Computational Reading Comprehension through Self-Attention and Convolution

Neil Movva
Department of Electrical Engineering
Stanford University
Stanford, CA 94305
nmovva@stanford.edu

Samir Menon
Department of Computer Science
Stanford University
Stanford, CA 94305
samir2@stanford.edu

Abstract

We present a performant solution to the SQuAD question-answering challenge. Though many current question-answering systems opt for recurrent neural network-based architectures, we follow the recent work of [1] to demonstrate an alternative architecture built on convolution and self-attention. After architectural experimentation and hyperparameter tuning, we achieve final scores of 62.7% F1 and 47.8% EM on the SQuAD development set.

1 Introduction

We aim to achieve competitive performance on the Stanford Question Answering Dataset (SQuAD) [2], which presents natural language questions, along with context within which answers are to be found (i.e. the answer phrase is always a contiguous subset of the given context, in the original order). We quantify performance using the standard F1 (harmonic mean of precision and recall) and EM (exact match) scores, evaluated at the token level.

The SQuAD dataset itself includes over 100,000 sets of (question, context, answer) examples, comprised of original text from Wikipedia subsequently labeled by humans. SQuAD's features make it well suited for NLP research, as it exhibits good diversity, high-quality labels, and most importantly offers a large number of training examples.

We propose a solution to SQuAD that balances both accuracy and computational efficiency, achieved through the coupling of self-attention mechanisms and convolutional feature processing. We borrow intuition from both the natural language processing and computer vision domains to motivate our experiments

2 Related Work

Previous approaches to machine question-answering have tended to rely on recurrent neural network (RNN) models, combined with some form of attention mechanism. Bidirectional Attention Flow (BiDAF) [3] is one particularly popular, recent example, and proposes a general approach of computing attentions both from the query to context and from the context to the query. Long short-term memory (LSTM) cells are used to improve feature embeddings before computing attention, and also to distill the query-aware feature vectors produced by the attention stage into final output. In both cases, the use of recurrent neural models is thought to capture more contextual information within each token; intuitively, we want words to be considered in relation to their usage context.

However, recurrent models are somewhat less computationally efficient than another type of neural architecture that has seen outstanding success in the field of computer vision: convolutional neural networks (CNNs). Broadly speaking, convolution operations can be more amenable to fast compu-

tation thanks to a hierarchically local memory access pattern and high parallelization potential (no recurrence relation) [4].

Additionally, we have seen CNN models succeed in capturing spatial context, within the domain of image perception. As we have seen recurrent models primarily used to capture global context for each token (i.e. relation to other words in given sequence), we intuitively consider the use of convolutional models to perform a similar purpose. CNNs achieve contextual awareness through the width of chosen convolutional filters, meaning each element maps to a definite receptive field. Generally, image processing tasks are mostly concerned with local context, so the receptive field of any given element tends to cover a small spatial area, with layer stacking used to gradually blend information across the entire image. This tendency is critical to improving memory locality, and thus computational performance. We posit that this is not a significant limitation for natural language tasks, since tokens tend to interact within relatively short sequences (e.g. descriptive modifiers, negation, multi-word phrases). Furthermore, feature vectors in the natural language domain represent characters or words as individual elements, suggesting higher spatial information density than the large but dimensionally shallow nature of RGB images.

3 Approach

3.1 Formal Problem Definition

The task presented by squad can be formally expressed as:
Given a pair (Q, C) of query Q and context C, find indices A_s and A_e within C such that the sequence of words from $C[A_s]$ to $C[A_e]$ represents the best answer to Q.

3.2 Model Architecture

At a high level, our model maintains some of the popular architectural choices in machine comprehension. In particular, we adopt many of the architectural structures proposed by [1], reimplementing high-level ideas in the TensorFlow framework [5] (see code for inline attributions). We can segment our architecture into 5 logically distinct 'layers' - sequentially, we have an embedding layer, an embedding encoder layer, an attention layer (bidirectional; context-to-query and query-to-context), a model encoder layer, and a final output layer. We discuss each of these layers in more detail below.

3.2.1 Input Embedding

We employ standard methods to obtain word embeddings for each token of the context and query, separately. Word embeddings are found from pre-trained GloVe vectors, with dimensionality of 300 [6]. Out-of-vocabulary words are simply mapped to a dedicated 'unknown' token, which has a default, fixed embedding.

3.2.2 Encoding Word Embeddings

Our encoding system is comprised of convolutional encoder blocks that are repeatedly stacked to process our word embeddings. We follow the recent work of [8] and employ depthwise separable convolutions, which have been widely shown to reduce computational requirements while improving generalization. We experiment with various architectural hyperparameters in our embedding encoder layer, varying the number of stacked blocks, convolution kernel field widths, and number of convolution kernels per block. These variations are discussed further in the following experiments section.

After the convolutions, we also apply self-attention, via multi-head attention as in [1]. This consists of feeding the output of the convolutions into standard multihead attention as the queries, keys and values. While attention-based models typically have the number of heads as an important hyperparameter, we found no significant change in performance of our model when the number of heads was set to more than 1, so we have set the number of heads to 1.

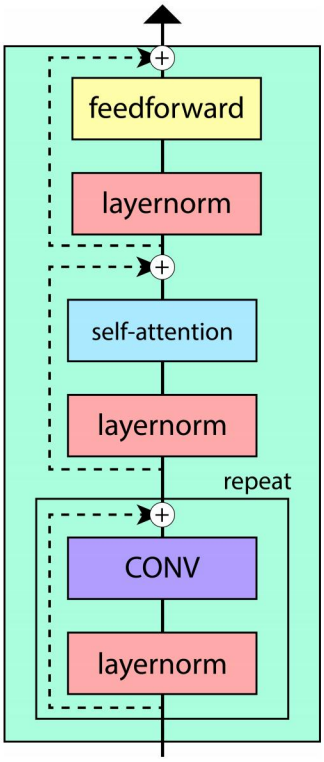

Figure 1: Encoder convolutional block. We apply layer normalization before each operation, as suggested in [7]. CONV layers are stacked to some variable depth (nominally 4, varied in experiments), before being sent to a self-attention mechanism. Every operation has a skip connection circumventing it, depicted here as dashed lines.

3.2.3 Bidirectional Attention

As seen in many other machine comprehension models, we employ attention mechanisms for both context-to-query and query-to-context. For context-to-query, we take every possible pair of context and query tokens and compute their similarity using some similarity function. As Seo et al. found [3], a simple trilinear mapping (that is, $[c, q, c \odot q]$) with a trainable weight matrix does well, so we use that.

The context-to-query attention tries to determine which query words each word in the context is most related to. First, for each context word, we normalize its similarities to each query word using an application of softmax. Then, we multiply the per-context-word normalized similarity matrix to the queries, resulting in a query attention vector for every context word.

The query-to-context attention is trickier. Intuitively, we are now finding the context words with the highest similarity to each query word. We use the per-context-word normalized similarity matrix again, but then multiply it by a per-query-normalized similarity matrix. Finally, we apply this similarity matrix to the queries, resulting in an query attention vector for every context word.

3.2.4 Model Encoding

The model encoding layers are where the bulk of this model's parameters are, and where, intuitively, most of the learning is happening. This is where the model can use the attention and the original input encodings to build an understanding of what the question is asking, what terms in the question are most relevant to the answer, and where in the context an answer might be found.

We stack 7 encoder blocks in each of the 3 model encoder layers: M0, M1 and M3. These encoder blocks have the same structure as the input encoder, but have fewer convolutions (2) and a smaller kernel size (5). All 3 model encoders share weights, taming the number of parameters and forcing the model to squeeze more information into its limited hidden state. We use M0 and M1 to produce the start probabilities, and M0 and M2 to produce the end probabilities. The idea here is to allow the network to share the information relevant to both the start and end probabilities in M0, and then have M1 and M2 contain the information particularly relevant to the start and end, respectively.

3.2.5 Final Output

In the output layers, the start and end logits are formed via a linear layer on top of the model encoder's outputs. These logits are fed to standard softmax with cross-entropy loss, and the start loss and end loss are summed.

4 Experiments

4.1 Experimental Setup

We performed all experiments on a modern Nvidia Pascal GPU (GP102) with 11GB of onboard memory; from this standard architecture, we expect our performance observations to generalize well to contemporary machines. For software, we use Tensorflow 1.5 compiled against Nvidia's cuDNN 7 library.

We use standard L2 regularization, dropout between all layers and sublayers, and layer dropout as in [7], making deeper layers progressively more likely to be dropped. We also used, as suggested in [1], the ADAM optimizer ([9]) with $B_1 = 0.8$, $B_2 = 0.999$, and $\epsilon = 10^{-7}$.

4.2 Computational Performance

Major motivation for this work comes from the observation that, on modern hardware, convolutions tend to be more efficient than recurrent operations. We find this to be generally true in our experimental setup, observing the following throughput statistics during training:

	Training (examples/sec)	Inference (examples/sec)
Our model	81.2	220.4
CS224n Baseline	55.7	108.1

Table 1: Performance of our convolution based model, vs. the CS224n baseline SQuAD implementation.

4.3 Model Accuracy

We evaluate our model on the SQuAD dataset, as provided and segmented by the CS224n course administrators. We train on the default training set without augmentation, and evaluate accuracy on the default dev set. At the time of writing, we regrettably do not have a result on the official CS224n test set, pending recent issues with the CodaLab platform.

	F1 Score (Dev)	EM Score (Dev)
Our model	62.7	47.8
CS224n Baseline	43.9	34.7

Table 2: Comparison of model accuracy, after 10 epochs of training.

4.4 Error Analysis

Across several example runs of our model, we notice two common failure modes: .

Firstly, it was repeatedly marking the end as occurring before the start. This is because the prediction procedure was to simply take the maximally probable start and end positions independently and simply return them. The model was unable to learn in an end-to-end fashion that marking the end before the start would always be incorrect. We suspect this is due to how loss was calculated. Unfortunately, adjusting the loss to account for the strict ordering is very tricky - it essentially requires some kind of search algorithm (or even possibly dynamic programming) within the network itself, since we'd need to take the probability distributions outputted and search for the highest start and end pair that had the start prior to the end. Making this compatible with backpropagation is difficult, since the search problem seems to have no simple gradient.

Instead, we can use search probability distributions at prediction time, looking for the maximal product of a pair of start and end probabilities satisfying some constraints. In our case, we chose two constraints: first, that the start occur before the end, and second, that the answer be no longer than 30 tokens. Using a relatively simple search algorithm, we can then find the best pair, and return these as our prediction.

4.5 Length Penalization

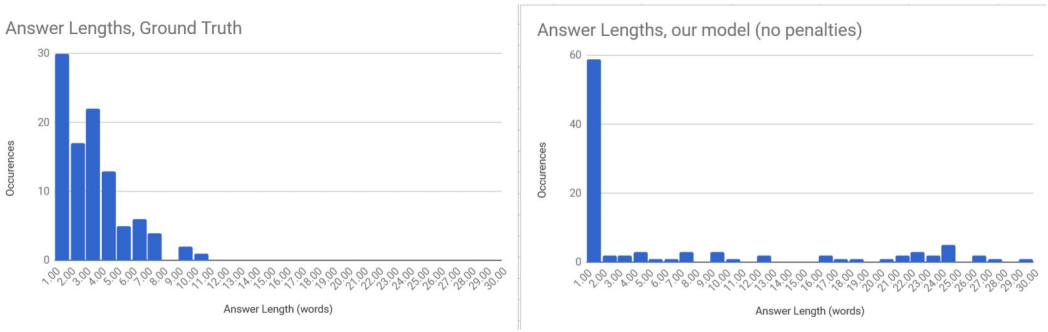

Figure 2: Histogram of true answer lengths, vs. our model's predicted answer lengths. Evaluated on a (consistent) 100-sample 'experiment' set.

Our initial model exhibited a strong preference to one-word answers, with a long tail of extremely lengthy answers (up to 30 words). We are especially concerned by the latter case, since longer answers erode our F1 score while also calling into question the learning performance of our model - intuitively, it is far easier to simply regurgitate context than to achieve semantic understanding of the input. Furthermore, we notice that many of the ground truth human examples choose short answer lengths, suggesting that some heuristic preferring shorter answers could bring our model's output more in line with human judge opinions.

Thus, we apply a simple penalty to answer length when selecting a best answer. It is important to note that this selection preference is applied downstream of the loss function; the model is still being trained to output the most likely probability distributions for start and end indices. Instead of simply selecting the most likely start and end, however, we compute a heuristic fitness score that may choose a less likely pair on the basis of having a shorter, and thus probably more human-friendly, answer. Fitness is computed exhaustively across all possible word pairs - while the computational complexity is unfortunately quadratic in the length of the context, we find the constant terms to be small enough to execute in negligible time on modern hardware.

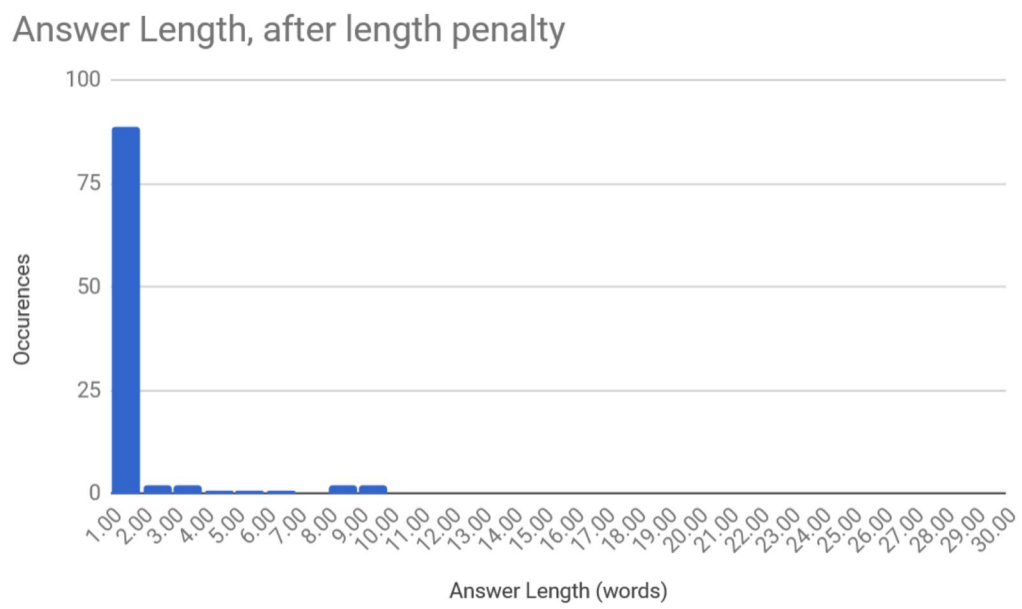

Figure 3: Histogram predicted answer lengths, after length penalization. Evaluated on a (consistent) 100-sample 'experiment' set.

We find our length penalization scheme to be extremely effective at reducing the occurrence of long answers. However, we now notice significantly more one-word answers, instead of the smoother answer length falloff exhibited by human judges.

4.6 Ablation Test: No Convolution

We wanted to test the extent to which convolutions were essential to this model's success. We ran an experiment in which all convolutions in each of the encoder blocks were disabled. Without these, the model's ability to learn was significantly impaired - without convolutions, the model could only achieve an F1 score of 32.8 and an EM score of 19.1. Clearly, convolutions were essential to this model's success. This is notably different than the result of [1], where removing all convolutions only reduced the F1 score by about 3 points. We suspect that our model relies much more heavily on convolutions, perhaps because we lack the character embeddings that they used and trained for shorter amounts of time.

4.7 Convolution Receptive Field Sensitivity

To further test the extent to which convolutions were being used to aggregate data about neighboring words, we ran an experiment with reduced kernel sizes: we changed the input encoder's kernel size from 7 to 3. This model was slightly faster to train, and achieved an F1 score of 55.5 and an EM score of 39.8. This indicates to us that words have perhaps surprisingly local effects; that is, they mostly changed the meaning of words within a fairly narrow window around them. However, we find the small (¡ 10%) performance gain insufficiently favorable for the significant accuracy loss incurred.

5 Discussion and Future Work

We find it very interesting that convolutions can effectively replace recurrent modules in machine comprehension models. Using experiments to test our hypotheses about how machine learning models are actually working has been rewarding. It's led to some intriguing new ideas about language: for example, our work suggests that words can be approximated as acting in a nearly local manner,

and that the vast majority of information contained in a sentence can be recovered without using a recurrent architecture.

5.1 Performance Studies

One broadly appealing feature of convolutions is their ease of parallelization, especially as machine architectures become increasingly wide and distributed. Since we test on a single node, single-GPU research system, we suspect the performance speedup we observe is actually an underestimate of speedup on larger, production-scale systems. In future work, we'd like to better characterize the performance advantages of convolutional models, and explore similar tradeoffs between speed and accuracy.

5.2 Brevity Penalization

While our length penalization experiments proved effective in reducing answer length, we now find that our model is often too brief with its answers, despite drawing from roughly the correct contextual area. Again, we impose a 'human preference' prior to declare such answers unpalatably brief, and would consider penalizing single-word answers in future experiments.

References

[1] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V. Le. Fast and accurate reading comprehension by combining self-attention and convolution.

[2] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text.

[3] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension.

[4] Erich Elsen. Optimizing RNN performance.

[5] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems.

[6] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization.

[8] Franois Chollet. Xception: Deep learning with depthwise separable convolutions.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization.