Building a QA system through local convolution with global self-attention (IID SQuAD track)

Stanford CS224N Default Project

Hugo Vergnes
Department of Statistics
Stanford University
hvergnes@stanford.edu

Hamza El Mosor
ICME
Stanford University
helmosor@stanford.edu

Abstract

Recurrent Neural Networks (RNN) were historically successful architecture for Machine Reading comprehension. QANet was the first architecture to achieve higher performance with only convolutions and self-attention. In this paper, we discuss our implementation of this architecture with an emphasis made on the encoder block and the choice of the optimization algorithm. We tested our implementation on Stanford Question Answering Dataset 2.0 [1].

1 Key Information to include

- Mentor: Grace Lam
- External Collaborators (if you have any): No
- Sharing project: No

2 Introduction

Machine Reading Comprehension (MRC) and Question Answering (Q&A) have been a growing research topic since the surge of public datasets like SQuAD [1]. Popular models treating this type of NLP tasks were initially based on recurrent networks [2]. These sequential models have the following drawbacks: they require a long time to train (as well as a long time at inference time), which prevents training the model on large datasets. These issues prevent such models to be implemented in real-time applications. The QANet paper proposes a method to get rid of the sequential aspect of these models. To do so, the QANet model used convolutions along with self-attention instead of RNNs in order to encode the query and the context of the Q&A task. The author reported a significant gain in computation time and performance.

Our approach was to build up from the initial architecture of the QANet model in order to achieve the best possible F1 score and EM score on the SQuAD dataset [1], thus improving from the BiDAF architecture [2].

3 Related Work

The BiDAF architecture [2] was the state of the art when QANet was published [3]. One of the main contributions of that paper was the bidirectional attention flow layer that coupled the query and the context to produce a set of query-aware vectors for each word in the context. The novelty is that the attention flow layer is not used to summarize the query and context into single feature vectors. The model focuses preferentially on certain words given a certain question. One of the limitations of this architecture is that the probabilities of the beginning and end of the answer are created independently, while one can expect it to be correlated.

QANet uses the contribution of the BiDAF architecture but removes the RNN bottleneck and replaces it with blocks using only convolutions. This led to a significant increase in performance.

When transformers were introduced [4], they demonstrated that higher performance could be achieved with only self-attention. The proposed architecture relies solely on the use of self-attention, where the representation of a sequence is computed by relating different words in the same sequence. It is interesting to note here that Transformers (state of the art in most NLP tasks), uses some of the same building blocks as QANet.

4 Approach

We coded from scratch the QANet network as described in the original paper [3] and illustrated in 1. In itself, this was already a fairly challenging task. Out of the 5 different layers in QANet (an embedding layer, an encoder layer, a context-query attention layer, a model encoder layer, and an output layer), only the context-query attention layer is identical to the BiDAF implementation. Thus we shared the same context-query attention implementation between the two models. The encoder discards the use of RNN-encoder, thus we can run the model in parallel that increased the forward and backward pass of the model.

QANet has a complex architecture including a lot of layers. A choice has been made to reduce the number of Encoders put Stacked Model Encoder blocks. Indeed, the original paper proposed 7 stacked encoders per encoder block. In this setting, training is very slow and we don't have enough computation resources to train this model several times. We thus decided to run more experiments on a smaller model using only 4 or 5 encoder blocks per 'Stacked Model Encoder Blocks'.

In the original paper, the authors detail the use of back-translation as a data-augmentation technique. Data-augmentation techniques both during train and test time are usually a good way to increase performance. We made the choice to leave out this part of the paper to focus more thoroughly on the implementation of the QANet paper as well as an analysis of its performance.

Our baseline corresponds to the initial BiDAF model given as a starter code, which we enhanced by adding the character embeddings like in the initial paper [2] (which was used for our implementation of QANet as well). The character embedding layer corresponds to a convolution layer of the input matrix (formed by the concatenation of the character embeddings of a given word), followed by a max-pooling layer to reduce the size of the embedding. The final output is concatenated with the word embedding to have the final embedding of the word.

5 Experiments

5.1 Data

The dataset that will be used is taken from the SQuAD 2.0 dataset (the dev set and the test set will be taken from the dev set of the official dataset). As explained in the project handout, the dataset breaks down as follows:
• train (129,941 examples): All taken from the official SQuAD 2.0 training set.
• dev (6078 examples): Roughly half of the official dev set, randomly selected.
• test (5915 examples): The remaining examples from the official dev set, plus hand-labeled examples.

The particularity of the SQuAD 2.0 dataset is that a third of the instances of the dataset has no answers in the given context, which means that the model should be able to figure out when there is or there is not an answer.

5.2 Evaluation method

We will use the F-1 score and the Exact Match (EM) to evaluate our model, as advised in the project handout. Since we are also interested in reducing the computation time, we will also report the time for inference and training (time per batch and time to convergence).

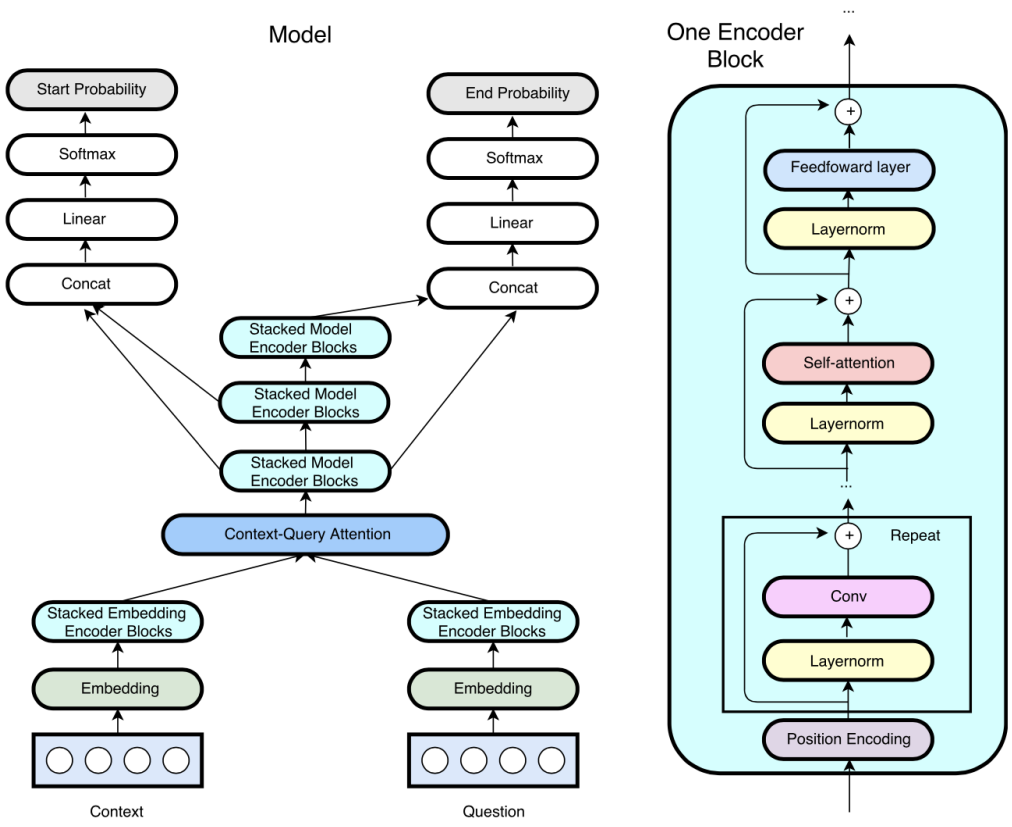

Figure 1: The original QANet architecture. The original implementation has 7 encoder blocks.

5.3 Experimental details

We aimed at having results that were relevant to the initial QANet paper. Thus we used a hidden size of 128. Due to computational costs on Azure, we limited the number of epochs to 30, which results in around 3.85M iterations. We tested different learning rates, but performance remained close to identical, so until specified otherwise we used a default learning rate of 0.5. We used a batch size of 32 for QANet. Unless specified otherwise we used the optimizer Adadelta with no weight decay. Unless specified otherwise we didn't use a learning rate scheduler. The dropout rate was fixed at 0.15, gradients were clipped at 5.0 to avoid throwing off the network early in the training. And the decay rate for the exponential moving average of parameters was fixed at 0.999.

All experiments were run on Azure by Microsoft using a standard NC6s v3 (6 vcpus, 112 GiB memory) using the credits provided by the course staff.

The hyperparameters values are summed up below:

- Batch size: 32
- Epochs: 30
- Drop out probability: 0.15
- Learning rate: 0.5
- Hidden size: 128
- Exponential moving average weight decay: 0.999
- Gradient clipping: 5.0

5.4 Results

The experiments concerning the QANet architecture are summarized in 1. On the test leaderboard, 63.823 F1-score and 60.169 EM for our QANet model with 6 encoder blocks.

It is very clear that the QAnet architecture outperforms the BiDAF network as we gained 10% in the F1-score against the baseline and 6% when the baseline is used with character embedding. It appears that the RNN encoder is indeed underperforming against the encoding block proposed by the original paper of QANet. When inspecting the answers, it appears that both architectures produce very good answers when the question asks for a date. They perform equally well when the answer is very short and require only a word (Thermodynamics, Interventionism, etc.). It appears that QANet outperforms BiDAF the most when the answer is a few words or longer. For example, the answer to: 'What president eliminated the Christian position in the curriculum?' is 1869-1909 for BiDAF and 'Charles W. Eliot' for QANet.

QANet is also remarkably efficient. The BiDAF architecture uses 2.4M trainable parameters while the QANet model with 7 encoding blocks per stacked module uses 1.2M trainable parameters with the hyper-parameters given by the original paper (Hidden size: 128 and 7 encoder blocks per stack). Each encoder accounts for 153k parameters, thus reducing the number of encoder blocks per stack is a good way to reduce. We explore this more in-depth and try to speed up training time by reducing the number of blocks stacked encoders.

In the original QANet paper [3], it is noted that this model is faster than the BiDAF architecture. We have not noted an increase in speed between the two architecture, this might be attributed to better parallelization of RNN encoders on modern GPU.

In our implementation, we see that the model doesn't lose too much performance by down-scaling the number of encoder block per stacked module. However, the gain in computing is quite significant. As we can see in 1, the performance drops significantly when we try to reduce the number of blocks below 3 encoding blocks. It remains stable mostly stable until that point.

Models	*Dev NLL*	*F1*	*EM*	*AvNA*
BiDAF	3.05	56.69	56.07	66.88
BiDAF w/ character embedding	3.05	60.55	57.18	67.11
C2Q attentions	3.24	55.56	52.33	64.17
CoAttention	3.14	58.81	55.42	66.32
CoAttention + LSTM dropout	3.34	59.96	53.22	64.18
QANet 6 blocks	**2.55**	**66.36**	**63.23**	**73.34**
QANet 5 blocks	2.59	66.23	62.75	72.51
QANet 4 blocks	2.57	66.28	63.08	72.11
QANet 3 blocks	2.63	65.67	61.97	71.57
QANet 5 blocks + 0.4 lr	2.65	65.69	62.19	72.00
QANet 5 blocks + CoAttention	x	x	x	x

Table 1: Performance on the Dev Set

6 Analysis

One of the outcomes of our experiments is that the coattention layer doesn't improve performance. Notice that the coattention method uses a lot more trainable parameters than the simple trilinear attention as used in BiDAF or QANet. Indeed, trilinear attention uses 385 trainable parameters for a hidden size of 128. For the coattention mechanism, depending on the size of the output LSTM this can easily be around 1M trainable parameters. This creates an imbalance of bias vs variance. The simpler trilinear attention is easier to train but will explain less variance whereas the more sophisticated Co-Attention layer can potentially approximate more complicated functions but its training is harder. In our attempts at refining the coattention layer, we tried different dropout rates and sizes of output LSTM. We reported the result of the best implementation (LSTM layers is 1 and dropout rate is 0.15). It managed to converge for the BiDAF architecture, even though it does not improve the baseline. However, coattention in QANet prevented the model from converging properly.

Since this layer is used multiple times, it adds too many layers to the model and creates gradient vanishing issues. One final remark is that the coattention paper uses a dynamic decoder that iterates over the potential answer span (As opposed to what is given in the handout of the default project). This does not match the decoder used in QANet where we concatenate output from the 3 different encoder blocks. In future work, it might be relevant to try to adapt the decoder to match more closely the used coattention decoder.

We also ran experiments using different optimizers. Our initial starting point was to use Adadelta as an optimization algorithm [5]. This algorithm is an extension of Adagrad [6]. However, it only accumulates the past gradient for a defined window w instead of accumulating all past gradients. Adadelta has the nice property that it eliminates the learning rate from the update rule. This removes the need to search for the best learning rate. Note here that the Pytorch implementation requires a learning rate, but it is used as an initial learning rate only.

We ran experiments using Adaptative Moment Estimation (Adam) optimizer [7]. Adadelta was a very popular choice before Adam optimizer was published but since then it has slowly been replaced. This optimizer stores an exponentially decaying average of the past squared gradients like Adadelta and RMSprop. It also provides an exponentially decaying average of the past gradients. The authors propose to use $(\beta_1, \beta_2) = (0.9, 0.999)$ where those parameters control respectively the weight on the past momentum and the past squared gradients (see equations (1) and (2) below). It appears that using those parameters coupled with a learning rate of 0.001 as proposed in the QANet paper increase the convergence speed and the performance of the model using Adadelta. The original QANet paper proposed the following values $(\beta_1, \beta_2) = (0.8, 0.999)$. This means that we put the previous gradients at time step $t - 1$ has less weight against the average as compared with the values proposed by the authors of the Adam paper. We observed a very small gain by doing so as seen in table 2. Since we only used one run of QANet to obtain those results, they are not statistically significant and we would need to reproduce this several times to confirm that this improves performance. Indeed, variance at the end of the training between two validation steps is around 0.2 in terms of F1 score. Which is similar to the gain in performance we observed.

In our experiments, Adam increases convergence speed drastically in comparison with Adadelta. Indeed, the F1 score reaches 60% in 350k steps whereas Adadelta requires 550k steps to reach the same performance.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{1}$$

Equation 1: Mean update used in Adam

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2}$$

Equation 2: Variance update used in Adam

To further refine our comparison of the result obtained between Adam and Adadelta we used a third optimizer: RMSprop. This is an unpublished algorithm that is fairly close to Adadelta. However, it requires a learning rate that will be used at each time step, which we set at 0.001 to be comparable with Adam. Using RMSprop as the optimizer helps us achieve an F1 of 66.04 as well as an EM of 62.34. As RMSprop does not improve performance significantly against Adadelta we then conclude that it is the exponentially decaying average of the past gradients used in Adam optimizer that improves convergence.

QANet authors also reported a different value used for numerical stability $\epsilon = 10^{-7}$. We kept the default value of $\epsilon = 10^{-8}$ as we didn't observe significant numerical stability issues. The experiments using different optimizers are summarized in table 2.

Models	*Dev NLL*	*F1*	*EM*	*AvNA*
QANet 4 blocks + Adadelta	**2.57**	66.28	63.08	72.11
QANet 4 blocks + RMSprop optimizer	2.78	66.04	62.34	72.51
QANet 4 blocks + Adam optimizer + lr 0.001	2.62	66.63	62.96	**73.23**
QANet 4 blocks + Adam optimizer + lr 0.001 + β_1 0.8	2.67	**66.66**	**63.15**	73.18
QANet 4 blocks + Adam optimizer + lr 0.5	x	x	x	x

Table 2: Performance on the Dev Set using different optimizers

7 Conclusion

The models we experimented with were all derived from the published architecture of BiDAF and QANet, with some variations and inspirations taken from other papers (such as coattention). We can notice that simple ideas such as adding character embeddings give a big boost in our metrics, which highlights the fact that improvements in Machine Learning Comprehension model can sometimes be achieved without any extravagant solutions.

We also highlighted different performance and convergence speed under different gradient descent algorithms. This includes giving elements to sustain that the exponentially decaying average of the past gradients used in Adam optimizer that improves convergence

Our models can undoubtedly be improved with a more extensive hyper-parameter search and longer training time. For example, changing the hidden sizes or the number of attention heads can have non-negligible impacts on the model's performance. QANet was also implemented with 7 encoding blocks but we needed to use fewer to save computational resources. We showed that QANet seems to hold decently well when shrinking the size of the encoder.

References

[1] Konstantin Lopyrev Percy Liang Pranav Rajpurkar, Jian Zhang. Squad: 100,000+ questions for machine comprehension of text. 2016.

[2] Ali Farhadi Hannaneh Hajishirzi Minjoon Seo, Aniruddha Kembhavi. Bidirectional attention flow for machine comprehension. 2016.

[3] Minh-Thang Luong Rui Zhao Kai Chen Mohammad Norouzi Quoc V. Le Adams Wei Yu, David Dohan. Qanet: Combining local convolution with global self-attention for reading comprehension. 2018.

[4] Niki Parmar-Jakob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Shazeer. Attention is all you need. 2017.

[5] Matthew D. Zeiler. Adadelta: An adaptive learning rate method. 2012.

[6] Yoram Singer John Duchi, Elad Hazan. Adaptive subgradient methods for online learning and stochastic optimization. 2011.

[7] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization. 2014.