolutional neural networks. Attention mechanisms like Match-LSTM and Answer Pointer (Wang & Jiang 2016) are more naturally suited than other methods for the task of selecting ranges from a variable-length sequence of text, and are often used in the decoding stage of highly-competitive systems proposed.

[1]CodaLab ID: ccy

2 Models

2.1 Simple Sequence Attention-Mix Boundary Model

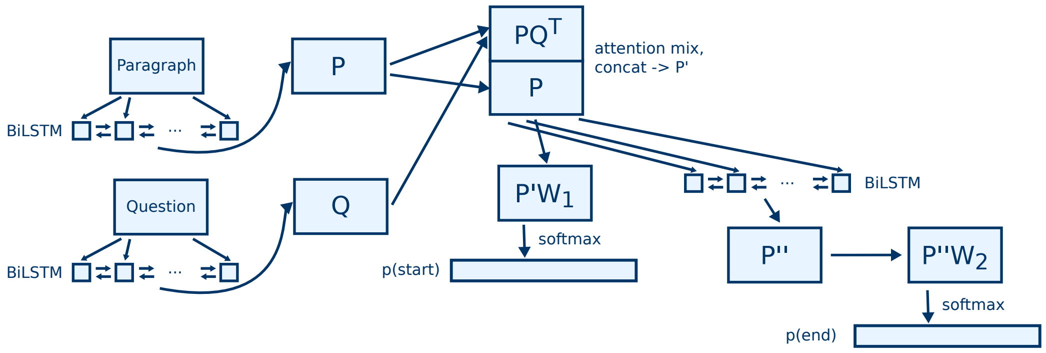

Figure 1: Illustration of the Sequence Attention-Mix Boundary Model.

Here, we first describe a simple sequence attention-mix boundary model as a baseline model we evaluated for reading comprehension. This model incorporates the question context in reading the paragraph text.

2.1.1 Model components

A figure diagramming the components of this model is provided in Figure 1.

Inputs The input to this model is given by two matrices PARAGRAPH and QUESTION of dimensions $d \times N_P$ and $d \times N_Q$, respectively, where d is the size of the word embedding and N_P and N_Q are the number of tokens in the paragraph and question, respectively.

Paragraph and question context representation layer The paragraph and question representations are each passed to a BiLSTM layer in order for the output at each token position to capture information about the surrounding context of each word. The output of the BiLSTM layers are the matrices P and Q.

Attention mix The attention mix layer tries to capture for each paragraph token which parts of the question is most relevant to that token. For each paragraph token, we obtain such a representation as rows of the the matrix PQ^T. We concatenate this with the original paragarph context respresentation P to form the matrix $P' = [PQ^T; P]$.

Start position prediction We pass the output of the attention mix through a single linear transformation $P'W_1$ and take its softmax to produce our prediction for $p(\text{start})$, where each position has our predicted probability of that token being the start of the answer span.

End position prediction We take the output of the attention mix and pass it through a BiLSTM, whose output forms P''. Next, we pass this through a single linear transformation $P''W_2$ and take its softmax to produce our prediction for $p(\text{end})$, where each position has our predicted probability of that token being the end of the answer span.

Inference For a given paragraph and question for which we obtain the output vectors $p(\text{start})$ and $p(\text{end})$ through this model, we infer the answer span as follows. We select the start and end paragraph token indices i and j $(i \leq j)$, respectively, as $\text{argmax}_{i,j;i\leq j}\, p(\text{start})_i p(\text{end})_j$.

2.1.2 Implementation and evaluation

We implemented this model in TensorFlow. We found that the . Our best model trained with an L2 regularization parameter of 0.001 achieved 0.25% EM training accuracy after 10 epochs.

2.2 More Complex Sequence Attention-Mix Token Model

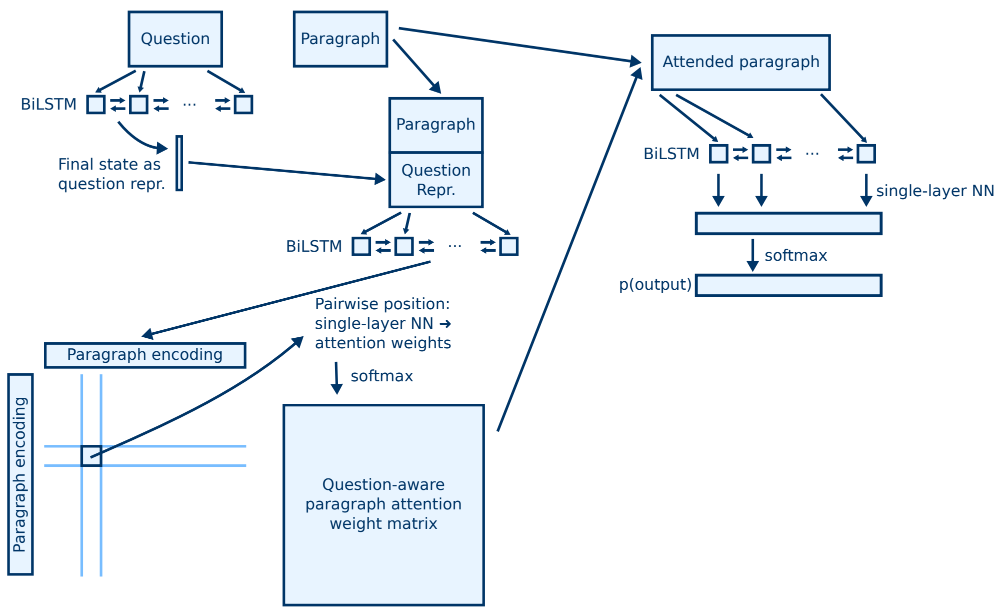

Figure 2: Illustration of the Question-Aware Attention-Mix Token Model.

Here, we describe a more complex sequence attention-mix token model, where we now wish to read the paragraph with a representation of the question in mind.

2.2.1 Model components

A figure diagramming the components of this model is provided in Figure 2.

Inputs The inputs are the two matrices PARAGRAPH and QUESTION as described in the previous model.

Question and paragraph representation We first pass the question through a BiLSTM layer and obtain our question representation as the final state of the BiLSTM. For each token of the paragraph, we concatenate this state vector. These tokens are then passed through a second BiLSTM to produce as outputs the question-aware paragraph encoding for each paragraph token.

Question-aware attended paragraph representation For each pair (i, j) of paragraph encodings, we calculate the pairwise attention weight from the two encodings using a single-layer neural network. These weights are passed through a softmax to obtain, for each paragraph token, the attention mixture weights for each other paragraph token, which we then use in conjunction with the paragraph matrix to produce the attended paragraph representation. paragraphOutput predictionWe pass the attended paragraph thorugh a BiLSTM, whose output vectors are each individually fed into a single-layer neural network to produce a weight for each paragraph token position. The softmax of these weights is taken as the probability $p(\text{output})$ for outputting a given token.

Inference In this model, we have implemented two different inference mechanisms: (1) the discontinuous token inference strategy and (2) the contiguous span inference strategy. In the first strategy, we interpret the output of the softmax $p(\text{output})$ as a probability and in order, we output the i-th token if $p(\text{output}) > 0.5$. In the second continuous span inference strategy, we seek to find the span from i to j (inclusive) that maximizes the probability $p(\text{output})_i p(\text{output})_{i+1} \cdots p(\text{output})_j$. We do this by computing the prefix sum $\sum_{i=0}^{i=k} \log p(\text{output})_i$ for all k and choose i and j as $\text{argmax}_{i,j;i \leq j}\, p(\text{output})_i p(\text{output})_{i+1} \cdots p(\text{output})_j$.

2.3 Implementation and evaluation

We implemented this model in Tensorflow. Of particular note is that the computation of the question-aware paragraph attention weight matrix, when implemented as a vectorized operation, involves computing tensors for the attention weight neural net for every possible pair of tokens in the paragraph. This means that the amount of memory used as an intermediate is $O(\text{length}(\text{longest paragraph})^2 \cdot (\text{net size}))$. In our initial implementation, we found that the GPU consistently ran out of memory during training. We alleviated this problem by reducing the attention weight neural net's hidden layer size and by training only on examples with paragraphs below 400 words in length.

We evaluated a number of different hyperparameters and achieved a final F1 score of 33% with an EM score of 15%.

2.4 Match-LSTM With Answer-Pointer Sequence and Boundary Models

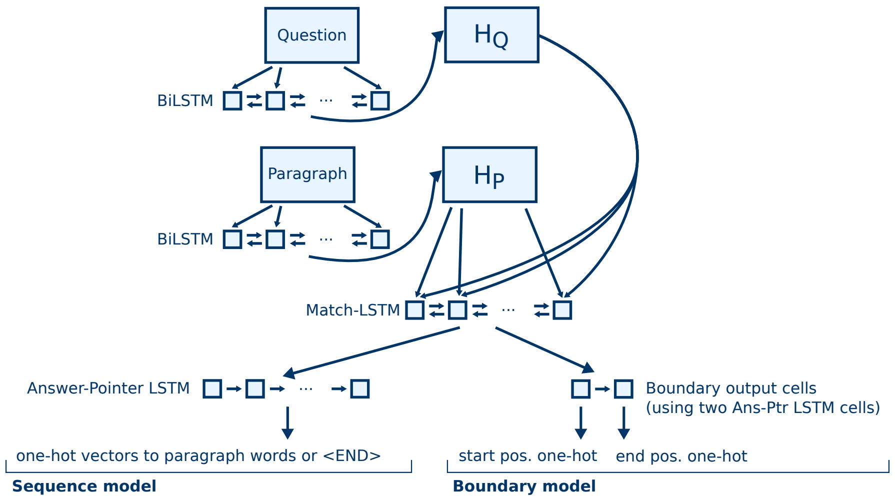

Figure 3: Illustration of the Match-LSTM Model.

Here, we describe our implementation of the Match-LSTM model of Wang & Jiang (2017). In this model, we first compute using two BiLSTMs the question and paragraph representations and pass it to a Match-LSTM. The intuition behind the Match-LSTM is that it concatenates and augments the paragraph representation at each token representation with a weighted sum of the question token representations, weighted by an attention mechanism. A further attention mechanism in the form of the Answer-Pointer LSTM is used to "attend" to the relevant answer spans.

2.4.1 Model components

A figure diagramming the components of this model is provided in Figure 3.

Inputs, question and paragraph representation As in the previous model, the input are the two PARAGRAPH and QUESTION matrices. These are each passed through a BiLSTM layer to obtain the context-aware question and paragraph representations H_Q and H_P.

The Match-LSTM layer For each paragraph token position, the Match-LSTM layer is responsible for augmenting the context-aware representation of that token from H_P with an attention-weighted sum of context-aware question token representations, before passing this augmented representation into a standard LSTM cell. For each paragraph position i, where $h_P^{(i)}$ is the paragraph representation at that point and $h_R^{(i-1)}$ is the previous LSTM hidden state, we compute the weight of each question representation position as α_i, where

$$G_i = \tanh\left(H_Q W_Q + \text{expand\_dims}(h_P^{(i)} W_P + W_R h_R^{(i-1)} + b_P)\right),$$

$$\alpha_i = \text{softmax}(G_i w + \text{expand\_dims}(b)),$$

where we let W_Q, W_P, W_R, b_P, w and b be learnable parameters, and we let expand_dims be the broadcast operation for tensor addition.

The Answer-Pointer layer The Answer-Pointer layer is responsible for attending to the particular paragraph positions that should be output by the model. This model has two variations: (1) the sequence model and (2) the boundary model.

We first describe the sequence model. Here, we model the answer span as attention to a sequence of (consecutive) tokens in the paragraph followed by a special "stop" token. We augment the output of the Match-LSTM layer (let it be H_R) with an extra $N_P + 1$st column $\tilde{H}_R = [H_R, 0]$. We then model the attention of the the kth step/cell of the Answer-Pointer LSTM layer as β_k, where

$$F_k = \tanh\left(\tilde{H}_R V + \text{expand\_dims}(h_A^{(k-1)} W + b_A)\right),$$

$$\beta_k = \text{softmax}(F_k v + \text{expand\_dims}(c)),$$

where V, W, b_A, v and c are learnable parameters. Here, $\beta_k \tilde{H}_R^T$ is passed as the next input to the LSTM. During inference, the answer span stops at a time step k when the attention vector β_k points to the $N_P + 1$st entry.

In the boundary model, we do not augment H_R with an extra column and use the same Answer-Pointer LSTM cell described above and model the attention of exactly two cells.

Inference During inference for the sequence model, we follow the attention vectors β_k from $k = 0, 1, \cdots$, taking the word to be output as the argmax index of that attention vector, until we reach a vector where the argmax index points to the $N_P + 1$st entry, indicating the end of the sequence.

During inference for the boundary model, we take the start i and end j positions to take $\text{argmax}_{i,j;i \leq j}\, p(\text{start} = i)p(\text{end} = j)$.

2.4.2 Implementation and evaluation

We implemented the model in TensorFlow, subclassing `BasicLSTMCell` as necessary to implement the Match-LSTM and Answer-Pointer cells. We made sure to carefully mask the variable-length sequences in batches, where applicable, so as to get correct softmax values.

We needed to perform an extensive hyperparameter search to achieve good performance with the boundary model (we focused on this because the sequence model too much longer to train and stagnated at 33% validation F1 accuracy). Initial selection of parameters, as suggested in Wang & Jiang (2017)'s paper of a hidden layer size and a LSTM hidden state size of 150 resulted in poor performance, overfitting on the training dataset despite a modest dropout regularization parameter of 0.1, with validation F1 stagnating at 37%. We achieved better performance by doubling the size of the hidden layer and state to 300, and achieved better performance still when the dropout regularization parameter was increased to 0.5. Since we still saw overfitting on the training set, with training F1 values around 90% while validation F1 hovered around 50%, we decided to perform a hyperparameter sweep of the dropout parameter, doing so in increments of 0.1 from 0.2 to 0.8. When we did so, we saw that when trained from scratch, models with higher dropout parameters (≥ 0.7) stagnated during training and did not achieve high performance. Seeing this, we tried to train models with higher dropout (0.6, 0.8) with a reasonable-performance model trained with dropout of 0.5 as the initial warm-start checkpoint. Here, we saw that performance was better than the models trained from scratch, but stagnated at a validation F1 of 50% and a training F1 around 70%. We experimented with using the Adamax instead of our default optimizer and found a slight increase in validation accuracy. We fine-tuned our final model by manually performing learning rate decay when we observed a plateau in loss values. Our final model was selected by taking checkpoints of models which achieved the highest validation set accuracy when periodically evaluated during training. Even with decayed learning rates, we saw that the performance of the model on the validation set was highly dependent on the particular checkpoint at which a snapshot was taken.

Our final Match-LSTM model achieved a F1 score of 57.5% and an exact match score of 46.3% on the official SQuAD dataset. To achieve this result, we performed a total of 75 separate training sessions on varying hyperparameters, each training for 5 to 10 epochs on the SQuAD training set, for several GPU-weeks of training time. Despite our efforts, we could not reproduce the results presented by Wang & Jiang's original Match-LSTM paper of a F1 score of 73.7% and an exact match score of 64.7% on the official SQuAD test set. Further investigation and experimentation is in order to account for this gap.

3 Infrastructure

3.1 Model parameter framework

We rewrote the entire training and evaluation model for clarity and to better support our large hyperparameter search experiments. We found it very helpful to be able to tune all options of the experiment, from selection of which model to construct and the parameters for training or evaluation, and to be able to save and load these parameters during experiments. During every training session, we wrote the exact hyperparameters used for a specific experiment along with the checkpoints generated for that session, and we constructed a machine-readable logging format to log relevant events during training, including loss per epoch, periodic evaluation results on the dev and test sets and gradient norms.

During development, we made sure to preserve the invariant that all tuning parameters and options were backwards-compatible, so that the TensorFlow graph could be correctly constructed for all previous parameter values. This makes it easy to look back retrospectively on old parameter values and evaluate comparative performance.

One particularly time-saving and exploritorily useful feature we implemented was the warm-starting the training of models with changed hyperparameters. This allowed several simultaneous experiments on hyperparameters starting with the parameter values of a single trained model, each run on a different machine in our cluster.

3.2 Computation Environment

Substantially all of the model training and evaluation computations were performed on a networked cluster of 8 computers with modern four-core Intel Xeon E3-1225 v3 CPUs, each with an attached NVIDIA GTX 1070 graphics card. Each machine was imaged with Ubuntu 14.04 and included NVIDIA's CUDA and cuDNN drivers and packages. We also evaluated the performance of model training with TensorFlow on alternative environments, such as that of a machine with a modern 14-core Intel Xeon E5-2683 v3 server CPU with an attached NVIDIA GTX 1070 graphics card, and on Google Cloud's GPU instances, but we suffered degraded performance, as outlined below.

3.3 Evaluation of CPU single-core performance

Surprisingly, for all models we have trained, training performance was much higher (as much as 2-3x lower in time required per batch) when done on consumer-grade 4-core Intel chips (like the Xeon E3-1225 v3, comparable to the Core i5 consumer chip), than when done on professional-grade 14-core Intel chips (like the Xeon E5-2683 v3 CPU), which have more cores but slower single-thread perforamnce, when the same model of graphics card (NVIDIA GTX 1070) was used on each machine. This was likely to be due to time spent in single-thread-bound bottlenecks when feeding data to be processed by the GPU. Amusingly, on the 14-core CPU, single-core speed actually differs between cores, where Core 0 runs at a faster speed than the other cores, so training performance depends on the arbitrary CPU core assignment by the kernel for a Tensorflow training process.

This provides an interesting dilemma for practitioners looking to set up hardware for a local computation environment. While practitioners may be drawn to higher-end chips which tend to have higher core-count but lower single-thread performance, they could in fact improve performance by purchasing cheaper consumer-grade parts. However, consumer-grade CPUs have a limited number of PCI-E transfer lanes for communication between the CPU and GPU, so practitioners building multi-GPU machines may be forced to choose between being bottlenecked by single-thread CPU core performance or being bottlenecked by PCI-E bandwidth.

3.4 Evaluation of cloud provider training performance

Modern datacenters use high-core-density chips like the ones described above to maximize compute power per physical machine. This has the same downside as described above where machines have more cores, but each core is slower, for lower single-threaded performance. Empirically, when running the same model on a GPU-enabled Google Compute Engine VM instance, compared with our local cluster setup, training time per epoch was 3 times slower, even though the machine used the higher-end (and 10 times more expensive) NVIDIA Tesla K80 rather than our NVIDIA GTX 1070 cards.

4 Future work

Future work can be done to improve the performance of the models described. One possibility is to add a character-level embedding for each word using max-pooled convolutional neural nets on the letters comprising each word. We were not able to obtain the published perforamnce of Wang & Jiang (2017) despite a very extensive hyperparameter search. While our final Match-LSTM models moderately overfit on the training data, experiments with increasing the amount of dropout used to combat this resulted in worse performance on the validation set; this suggests future work into the further searches for regularization parameters are needed to better understand this aspect of the model.

On the infrastructure front, an effort can be made to support distributed training in TensorFlow, a step that would allow even faster iteration and turnaround in experimentation.

5 Conclusion

In this paper, we evaluated a number of different models to solve the question-answering problem posed by the Stanford Question Answering Dataset. We built up infrastructure, both physical and conceptual for iterating quickly and performing many experiments on our models. We conducted extensive hyperparameter search and found parameters that gave reasonable performance for the Match-LSTM model.

References

[1] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang (2016) SQuAD: 100,000+ Questions for Machine Comprehension of Text.

[2] Danqi Chen, Jason Bolton, Christopher D. Manning (2016) A Thorough Examination of the CNN/Daily Mail Reading Comprehension Task.

[3] Hermann, K. M., Kocisky, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., Blunsom, P. (2015) Teaching machines to read and comprehend.

[4] Shuohang Wang, Jing Jiang (2016) Machine Comprehension Using Match-LSTM and Answer Pointer