A PyTorch implementation of QANet

Stanford CS224N Default Project

Dat Nguyen
Department of Computer Science
Stanford University
dpnguyen@stanford.edu

Grading option: no grades needed. I am only looking for feedbacks; please feel free to take your time or even ignore this if you are busy. Thank you!

Abstract

We implement QANet, a question-answering model based on convolutions and self-attentions, and report on the implementation details needed to make the model run well on PyTorch. Our best single model achieved an F1 & EM score of 64.35 & 60.694 on the SQuAD 2.0 course test set[1] which ranked 12-th in the non-PCE leader board at the time of writing.

This report is written using the CS224N template[2] which is built on NeurIPS 2019 template[3].

1 Key Information to include

- Mentor: Nick Hirning
- External Collaborators (if you have any): None
- Sharing project: None

2 Introduction

In [1], Adams et al proposed QANet, a question-answering model based on convolutions and self-attentions instead of recurrent neural networks. At the time of publication of the paper (Apr 23, 2018), the most successful question-answering models were generally based on recurrent neural networks (RNNs) with some attention mechanism. One weakness of these models is that their recurrent nature prevents parallel computation, making both training and inference slow. There had been efforts to speed up the RNNs by avoiding bi-directional attention [2] or deleting the context-query attention module [3], but these models had to sacrifice accuracy: On the SQuAD v1.1 dataset, their F1 scores were around 77.

QANet solves this problem by moving away from RNNs and instead using convolutions and self-attentions as the main building blocks. This feed-forward nature gives the model a significant speed advantage (reportedly between 3 to 13 times faster in training and 4 to 9 times in inference) over its RNN counterparts. Taking advantage of this speed boost, the authors trained the model with augmented data and achieved an F1 score of 84.61 on the SQuAD dataset, which was significantly better than the best published result of 81.8 at the time.

In this project, we implement QANet on PyTorch and test on SQuAD 2.0. While the transition to SQuAD 2.0 itself is straightforward, it is difficult to reproduce the performance, especially speed, reported in the original paper. This issue has been reported by many students in the class and many

---

[1] This course test set is different from the official test set. See section 5.1
[2] https://www.overleaf.com/project/5e6087e7b2da9800019ce574

Stanford CS224N Natural Language Processing with Deep Learning
open Github repositories. While we could not resolve this issue, we identified a few ingredients that might play an important role in the model’s performance. We describe them and our experiments in section 5.

On the course SQuAD v2.0 test set, our best single model achieved an F1 & EM score of 64.35 & 60.694, which ranked 12-th in the non-PCE leader board at the time of writing. This performance is in line with the remark by [4] that “a strong neural system that gets 86% F1 on SQuAD 1.1 achieves only 66% F1 on SQuAD 2.0” and the reported 84.61 F1 score on SQuAD 1.1 of QANet.

3 Related Work

Machine question-answering has been an active area of research in the past few years. Datasets like SQuAD [4] and TriviaQA [5] are now very popular benchmarks for many NLP studies. The information-retrieval nature of these datasets makes the task more accessible and has allowed the success of many models with attention mechanism, including BiDAF [6], FusionNet [7], and QANet [1).

Prior to QANet, most successful SQuAD models used an RNN-based encoder-decoder architecture, which was also popular in many other NLP tasks. A breakthrough was made in [8], where the Transformer architecture was proposed. This architecture relies solely on attentions, achieved great results in neural machine translation, and has since then shown advantages over RNNs in many other tasks.

There has been works [9] showing that, theoretically, multi-head attentions with enough heads can replace convolutions. Nevertheless, QANet uses both convolution and self-attention. The authors reported that adding convolutional layers gave a 2.7 F1 gain on SQuAD 1.1 and enabled the use of stochastic layer dropout [10], which gave another 0.2 F1 gain.

4 Approach

4.1 Model overview

For the most parts, our implementation of QANet closely follows the original paper. The model consists of 5 main parts, illustrated in the left part of figure 1: An input embedding layer, an embedding encoding layer, a context-query bi-attention layer, 3 repeated model encoding layers, and an output layer.

Input Embedding Layer The model uses the standard technique of representing words by concatenating their word vectors and their character vectors. The word vectors are pretrained vectors from GloVe and are fixed during training, while the character vectors are trainable vectors obtained by convolving the vectors of every character of the word. The concatenated vectors are then passed through a two-layer highway network, like in [6].

Embedding Encoding Layer and Model Encoding Layers Each embedding encoding layer and model encoding layer is built from 1 and 7 (respectively) encoder blocks, illustrated in the right part of figure 1. Each encoder block consists of a positional encoding layer, several convolution layers, a self-attention layer, and a feed-forward layer. The idea is that “convolution captures the local structure of the text, while the self-attention learns the global interaction between each pair of words.” The positional encoding sublayer and self-attention sublayer are the same as those in the Transformer model [8]. The convolutional sublayers uses depthwise separable convolutions ([11] and [12]), which has fewer parameters than traditional convolutions. The feed-forward sublayer is a composition of linear layers and ReLU activation.

The Embedding Encoder Layer precedes the Context-Query Bi-attention Layer, which is followed by 3 repeated Model Encoder Layers.

Context-Query Bi-attention Layer This module is pretty standard: A similarity matrix $S$ between context and query words is use to compute the context-to-query and query-to-context attentions: $A = \bar{S} \cdot Q^\top$ and $B = S \cdot C^\top$, where $Q$ and $C$ are the encoded query and context, and $\bar{S}$ and $\bar{S}$ are the row- and column-normalized matrix of $S$ using softmax.
Output Layer  The outputs of the first 2 model encoder blocks are passed through a linear and then a softmax layer to compute the probability distribution of the starting position of the answer. Similarly, the probability distribution for ending positions is computed using the first and third encoder block. The loss function is then the negative log probabilities of the true starting and ending positions, averaged over the dataset.

Inference  At inference time, the answer is chosen to maximize the product of the probabilities of starting and ending positions. To accommodate unanswerable questions, we add an out-of-vocabulary word at the beginning of every context paragraph and every question. If the model selects this out-of-vocabulary word as the answer, we make it predict “not answerable.”

4.2 Baseline

For our baseline, we use the BiDAF model proposed in [6]. The course starter code provide an implementation of this model without the character-level embedding layer. We add this layer to the starter code to obtain a slightly stronger baseline.

4.3 Implementation

We implemented the character-embedding layer using previous course assignments. For the encoder blocks, we looked at the transformer tutorial [13] and the QANet TensorFlow implementation [14] for references but wrote our own PyTorch codes. Everything else is reused from the course starter code with minor modifications.

Initially, we planned to implement and use the data augmentation process described in the original paper. However, our model was very slow, and we believe that the data augmentation process only fits nicely into the original paper because their QANet model is much faster than everything else. Thus, we lost interest in data augmentation and decided to focus on speeding up our model. Unfortunately we did not succeed.
5 Experiments

5.1 Data

We use the SQuAD 2.0 dataset provided in the starter code. The official SQuAD dataset consists of roughly 150k pairs of paragraph-question. The paragraphs are from Wikipedia, and each question is either unanswerable using the provided paragraph or has an answer that is a chunk of text taken directly from the paragraph. This means that the model has to decide whether a question is answerable, and if so, select a span of text in the paragraph that answers the question. About half of the questions are unanswerable.

The course splits the official SQuAD dev set into a smaller dev set (consisting of 6087 examples) and a course test set (5915 examples). The course test set is released to students, who are expected to submit their answers in a CSV file. This makes the submission process simpler and at the same time keep the official SQuAD test set secret.

5.2 Evaluation method

We use the default EM and F1 metrics from SQuAD: The EM metric gives 1 point for answers that are exactly the same as the reference answers and 0 points for others. The F1 metric is less strict: it is the harmonic mean of the precision and recall of predictions. We also report AvNA, which is the percentage of correct predictions of whether or not a question is answerable.

5.3 Experimental details and analysis

In this section we describe the configurations that we have experimented with. Because there are a lot of experiments, we describe their effects to performance right away instead of doing that in a separate “results” section.

5.3.1 Hyperparameters

Common settings For the most parts, we stay close to the settings in the original paper: We use an Adam optimizer with $\beta_1 = 0.8, \beta_2 = 0.999, \epsilon = 10^{-7}$ and $3 \cdot 10^{-7}$ L2 weight decay; the learning rate increases inverse-exponentially from 0 to 0.001 in the first 1000 steps then stays constant after that. We use a dropout rate of 0.1 for word-level embeddings, 0.5 for character-level embeddings. Inside each embedding encoding layer and model encoding layer, the sublayer at position $l$-th has dropout rate $0.1 \frac{1}{L}$, where $L$ is the total number of sublayers in the layer. There is also a dropout layer of rate 0.1 between every main layers of the model.

Embedding dimension For word vectors, we use 300-dimensional pretrained GloVe vectors for word-level embedding and experimented with 100- and 200-dimensional learnable character-level embedding. Even though the original paper uses 200 for character-level embedding dimension, we got better results with 100.

![Figure 2: Performances on dev set. The horizontal axis is the number of steps. In magenta: one of our best performing models. In gray: the same model but with character embedding size 200 instead of 100.](image-url)
**Hidden size** The hidden size throughout the model is 128, like in the original paper. We also experimented with hidden size 96 and found that it decreased both F1 and EM scores by around 0.5 - 1.0 points while not giving any speed gain. For example, switching the hidden size from 128 to 96 from our best model decreased both F1 and EM scores by 0.8.

**Attention heads and batch size** We could not use 8 attention heads with batch size 32 like in the original paper due to the lack of GPU memory. This is consistent with the findings in [14]. The biggest model we could fit in our 8GB GPU memory has 4 attention heads and batch size 32; even this model needed some memory optimization to fit in and constantly used 8122 out of 8129 MB of available memory.

In our experiments, models with 8 attention heads and batch size 16 started out better but ultimately performed worse than those with 4 attention heads and batch size 32.

![Figure 3: Performances on dev set. The horizontal axis is the number of steps. In teal: our best-on-test-set model (which is actually second best on dev set, see figure 5 and the remark above it). In gray: the same model but with 4 attention heads instead of 8 and batch size 16 instead of 32.](image)

**5.3.2 Layer implementation**

**Weight initialization** For most layers, we use PyTorch’s default initialization. However, for most linear projections, we experimented with Xavier initialization and Kaiming initialization. We found that it was crucial to use Xavier initialization for the linear layers in the Context-Query Bi-attention layer and the Output layer:

![Figure 4: Performances on dev set. The horizontal axis is the number of steps. In teal: our best-on-test-set model (which is actually second best on dev set, see figure 5 and the remark above it). In cyan: the same model but with Kaiming initialization for the linear layers in the Context-Query Bi-attention layer and the Output layer. The difference in F1 scores is at least 1.5 for the entire training process.](image)

We suspect that this is one reason we (and many other student groups, according to their Piazza posts) initially got performances that were significantly worse than those obtained by the TensorFlow implementations like [14]. PyTorch by default uses Kaiming initialization for linear layers, while TensorFlow uses Xavier initialization.

**Depthwise Separable Convolution** The default way to implement depthwise separable convolution in PyTorch is to use the `groups` feature in `Conv1d`. We experimented with some other options:

1. Simply replacing depthwise separable convolution with traditional convolution.
2. Implementing depthwise separable convolution as a traditional convolution whose filter is the product of two smaller-ranked filters: If the cross-correlation (which is how PyTorch really implements convolution) weights of the pointwise and depthwise layers are $P(o, i, 1)$ and $D(o, 1, k)$, then their composition is a traditional cross-correlation with weight $W(o, i, k) = P(o, i)D(o, k)$.

3. Instead of thinking of each input as an 1D sequence of length $L$ with $D$ features, we think it as a 2D input of width $L$, height $D$, and 1 feature, then apply a Conv2d to it with kernel size $k \times 1$. This is equivalent to a depthwise separable convolution where all features share the same depthwise filter.

We did not observe any significant difference in speed between the default implementation and option 1 and 2, despite the fact that depthwise separable convolution uses much fewer parameters and computations than traditional convolution. This suggests that the implementation of depthwise separable convolution in PyTorch has not been optimized. This finding is consistent with many reports on discuss.pytorch.org.

Despite having fewer parameters and using a well-optimized module Conv2d, option 3 was the slowest in our experiments: switching from the default implementation to option 3 raised the time per epoch from around 45-50 minutes to 60-70 minutes. We do not understand why this was the case.

Accuracy-wise, we did not notice any significant difference in F1 and EM scores among these implementations after 4 epochs. Due to time constraint we did not run these experiments for longer and went with the default implementation.

**Activation: a curious case.** We use leaky ReLU for each convolutional layer. We found that this choice performed slightly better on the dev set than the ReLU activation. Interestingly, when we forgot to include the activation by mistake, the model performed better on the dev set (+0.65 F1) but worse on the test set (-0.79 F1).

![Figure 5: Performances on dev set. The horizontal axis is the number of steps. In magenta: our best performing model on the dev set. This model does not use any nonlinear activation in convolutional layers. In teal: the same model but with leaky ReLU activation in convolutional layers. These are the two models that we submit to the test leader board. The second model is outperformed on the dev set but gets +0.79 F1 on the test set.](image)

Theoretically, a composition of multiple convolutions without activation is just one convolution with a bigger kernel size. However, when we tried reducing the number of convolutions in each encoder block and increasing the kernel size, the performance was much worse. We do not have an explanation for this phenomenon.

**Self-attention** We use the nn.MultiheadAttention module from PyTorch. We also experimented with using the nn.TransformerEncoderLayer module, which combines our self-attention layer and feed-forward layer, except that it applies LayerNorm and skip connection in a different order than that of the original paper. We found that the former approach led to better results. This might imply that applying LayerNorm in the right order is very important for our model.

**Sharing weights in the model encoder** As in the original paper, we use different weights for the 7 encoder blocks in each repetition of the model encoder layer. We also experimented with using the same weights for all these blocks. In the latter approach, a linear layer with skip connection is applied between each two blocks to add more expressive power. In our experiments, sharing weights
reduced the dev F1 and EM by 1.5. This approach is also slower and needs more memory, probably because its computational graph has an extra linear layer.

**Inference**  
The default inference method in the starter code is as follows: Recall that we add an out-of-vocabulary word at the beginning of every context paragraph to represent the "not answerable" option. A pair of \((\text{start}, \text{end})\) positions is legal if:

1. \(\text{start} \leq \text{end} < \text{start} + M\) for some chosen max length \(M\). In our case \(M = 15\).
2. If \(\text{start} = 0\) then \(\text{end} = 0\). Here 0 is the index of the added out-of-vocabulary word.

Once the probability distributions \(p_1\) and \(p_2\) for the starting and ending positions of the answer are predicted, we compute the joint probability \(p_1 \times p_2\) for each \((\text{start}, \text{end})\) pair and choose the pair with the highest probability among those who are legal. If \(\text{start} = \text{end} = 0\), the model predicts "not answerable."

We experimented with a slightly different method: Instead of computing the joint probability \(p_1 \times p_2\), we simply let the model independently pick the starting position with the highest \(p_1\) and the ending position with the highest \(p_2\). If the choice is illegal (in the above sense), the model predicts "not answerable."

In our experiments, the second inference method generally led to a small increase (around \(0.1 - 0.2\)) in dev F1 and EM scores. This might suggest that the model can learn to predict legal pairs by itself most of the time, and in the cases it gets confused by an illegal pair, it's generally better to make the model predict "not answerable" instead of forcing it to choose the best legal pair.

However, we do not know if this is specific to our model: In all of our experiments, QANet was overconfident and predicted "not answerable" only about 43% of the time, while roughly 50% of the questions are unanswerable. It might be the case that the second inference method slightly improved our performances only because it always makes models predict "not answerable" more often.

**Remark.** We only experimented with the second inference method at test time. All evaluations during training use the first inference method. In particular, all Tensorboard plot in this workshop use the first inference method.

### 5.4 Results

Our best model achieved a 64.35 F1 and 60.964 EM score on the course test set, which ranked 12-th in the non-PCE leader board at the time of writing. This model uses 100-dimensional character-level embedding, hidden size 128, 4 attention heads, batch size 32 and Xavier initialization for most linear weights. It took about 48 minutes per epoch and 20 hours total to train.

To compare, the baseline BiDAF model (with character embedding) achieved 64.01 F1 and 60.85 EM on the dev set. It took about 20 minutes per epoch and 8 hours total to train.

<table>
<thead>
<tr>
<th>Model</th>
<th>dev set</th>
<th>test set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EM</td>
<td>F1</td>
</tr>
<tr>
<td>BiDAF (starter code)</td>
<td>58.19</td>
<td>61.55</td>
</tr>
<tr>
<td>BiDAF (with character-level embedding)</td>
<td>60.85</td>
<td>64.41</td>
</tr>
<tr>
<td>QANet</td>
<td>63.43</td>
<td>66.63</td>
</tr>
<tr>
<td>QANet (without activation in convolution)</td>
<td>63.89</td>
<td>67.28</td>
</tr>
</tbody>
</table>

Table 1: Performances of the baseline BiDAF model and our best QANet models.
our implementation was suboptimal but we are not sure what exactly was wrong: For each layer our implementation seems fairly standard, and during training our GPU utilization was almost always around 99%.

Another hypothesis is that our 8GB GPU memory is too small for the parallelizable nature of QANet to show advantage over BiDAF. We obtained permission from the teaching team to use a virtual machine with bigger memory to test this hypothesis; however, unfortunately we could not carry out this experiment. This is because none of the bigger GPUs that are available to our subscription actually have 16 GB memory; they are just multiple copies of the smaller GPU bundled together. This means that to fully parallelize our training, we would need to do more than simply distributing the batch size using `nn.DataParallel`. We ended up not being able to do that.

Figure 6: Performances on dev set. The horizontal axis is time. In teal: Our best-on-test-set model. In red: The baseline BiDAF model with character-level embedding. The two models reached 64.41 F1 around the same time. After that our QANet model quickly reached 65 F1, while the baseline model stagnated.

6 Qualitative Analysis

Here are some randomly picked questions where our QANet model did not answer correctly.

**UUID:** 18dc1aeb3598fa95a81bb8ecd

**Question:** What type of arts flourished in the Yuan?

**Context:** In the China of the Yuan, or Mongol era, various important developments in the arts occurred or continued in their development, including the areas of painting, mathematics, calligraphy, poetry, and theater, with many great artists and writers being famous today. [...] Another important consideration regarding Yuan dynasty arts and culture is that so much of it has survived in China, relatively to works from the Tang dynasty and Song dynasty, which have often been better preserved in places such as the Shōsōin, in Japan.

**Answer:** painting, mathematics, calligraphy, poetry, and theater

**Prediction:** N/A

**Comment:** Maybe our model could not answer this question because it could not match the word “flourished” in the question to any word in the context. One could argue that this is a bit hard because the sentence containing the answer does not have any word that is a clear synonym of “flourished.” However, the baseline BiDAF model answered this question correctly. We are not sure why.

**UUID:** 1efa67d3bf6c0dbc3ebbce1

**Question:** Of Poland’s inhabitants in 1901, what percentage was Catholic?

**Context:** Throughout its existence, Warsaw has been a multi-cultural city. According to the 1901 census, out of 711,988 inhabitants 56.2% were Catholics, 35.7% Jews, 5% Greek orthodox Christians and 2.8% Protestants [...] After the war, the new communist authorities of Poland discouraged church construction and only a small number were rebuilt.

**Answer:** N/A

**Prediction:** 56.2 %
Comment: Our model made a mistake here probably because it thought Poland was a synonym for Warsaw. According to https://projector.tensorflow.org/, the nearest GloVe vector (in cosine similarity) to Warsaw is precisely Poland, so this mistake is very understandable. The baseline BiDAF model made the same mistake here.

7 Conclusion

In this project, we build a PyTorch implementation of QANet, a question-answering model based on convolution and self-attention instead of RNNs. While the feed-forward nature of QANet is ideal for parallel computation, we find it very difficult to take advantage of this property: In contrast to the findings of the original paper, we did not observe any speed gain over the baseline BiDAF model.

Nevertheless, our model achieved reasonable F1 and EM scores. We find that weight initialization and small implementation details play very important roles in performance. Because of this, some care is needed when one translates a TensorFlow implementation to a PyTorch one.

References


