Faster Attention for Question Answering

Stanford CS224N Default Project (no extenal mentor or collaborators and not sharing project)

Jack Beasley
Symbolic Systems Program
Stanford University
jbeasley@stanford.edu

Abstract

In this project (a default final project on the IID track), I built a question-answering system for SQuAD 2.0 by exploring both the BiDAF model [1] through modifications of the default baseline as well as a from scratch implementation of QANet [2], a self-attention [3] based question-answering architecture. The BiDAF modifications which added character embeddings achieved a small, but significant improvement over the baseline model on the test set with **F1: 64.083, EM: 60.423**. However, the QANet models only nearly matched the baseline BiDAF scoring with character embeddings. Curiously, not only did my QANet under-perform the baseline in model performance, it also turned out to be significantly slower to train and at inference time on GPUs. Though profiling, I found that the QANet model is indeed faster on CPUs, however significantly under-performs the baseline BiDAF model on GPUs because the BiDAF model's slowest component, the RNN, is implemented as a highly optimized CuDNN routine on GPUs that the custom QANet encoder block did not benefit from. Finally, this profiling also shows that faster attention mechanisms, as explored in the literature [4] [5], are unlikely to improve performance on this particular SQuAD 2.0 workload as additional instruction overhead would likely wash out any performance gains absent better operation compilation for GPUs or a custom GPU kernel.

1 Introduction

While in recent years, pre-trained embeddings using architectures like BERT [6] have lead to impressive advances many text-based tasks, including question-answering, pre-trained models tend to be very large, requiring long pre-training times and powerful hardware. These large transformer-based models perform well, but are difficult to modify and experiment with due to their resource requirements, prompting a variety of efforts to reduce algorithmic complexity [7].

While these techniques have been aimed at reducing pre-training resource usage for large transformer-based models like BERT, many of these techniques focus on the self-attention mechanism of the transformer architecture, so should in-principle apply to any transformer-based model. Thus, for this default final project, I aim to asses how well these new efficiency improvements to transformers might apply to QANet [2], a state-of-the-art Transformer-based question-answering model from the era before BERT and pre-trained embeddings. If these algorithmic performance improvements improve the train and inference time of QANet, those speed improvements could lead to model improvements by making bigger models and longer training runs feasible.

Thus, this project first begins with some modifications to the baseline BiDAF model, namely adding character embeddings to create a stronger baseline model which outperformed the original baseline. Next, I built a from-scratch implementation of QANet, evaluate each model for both prediction performance and runtime performance.

2 Related Work

Before self-attention and Transformer-based architectures were developed [3], RNN-based models had proved to be quite successful on question answering tasks. In particular, the BiDAF model given as a baseline [1] was a successful model that achieved state-of-the-art performance on SQuAD (pre SQuAD 2.0). BiDAF is a six-layer model which consists of the following layers:

1. **Character embedding:** A embedding of words using a character-level CNN [8]
2. **Word embedding:** Pre-trained GLoVe word embeddings [9] combined with character embeddings and passed through a two-layer Highway network
3. **Context Embedding Layer:** LSTMs are applied both to the context and the question to get the matrix X from the context words and the matrix U from the query words
4. **Attention Flow Layer:** Attends to X and U from the context and query respectively
5. **Modeling layer** Attends to X and U from the context and query respectively

Building from BiDAF by incorporating the ideas from Transformer architectures, then used in machine translation [3], QANet replaces the LSTM-based context embedding and modeling layers with stacks of "Encoder blocks" which consists of a closed-form positional encoding, a convolutional layer, a self-attention layer, then finally a feed-forward layer as seen in figure 1. These blocks are then stacked to take the place of the LSTM embedding and modeling layers.

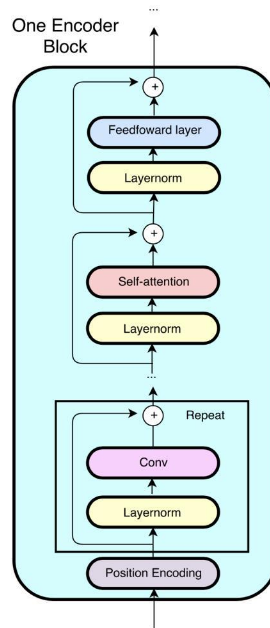

Figure 1: QANet encoder block

This architecture eliminates slow LSTMs and thus runs much faster. The QANet authors documented a 2.9-13.3X speedup of train iteration time and a 3.7-8.8X speedup for inference time over bidirectional LSTMs like BiDAF. Because of this speedup, the authors were able to train for much longer with more data using data augmentation techniques. This train-time speed increase combined with the models performance resulted in QANet reaching BiDAF's best F1 score of 77.0 with a fifth the training time.

However, there have been further improvements on self-attention performance that could improve further. To understand this, consider the multi-headed self-attention layer [3] that is included in the QANet architecture. Multi-headed self-attention is built from scaled dot-production attention as defined in equation 1. Let there be matrices $Q, K, V \in \mathbb{R}^{n \times d}$ where n is the input sequence length and d is the embedding dimension.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

This attention is then weighted by a weight matrix W_i to form the attention head i as defined in equation 2.

$$\text{Head}(Q, K, V, i) = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2}$$

Finally, the output of each of the h heads is concatenated to form the multi-headed attention output in equation 3.

$$\text{MultiHead}(Q, K, V, h) = \text{Head}(Q, K, V, 1) \oplus \ldots \oplus \text{Head}(Q, K, V, h) \tag{3}$$

The key performance issue that Linformer and Reformer seek to alleviate is the scaled-dot product attention operation as the multiplication term QK^T multiplies two $n \times d$ matrices, which is an $O(n^2)$ operation with respect to sequence length in both time and space.

To address this, both the Reformer paper and the Linformer paper find ways of avoiding the calculation of the full matrix $S = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$. Because the softmax output is definitionally dominated by a few terms where Q_i and K_i are similar, the Reformer used Locality-Sensitive-Hashing (LSH) to efficiently find which vectors are similar, then compute an approximation of S using only those. This approach performs well and reduces the computation of self-attention to $O(n \log n)$, however, with a fairly large constant which means it is only faster when n is fairly large [4].

The Linformer paper takes a different approach to this same problem by noticing that S is a low-rank matrix and formally proving this to be so. Intuitively this makes sense because the softmax output is dominated by only a few similar parts of Q and K. Instead of using LSH to exploit this fact, the Linformer projects K and Q into a k dimensional space, where $k \ll n$ but k is large enough such that the first k singular values are cumulatively much larger than the remaining singular values. Because k is a constant unrelated to length, the matrix multiplication becomes an $O(n)$ linear time and space operation while maintaining good performance [5].

While these methods do result in very significant reductions in algorithmic complexity and faster performance on pre-training tasks, it remains to be seen if they will result in better performance on Transformer models with multi-headed self-attention, but no pre-training.

3 Approach

My approach can be split into three portions: implementing character embeddings for BiDAF, implementing and tweaking QANet and profiling the runtime performance of both models.

3.1 BiDAF

The starter code included a BiDAF model (which I refer to simply as "BiDAF"), which achieved an F1 of 60.91 and EM of 57.64 using only pre-trained GloVE word vector inputs.

To extend this model, I added CNN-based character-level embeddings. These embeddings start from character vectors, then are passed through a convolutional layer with hidden size 32 and kernel size 1, then max-pooled. These character-level embeddings are then concatenated with the GloVE word embeddings and sent through a two-layer highway encoder. The CNN character embedding work follows from [8] and the overall architecture follows directly from the original BiDAF paper [1]. I refer to this model as "CharBiDAF".

3.2 QANet

After extending BiDAF to use character embeddings, I implemented QANet as a modification of the original BiDAF models as QANet shares the same embeddings, query-to-context and context-to-query attention and output layers.

The largest implementation piece of this was implementing the encoder block abstraction (figure 1), which is used throughout the QANet model. My implementation begins with a positional encoding layer of alternating sine and cosine patterns as defined in the original transformer paper [3]. I could have used learned positional encodings, however, some recent work seemed to indicate that this sine and cosine encoding scheme performs similarly or better to learned encodings [10]. The positional encoding is a simple periodic function of position and dimension as seen in equation 4:

$$PE(p,i) = \begin{cases} \sin(p/10000^{\frac{2i}{d_{model}}}) & \text{if } i \text{ is even} \\ \cos(p/10000^{\frac{2i}{d_{model}}}) & \text{if } i \text{ is odd} \end{cases} \quad (4)$$

After this position encoding, the embedding is passed through a layer norm, followed by four depth-wise separable convolutional layers with kernel size 7 and residual connections as in the QANet paper. After the convolutional layers, the embedding is sent through a layer norm and then to a multi-headed self-attention layer with 8 attention heads and residual connections. Finally, the attention output is sent through a layer norm and then two feed-forward layers with residual connections and output.

A single encoder block then replaced the LSTM context and query encoders from BiDAF with shared weights for both context and query embeddings.

A stack of two encoder blocks replaced the LSTM modeling layer in the "QANet" and "CharQANet" and a stack of four encoder blocks formed the model layer for "CharTallQANet".

3.3 Measuring Performance

To measure performance, I used the PyTorch profiler which can record function runtime on both CPUs and GPUs. This profiler records function names, start times and end times for all executed functions during train time and allows for user-defined annotations. Because this trace data is detailed, it can get quite large and is thus impractical to record an entire training run, thus, I instead recorded only the first four batches of a training run for each model, which resulted in a profile size of $\tilde{1}$ GB while averaging function runtime over the four batches. While four is a fairly small number of trials to average over, in practice, the time per batch varies very little during training so these results should easily generalized to the whole training process.

4 Experiments

4.1 Data

I used SQuAD 2.0 as defined for the default final project rules for training. All model results are from models trained on the default train dataset and evaluated on the default validation and test datasets.

4.2 Evaluation method

In terms of model evaluation, I track EM and F1 score on the SQuAD question answering task as defined by the rules of the default final project.

I evaluated train speed using the average time for the total train, forward portion and backward portion averaged over four batches in the training set, using the PyTorch profiler as described in the approach section.

4.3 Model Results

The CharBiDAF improvements resulted in an F1 of 63.99 and an EM of 60.36 on the validation dataset and an even-higher F1 of 64.08 and EM of 60.42 on the test dataset, which turned out to be my best-performing model.

The "QANet" and "CharQANet" models both slightly underperformed the baseline "BiDAF" on the validation set, getting F1 of 60.68, EM of 57.0 and F1 of 60.38, EM of 56.76 respectively. The "CharTallQANet" with two extra model layers slightly outperformed the baseline with an F1 of 61.29 and EM of 57.54. Full validation results can be found in table 4.3. However, all QANet models significantly underperformed on the test dataset as seen in 4.3.

Model Name	F1	EM
BiDAF	60.271	56.720
CharBiDAF	64.083	60.423
CharQANet	58.114	54.117
CharTallQANet	58.476	54.658

Table 1: Model test results on the test dataset

Model Name	F1	EM	NLL	AvNA
BiDAF	60.91	57.64	3.01	67.62
CharBiDAF	63.99	60.36	2.96	70.51
QANet	60.68	57.02	3.0	68.63
CharQANet	60.38	56.76	3.08	68.24
CharTallQANet	61.29	57.54	3.05	69.47

Table 2: Model test results on the validation dataset

4.4 Performance Results

Interestingly, I was only able to replicate the QANet speedup over BiDAF on CPUs and not on GPUs. On a CPU, I observed a 3.3X speedup, which is within the range of what the original paper found. However, on GPU, the QANet is slower and only runs 0.83X as fast as BiDAF. Note that GPU utilization is high on GPU runs as the CPU time matches the GPU time, meaning that the CPU is waiting for GPU calls to complete, as we'd expect from an ML workload that is doing nearly all computation on the GPU rather than the CPU 4.4. For the forward pass, QANet ran 2.2X faster as BiDAF on CPUs and nearly identically on GPUs 4.4. On the backward pass, QANet ran 3.9X faster then BiDAF on CPUs and 0.8X as fast on GPUs.

Model	Hardware	CPU Time	GPU Time	Total Time
BiDAF	CPU	136.969s	0s	136.969s
BiDAF	GPU	3.108s	3.109s	3.109s
QANet	CPU	41.786s	0s	41.786s
QANet	GPU	3.721s	3.721s	3.721

Table 3: Model train iteration runtime with batch size 64

Model	Hardware	CPU Time	GPU Time	Total Time
BiDAF	CPU	35.661s	0s	35.661s
BiDAF	GPU	1.753s	1.753s	1.753s
QANet	CPU	15.735s	0s	15.735s
QANet	GPU	1.726s	1.726s	1.726s

Table 4: Model forward iteration runtime with batch size 64

Model	Hardware	CPU Time	GPU Time	Total Time
BiDAF	CPU	101.290s	0s	101.290s
BiDAF	GPU	2.118s	2.118s	2.118s
QANet	CPU	26.029s	0s	26.029s
QANet	GPU	2.588s	2.588s	2.588s

Table 5: Model backward iteration runtime with batch size 64

5 Analysis

5.1 Model

While disappointing that the QANet models could not outperform the BiDAF with character embeddings, it is not all that surprising given that the GPU performance of QANet turned out to be slower. In the original QANet paper, the QANet model outperforms by being faster than BiDAF and using

than directly executing them. To see this, consider that a 4X performance increase on CPUs actually slowed GPU code because the many smaller operations that make up QANet increase the instruction overhead absent custom GPU kernels or better GPU operation compilation enough to wipe out that architectural speed up. Shrinking bigger QANet operations into several smaller still operations as Linformers and Reformers do would likely only increase this overhead.

6 Conclusion

While the QANet implementation did not perform well, this project was a great way to explore model training performance and uncovered an interesting, if artificial tradeoff between operation efficiency and number of operation kernels used for GPU computations. While Nvidia has eliminated this trade-off for LSTMs with a custom GPU kernel in CuDNN, this result does underscore the work on matrix operation compilers that could effectively fuse many matrix operations into a single, optimized kernel without the engineering effort of writing a highly-optimized GPU kernel in a low-level language like C.

7 Acknowledgements

This project was a great learning experience for me that was a fantastic introduction to natural language processing, deep learning systems and a great vantage point to see how ML and systems fit together. Thanks to the course staff for making this project and the learning this quarter possible!

8 Appendix: Figures

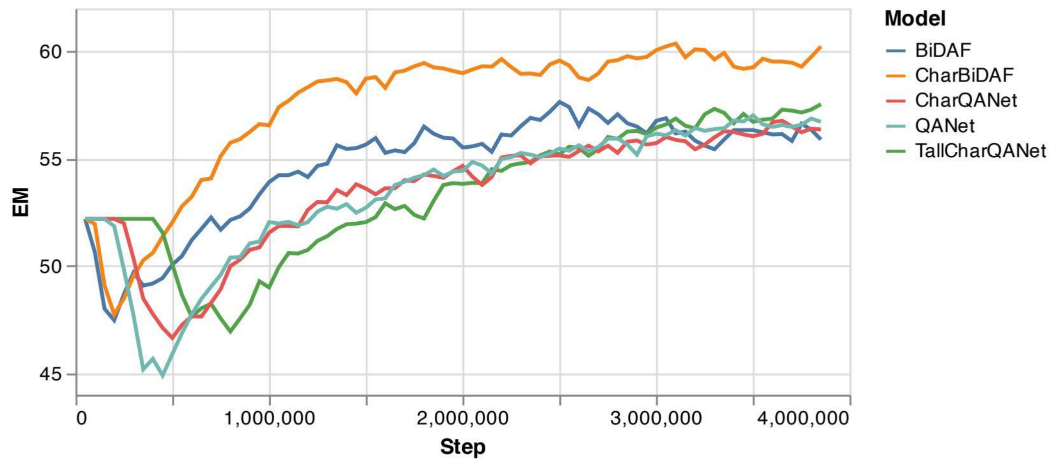

Figure 6: Dev EM score during training

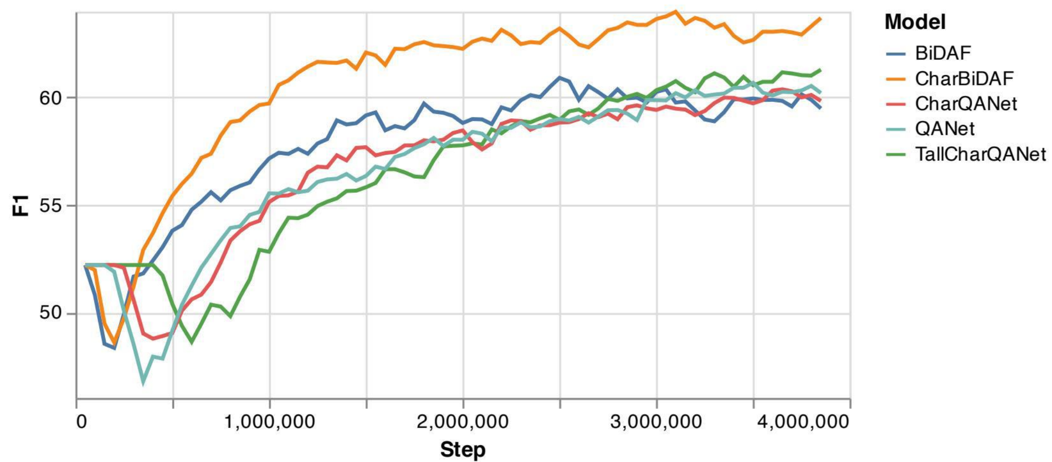

Figure 7: Dev F1 score during training

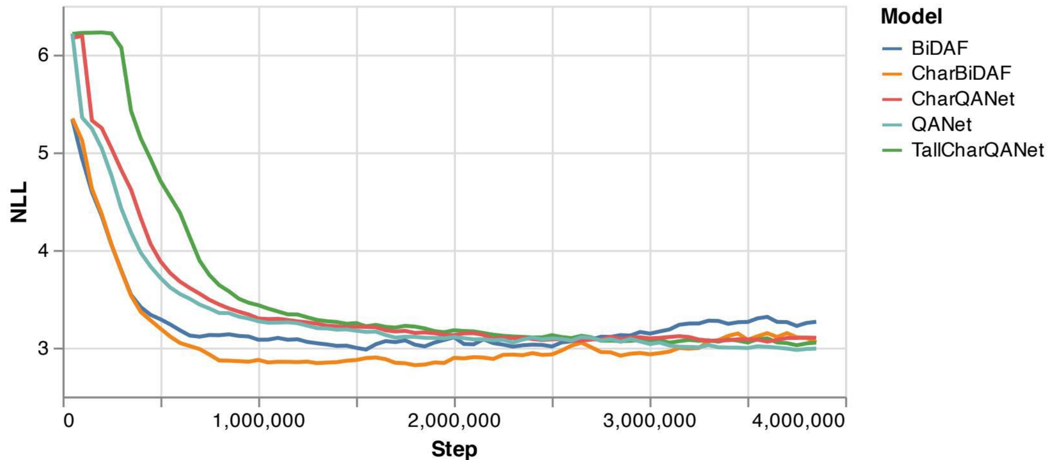

Figure 8: Train log loss during training

References

[1] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. *arXiv:1611.01603 [cs]*, June 2018.

[2] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V. Le. QANet: Combining Local Convolution with Global Self-Attention for Reading Comprehension. *arXiv:1804.09541 [cs]*, April 2018.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv:1706.03762 [cs]*, December 2017.

[4] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The Efficient Transformer. *arXiv:2001.04451 [cs, stat]*, February 2020.

[5] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-Attention with Linear Complexity. *arXiv:2006.04768 [cs, stat]*, June 2020.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]*, May 2019.

[7] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient Transformers: A Survey. *arXiv:2009.06732 [cs]*, September 2020.

[8] Yoon Kim. Convolutional Neural Networks for Sentence Classification. *arXiv:1408.5882 [cs]*, September 2014.

[9] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[10] Yu-An Wang and Yun-Nung Chen. What Do Position Embeddings Learn? An Empirical Study of Pre-Trained Language Model Positional Encoding. *arXiv:2010.04903 [cs]*, October 2020.