Bi-directional attention flow

Model for Question Answering

1. Summary

Question Answering (QA) is a well-known task in the Natural Language Processing (NLP) field. It got lots of interest among many companies and research groups in the last few years because of its latent potential in understanding unstructured text.

For our task, a QA model refers to a model which takes at least two inputs, a question and a context paragraph containing the answer, and output two values: the start and end indices of the answer in the context.

One of the key factors behind the success of these models is the neural attention mechanism, which enables the system to focus on a targeted area within a context paragraph, that is most relevant to answer the question. One of the first successful attention mechanism is called the Bahdanau attention and has been first implemented to solve a neural machine translation problem in 2015.

By now, lots of others attention mechanisms have been designed, in particular the self-attention one in 2017. This mechanism is the building-block of a new great variety of models called transformers. These latest are nowadays the state-of-the-art models in NLP. They can solve almost every problem related to this field and they can do it better than other models.

Therefore, in this paper we will not talk about transformers but about the BIDAF model which implements the bi-directional attention flow (BIDAF) mechanism. Our benchmark is the SQUAD (Stanford Question Answering Dataset) dataset, which consists of questions related to a paragraph (called context). The answer of a given question is obviously contained in the corresponding context.

In the SQUAD webpage, there is a leaderboard, which classify the most successful models for this task according to one main metric which is the F1 score (actually, there is also one another metric, called EM, standing for Exact Match, but here we focus only on the F1-score). Of course, the best models are transformer-based but one of the most successful non-transformer-based models is the BIDAF one, with a F1-score of about 70 %.

In this paper we tried to implement this model. We choose the BIDAF rather than a transformer because this one is not very hard to implement but cover lots of concepts. We can do it from scratch, and so learn lots of things, whereas transformers are much more complicated, and are not very challenging to implement, because there are tons of library that allow you to do so.

Eventually, we will discuss about our results and improvements than can be made.

2. Background

In order to better understand the architecture of the BIDAF model, we need to focus on three key points that can be tricky to understand:

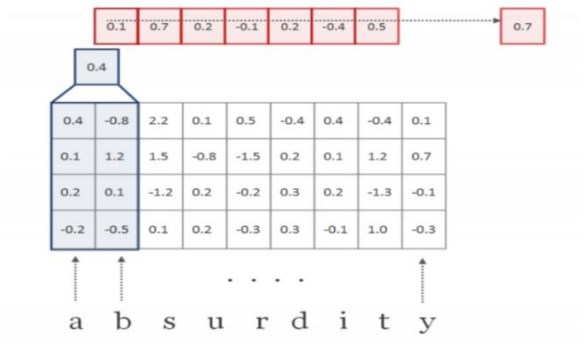
- Character embedding
- Highway network
- The Bi-directional attention flow mechanism

2.1. Character embedding

In NLP, the most common way to obtain features from tokens is to use an embedding layer. This layer transforms tokens indices into vectors of numbers. Therefore, these tokens are almost always words and not characters. Thus, it is a bit tricky to handle out of vocabulary (OOV) words. In most cases, we simply assign a random vector to these tokens, but sometimes it is not enough. Indeed, this could end up confusing the model.

It is why we need character embedding. This layer helps to better deal with OOV words. Indeed, there are only few characters in a language. By concatenating or processing these characters embedding, we end up with a word embedding that is more reliable than a random vector.

We now need to work at the character level (so we need to create a new vocabulary). Then, we use a simple embedding layer (like in the word embedding part) to obtain fixed-sized characters embedding. We then use a one-dimensional convolutional neural network followed by a max pooling: the kernels will slide over the word (i.e the characters embeddings), we end up with a new vector on which we apply a max pooling (to obtain fixed-sized features). The dimension of the word embedding is equal to the number of kernels used.



We can notice that we could use a LSTM layer instead of a 1D CNN to obtain fixed-sized embedding.

2.2. Highway network

A highway network is similar to a feed forward network. The difference is that only a fraction of the input is transformed according to a gate mechanism, the remaining fraction is permitted to pass through the network unchanged. The ratio of these fractions is managed by **t**, the *transform gate* and by (**1-t**), the *carry gate.* The value of **t** is calculated using a sigmoid function and is always between 0 and 1.

$$\mathbf{z} = \mathbf{t} \odot g(\mathbf{W}_H \mathbf{y} + \mathbf{b}_H) + (\mathbf{1} - \mathbf{t}) \odot \mathbf{y}$$

$$\mathbf{W}_H, \mathbf{b}_H : \text{Affine transformation}$$
$$\mathbf{t} = \sigma(\mathbf{W}_T \mathbf{y} + \mathbf{b}_T) : \textit{transform} \text{ gate}$$
$$\mathbf{1} - \mathbf{t} : \textit{carry} \text{ gate}$$

The role of this layer is to adjust the relative contribution from the word embedding and the character embedding. Indeed, for a word which has an embedding, we would like to grant more importance to this one instead of the one created by the 1D CNN. On the other hand, if the word does not have an embedding, we would like to grant more importance to the one created by the 1D CNN. The highway network allows us to do that.

2.3. The Bi-directional attention flow mechanism

Attention mechanism in the QA domain typically use attention to focus on a small portion of the context and summarize it with a fixed-sized vector. It is what we call a uni-directional attention (because the query is not processed). The BIDAF network introduces a multi-stage hierarchical process that represents the context at different levels of granularity and use bidirectional attention flow mechanism to obtain a query-aware context representation without early summarization.

Hence, the attention is computed in two-directions: from query to context as well as from context to query. Both attentions are derived from a shared similarity matrix S, between the contextual embeddings of the context and the query.

$\alpha(\mathbf{a}, \mathbf{b}) = w^T \times [\,\mathbf{a} \,;\, \mathbf{b} \,;\, \mathbf{a} \circ \mathbf{b}]$, where...

- $\mathbf{a}$ and $\mathbf{b}$ are the two input vectors
- $w^T \in \mathbb{R}^{6d}$ is a trainable weight vector
- $\circ$ is element wise multiplication
- $[\,;\,]$ is vector concatenation across row

In a nutshell, the similarity matrix computes the similarity between each feature of the query and each feature of the context and vice versa. (there is a need to tile both inputs matrices). The output matrix has as many rows as they are features in the context, and as many columns as they are features in the query.

Then, we can compute the context to query attention. That signifies to find which query words are most relevant to each context word. To do so, we just need to compute the row-wise softmax of the similarity matrix, which give for each context feature, the attention probability distribution. To finally obtain the new weighted query features, we need to do a dot product between this new attention probability distribution matrix and the query contextual embedding matrix.

To compute the query to context attention, we need this time to compute the max of the similarity matrix across each column. We then compute the softmax of the resulting vector to end up with the attention probability distribution. Finally, we obtain the new weighted context features vector by doing a dot product between this new attention probability distribution vector and the context contextual embedding matrix, we then need to tile the resulting vector as many times as they are features in the context to obtain the final query to context attention matrix.

The final step of this attention mechanism is the megamerge. The contextual embeddings and the attentions matrices are processed together to yield the matrix G. Each column of this matrix can be considered as the query aware representation of each context word.

$\beta(\mathbf{a}, \mathbf{b}, \mathbf{c}) = [\,\mathbf{a} \,;\, \mathbf{b} \,;\, \mathbf{a} \circ \mathbf{b} \,;\, \mathbf{a} \circ \mathbf{c}\,]$, where...

- $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are the three input vectors
- $\circ$ is element wise multiplication
- $[\,;\,]$ is vector concatenation across row

In the formula above:

- a is the context contextual embedding matrix
- b is the context to query attention matrix
- c is the query to context attention matrix

3. System description and implementation

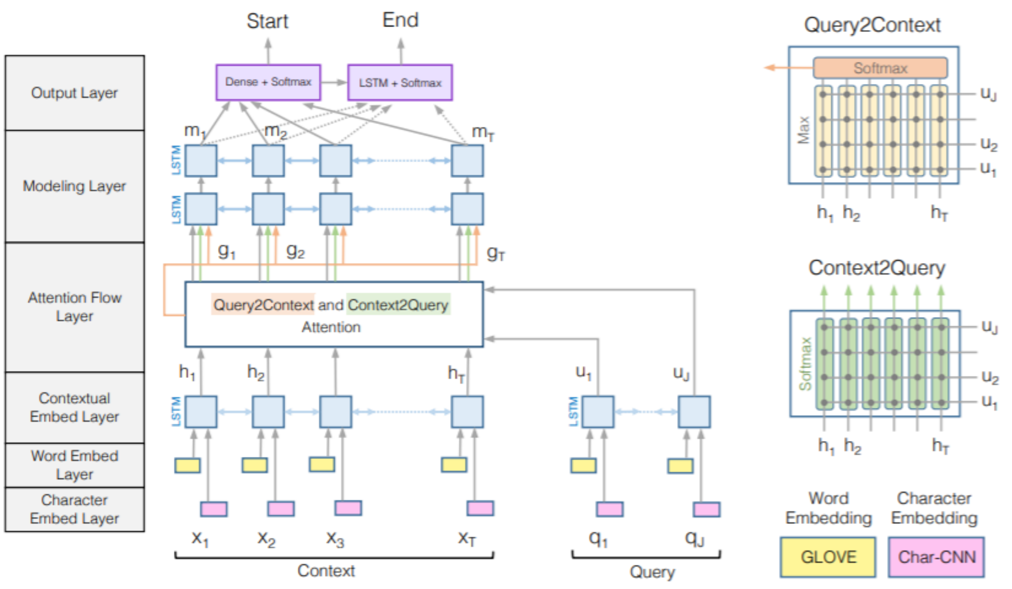


Figure 1: BiDirectional Attention Flow Model *(best viewed in color)*

In the original paper, it is state that the model consists of six layers (we can see them in the image above), but we can resume them in only 3:

3.1. Embeddings layers

The model has 3 embeddings layers (word-level, character-level and contextual). The purpose of these layers is to change the representation of words in the query and the context from strings into vectors of number.

3.1.1. Tokenization

The incoming query and its context are first tokenized, i.e these two longs strings are broken down into their consistuent words. We tokenized at both the word and character level (as it is state in the original paper). Finally, because we work with batches, we have to pad or truncate these sequences of tokens.

Our setup:

- At the word level:
  - Context max length: 400 tokens
  - Question max length: 25 tokens
- At the character level:
  - A word must contain 15 tokens

These numbers come from analysis of the training set

We used the word_tokenize function of the NLTK library to break sentences into bunch of words and the tensorflow Tokenizer to create the vocabulary, at both the word and character level. We have limited the size of the character vocabulary to the most 200 used tokens, all other tokens are treated as UNK tokens.

So, because we work at both the word and the character level, our model takes 4 inputs.

- Word level context matrix, **wc** of shape (batch_size, context_length)
- Word level query matrix, **wq** of shape (batch_size, question_length)
- Character level context matrix, **cc** of shape (batch_size, context_length, word_length)
- Character level question matrix, **cq** of shape (batch_size, question_lenght, word_length)

3.1.2. Word level embedding

wq and **wc** matrices are then subjected to the embedding process, where they are converted into vectors of numbers. These vectors capture the syntax and the semantics of the words, enabling us to perform various mathematical operations on them.

<u>Our setup:</u>

- We used pre-trained GloVe embeddings and we assigned random vectors to OOV words.
- The embedding size is 300

The output of this layer is:

- The matrix **wq** of shape (batch_size, question_length, embedding_size)
- The matrix **wc** of shape (batch_size, context_length, embedding_size)

3.1.3. Character level embedding

Only **cc** and **cq** are processed.

<u>Our setup:</u>

- The embedding size of the character embedding layer is 8
- We choose to use 300 kernels, in order to end up with an embedding size equal to the GloVe embedding size.
- The kernel size is 3x3

The output of this layer is:

- The matrix **cc** of shape (batch_size, context_length, embedding_size)
- The matrix **cq** of shape (batch_size, question_length, embedding_size)

3.1.4. Concatenation and highway network

The output of the word and character embedding layers are then concatenated for both the question and the context, and pass through a highway network.

So we end up with 2 matrices:

- **H**, representing the context, of shape (batch_size, context_length, 2* embedding_size)
- **U**, representing the query, of shape (batch_size, question_length, 2* embedding_size)

So far, we have two matrices, **H** and **U**, of embedding but it is still not enough. Indeed, these words representations do not consider the word's contextual meaning (the meaning derived from the words surroundings).

3.1.5. Contextual embedding

In order to end up with an embedding that can understand a word in its context, we used a bidirectional LSTM network.

Our setup:

- Output dimension of the LSTM is equal to the embedding size
- We used dropout with 0.2 rate

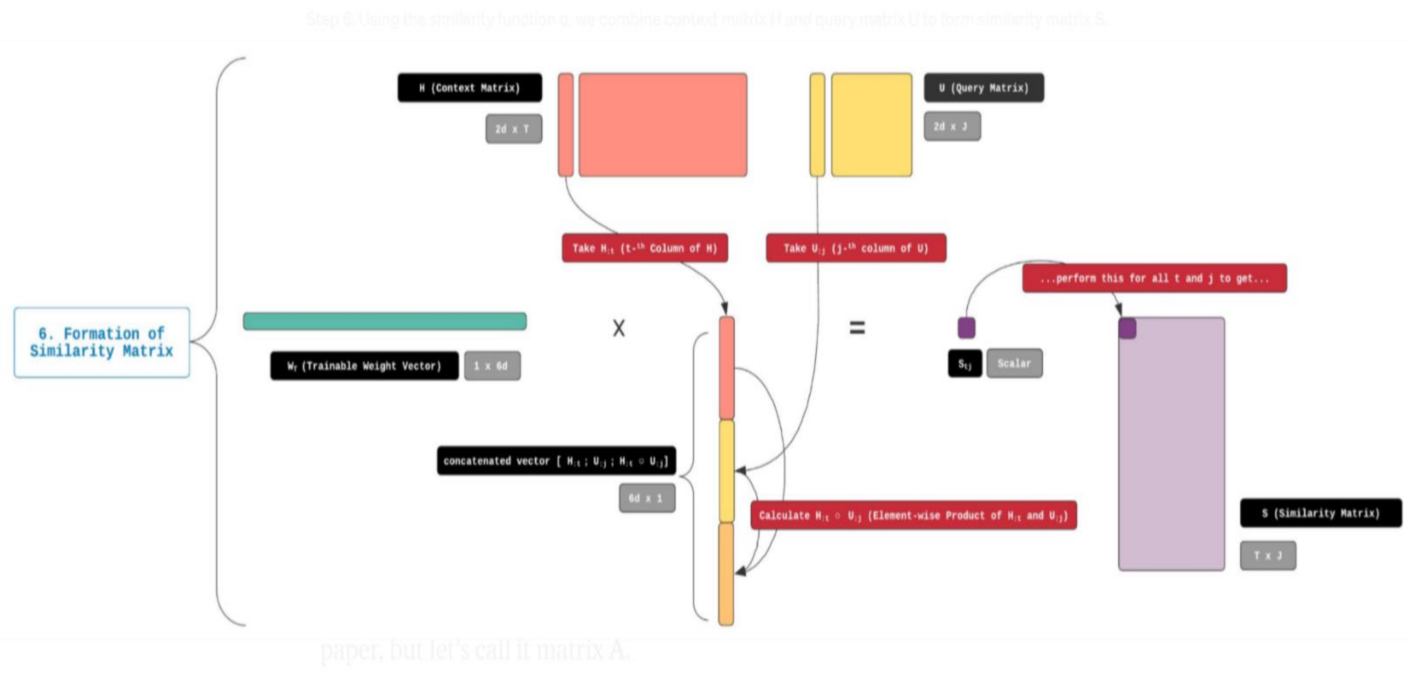
The shape of **H** and **U** is left unchanged.

3.2. Attention and Modelling layers

These layers use several matrix operations to fuse the information contained in the query and the context. The output of these steps is another representation of the context that contains information from the query.

3.2.1. Formation of the similarity matrix

We used the formula given in the Background part to compute this matrix. Here is a schema of the process:



H (Context Matrix)
2d x T
U (Query Matrix)
2d x J
Take H:t (t-th Column of H)
Take U:j (j-th column of U)
...perform this for all t and j to get...
6. Formation of Similarity Matrix
Wc (Trainable Weight Vector)
1 x 6d
x
=
Stj
Scalar
concatenated vector [H:t ; U:j ; H:t ∘ U:j]
6d x 1
Calculate H:t ∘ U:j (Element-wise Product of H:t and U:j)
S (Similarity Matrix)
T x J

The tricky point to form this matrix is to tile the inputs matrices

- The shape of the **similarity matrix** is (batch_size, context_length, question_length)

3.2.2. Context to Query Attention

Steps to follow in order to create this matrix have already been covered in the Background part

- The shape of this new matrix, **C2Q**, is (batch_size, context_length, 2 * embedding_size).

3.2.3. Query to context attention

Steps to follow in order to create this matrix have already been covered in the Background part. The tricky point is to remember that we have to tile the output vector context_length times in order to end up with the final matrix.

- The shape of this new matrix, **Q2C**, is (batch_size, context_length, 2 * embedding_size)

3.2.4. The Megamerge

Finally, the contextual embeddings and the attention vectors are combined to yield **G**.

- **G** is the concatenation of **H**, **C2Q**, **H** * **C2Q** and **H** * **Q2C** (see the Background part)
- The shape of **G** is (batch_size, context_length, 8 * embedding_size)

We can think of each row of **G** as a vector representation of a context word that is aware of the existence of the query and has incorporated relevant information from it.

3.2.5. Modelling layer

We then pass the matrix **G** through 2 bi-directional LSTM. The output of this layer captures the interaction among the context words conditioned on the query.

Our setup:

- Output dimension of the LSTM is equal to the embedding size
- We used dropout with 0.2 rate

The output of this layer is:

- The matrix **M** of shape (batch_size, context_length, 2 * embedding_size)

Each column of **M** is expected to contain contextual information about a context word with respect to the entire context paragraph and the query.

We then need to transform this matrix to a bunch of probability values. These probability values will be used to determine where the answer starts and ends.

3.3. Output layer

We obtain the probability distribution of the start index over the entire context by using this simple formula:

$$\mathbf{p}^1 = \mathrm{softmax}(\mathbf{w}_{(\mathbf{p}^1)}^\top[\mathbf{G};\mathbf{M}]),$$

So we concatenate **G** and **M**, we pass the result to a Dense layer and we apply the softmax function.

To obtain the end index, we pass **M** to another bi-directional LSTM (the output is **M2**) and we compute:

$$\mathbf{p}^2 = \mathrm{softmax}(\mathbf{w}_{(\mathbf{p}^2)}^\top[\mathbf{G};\mathbf{M}^2])$$

Our setup:

- After each dense layer, we used dropout with 0.2 rate

4. Experimental setup and results

We used the CategoricalCrossentropy loss of the Keras library, because the start and end indices have been one hot encoded (training set).

We used the Nadam (Adam with Nesterov momemtum) optimizer with a learning rate of 5e^{-4}.

We trained during 15 epochs. A validation set has been design (thanks to the train_test_split function of the scikit-learn library). The model weights are saved on google drive each time the validation loss reach a minimum.

Launch the preprocessing and the training of the model is simple:

- Run the bidaf_preprocessing notebook (you should maybe override some path and install some packages)
- Run the bidaf_model notebook (you should maybe override some path and install some packages)

Once the model is trained (work only with the SQUAD 2.0 train set), you can run the compute_answers.py script and give an unseen dataset, in the json format and formatted as the training set, as argument. A prediction file will be created, containing the id of each sample as well as the predicted answer.

You can then run the evaluate.py script which takes as argument the dataset (in json format) as well as the prediction file. It will output the F1 score.

Once the model has been trained, we created a prediction for the validation set and we run the evaluate.py script

```
{
  "exact": 51.07678171481077,
  "f1": 65.8105272912672,
  "total": 17413,
  "HasAns_exact": 51.07678171481077,
  "HasAns_f1": 65.8105272912672,
  "HasAns_total": 17413
}
```

We obtained a F1-score of 65.81%. In the SQUAD webpage, the score of the BIDAF model is around 70%.

5. Discussion

Our model yields a good F1-score, but it does not reach the score of the original model. We try to investigate some issues that could improve the performance

5.1. Pre-trained characters embeddings

The embedding layer (at the character level) is trainable. Instead, we could make this layer non-trainable by relying on GloVe to create character embeddings. For instance, to create the embedding for the character 'a', we could make an average of all words embeddings containing this letter.

More info: https://github.com/minimaxir/char-embeddings

5.2. Use GRU instead of LSTM

Training with GRU is generally faster than LSTM.

5.3. Use others merging techniques than concatenation

Embeddings from the word embedding layer and the characters embedding layers are concatenated. We can try to use another technique, like averaging or summing these vectors.