Published in Towards Data Science · Following

Meraldo Antonio · Follow
Aug 28, 2019 · 13 min read · Listen

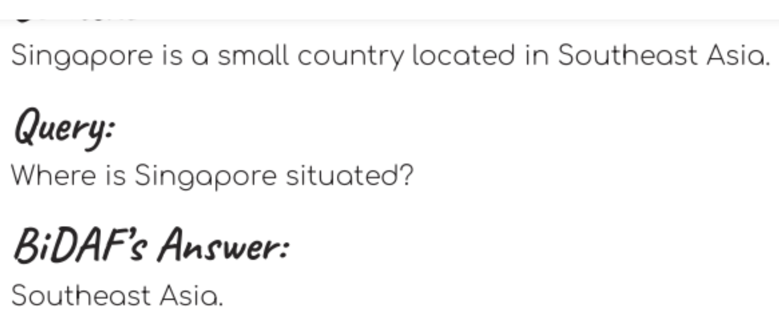
THE DEFINITIVE GUIDE TO BIDAF — PART 2 OF 4

Word Embedding, Character Embedding and Contextual Embedding in BiDAF — an Illustrated Guide

BiDAF is a popular machine learning model for Question and Answering tasks. This article illustrates how BiDAF uses three embedding mechanisms to convert words into their vector representations.

This article is the second in a series of four articles that aim to illustrate the working of **Bi-Directional Attention Flow (BiDAF),** a popular machine learning model for question and answering (Q&A).

To recap, BiDAF is an *closed-domain, extractive* Q&A model. This means that to be able to answer a **Query**, BiDAF needs to consult the an accompanying text that contains the information needed to answer the Query. This accompanying text is called the **Context.** BiDAF works by extracting a substring of the Context that best answers the query — this is what what we refer to as the **Answer** to the Query. I intentionally capitalize the words Query, Context and Answer to signal that I am using them in their specialized technical capacities.


Singapore is a small country located in Southeast Asia.

Query:
Where is Singapore situated?

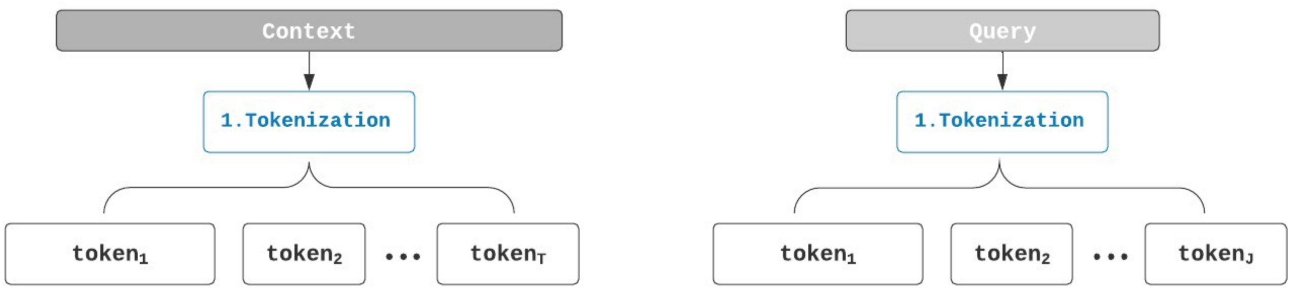
BiDAF's Answer:
Southeast Asia.

An example of **Context, Query,** and **Answer.** Notice how the Answer can be found Verbatim in the Context.

The first article in the series provided a high-level overview of BiDAF. In this article, we will focus on the first portion of the BiDAF architecture — the first thing that takes place when the model receives an incoming Query and its accompanying Context. To facilitate your learning, a glossary containing the mathematical notations involved in these steps is provided at the end. Let's dive in!

Step 1. Tokenization

In BiDAF, the incoming Query and its Context are first tokenized, i.e. these two long strings are broken down into their constituent words. In the BiDAF paper, the symbols **T** and **J** are used to denote the number of words in Context and Query, respectively. Here is a depiction of the tokenization:



Step 1. The incoming Query and its accompanying Context are tokenized into their constituent words.

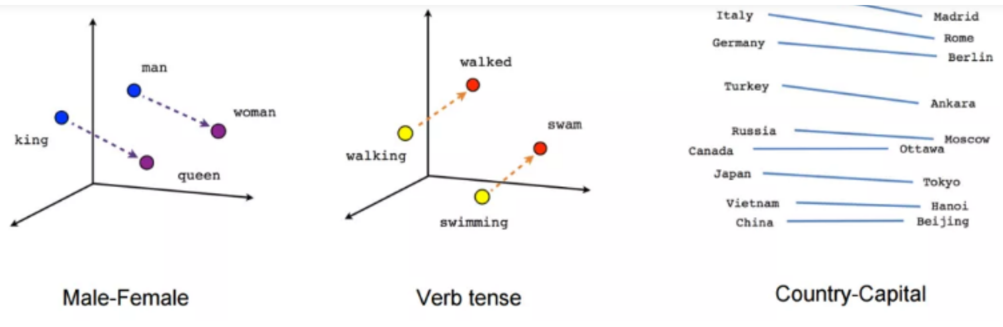
Step 2. Word Level Embedding

The resulting words are then subjected to the embedding process, where they are converted into vectors of numbers. These vectors capture the grammatical function (*syntax*) and the meaning (*semantics*) of the words, enabling us to perform various mathematical operations on them. In BiDAF, embedding is done on three levels of granularity: on the character, word and contextual levels. Let's now focus on the first embedding layer — the word embedding.

The word embedding algorithm used in the original BiDAF is GloVe. In this article, I will only give a brief overview of GloVe because there already exist several excellent resources that explain how the algorithm works. But if you are short on time, here is a very simplified summary of GloVe:

> GloVe is an unsupervised learning algorithm that uses co-occurrence frequencies of words in a corpus to generate the words' vector representations. These vector representations numerically represent various aspects of the words' meaning.

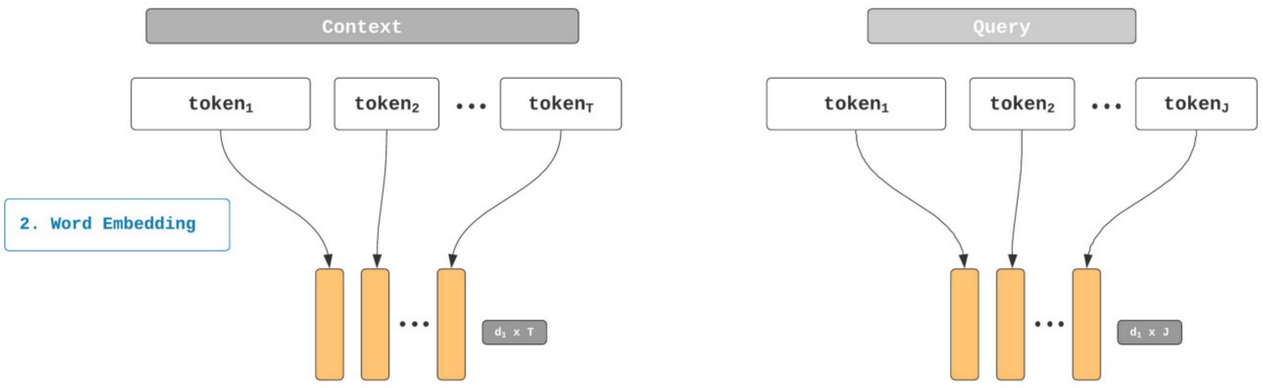
As the numbers that make up GloVe vectors encapsulate semantic and syntactic information about the words they represent, we can perform some cool stuff using these vectors! For instance, we can use subtraction of GloVe vectors to find word analogies, as illustrated below.



The distance between two GloVe vectors in space encapsulates a meaningful concept, such as gender, tense variation and country-capital relationship.

BiDAF uses pre-trained GloVe embeddings to get the vector representation of words in the Query and the Context. “Pre-trained” means that the GloVe representations used here have already been trained; their values are frozen and won’t be updated during training. Thus, you can think of BiDAF’s word embedding step as a simple dictionary lookup step where we substitute words (the “keys” of the GloVe “dictionary”) with vectors (the “values” of the “dictionary”).

The output of the word embedding step is two matrices — one for the Context and one for the Query. The lengths of these matrices equal the number of words in the Context and the Query (**T** for the Context matrix and **J** for the Query matrix). Meanwhile, their height , **d1**, is a preset value that is equal to the vector dimension from GloVe; this can either be 50, 100, 200 or 300. The figure below depicts the word embedding step for the Context:



Step 2. The word embedding step converts Context tokens into a **d1**-by-**T** matrix and Query tokens into a **d1**-by-**J** matrix

Step 3. Character Level Embedding

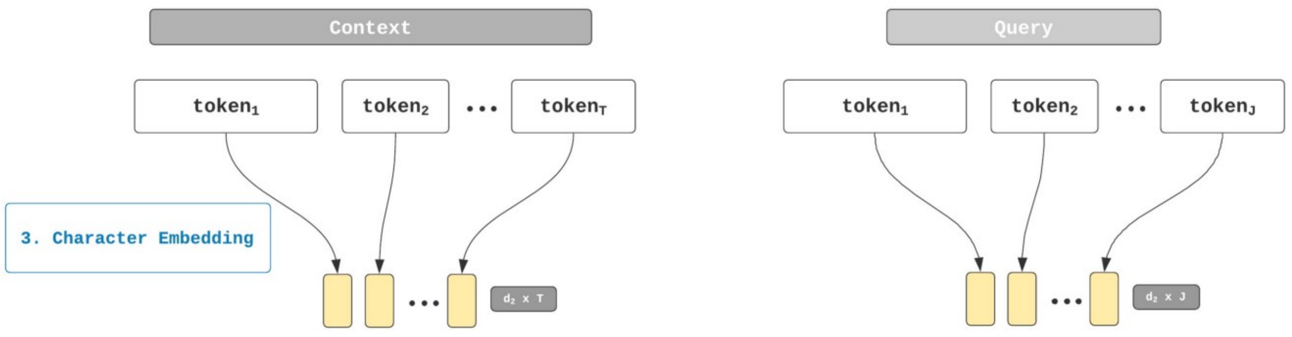
Okay, so with GloVe, we obtain the vector representations of most words. However, the GloVe representations are not enough for our purpose!

The pretrained GloVe “dictionary” is huge and contains millions of words; however, there will come a time where we encounter a word in our training set that is not present in GloVe’s vocabulary. Such a word is called an out-of-vocabulary (OOV) word. **GloVe deals with these OOV words by simply assigning them some random vector values.** If not remedied, this random assignment would end up confusing our model.

Therefore, we need another embedding mechanism that can handle OOV words. This is where the character level embedding comes in. **Character level embedding uses one-dimensional convolutional neural network (1D-CNN) to find numeric representation of words by looking at their character-level compositions.**

through a word, character by character. These scanners can focus on several characters at a time. As these scanners sweep along, they extract the information from the characters they are focusing on. At the end of these scanning process, information from different scanners are collected to form the representation of a word.

The output of the character embedding step is similar to the output of the word embedding step. We obtain two matrices, one for the Context and the other for the Query. The lengths of these matrices equal the number of words in the Context and in the Query — **T** and **J** — while their height depends on the number of *convolutional filters* used in 1D-CNN (to know what a "convolutional filter" is, do read the next section). The height is denoted as **d2** in the diagram. These two matrices will be concatenated with the matrices that we obtained from the word embedding step.



Step 3. The character embedding step converts Context tokens into a **d2**-by-**T** matrix and Query tokens into a **d2**-by-**J** matrix

Additional Details on 1D-CNN

The section above only presents a very conceptual overview of the workings of 1D-CNN. In this section, I will explain how 1D-CNN works in details. Strictly speaking, these details are not necessary to understand how BiDAF works; as such, feel free to jump ahead if you are short on time. However, if you are the type of person who can't sleep well without understanding every moving part of an algorithm you are learning about, this section is for you!

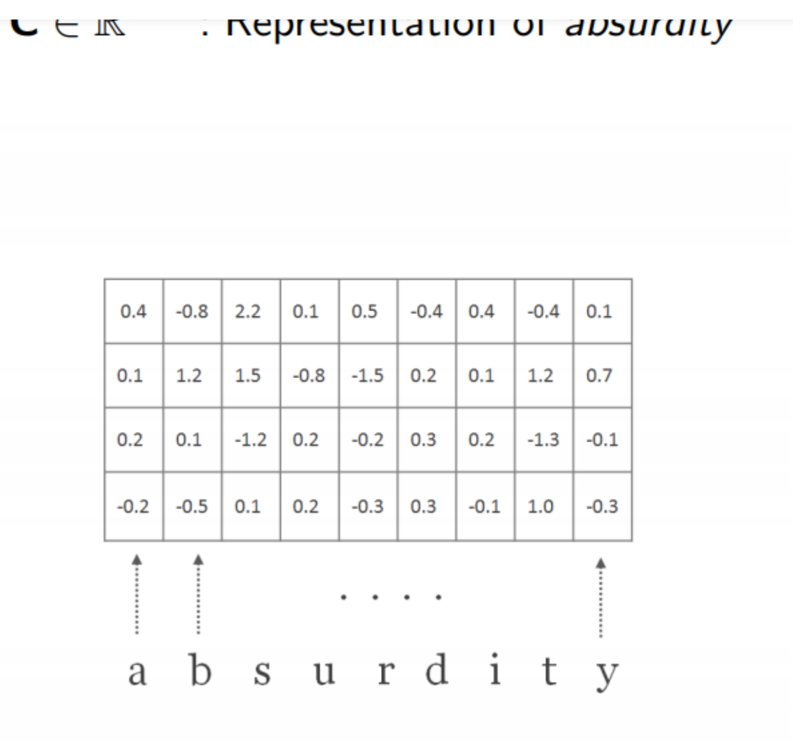
The idea that motivates the use of 1D-CNN is that not only words as a whole have meanings — word parts can carry meaning, too!

For example, if you know the meaning of the word "underestimate", you will understand the meaning of "misunderestimate", although the latter isn't actually a real word. Why? Because you know from your knowledge of the English language that the prefix "mis-" usually indicates the concept of "mistaken"; this allows you to deduce that "misunderestimate" refers to "mistakenly underestimate" something.

1D-CNN is an algorithm that mimics this human capability to understand word parts. **More broadly speaking, 1D-CNN is an algorithm capable of extracting information from shorter segments of a long input sequence.** This input sequence can be music, DNA, voice recording, weblogs, etc. In BiDAF, this "long input sequence" is words and the "shorter segments" are the letter combinations and morphemes that make up the words.

To understand how 1D-CNN works, let's look at the series of illustrations below, which are taken from slides by Yoon Kim et. al., a group from Harvard University.

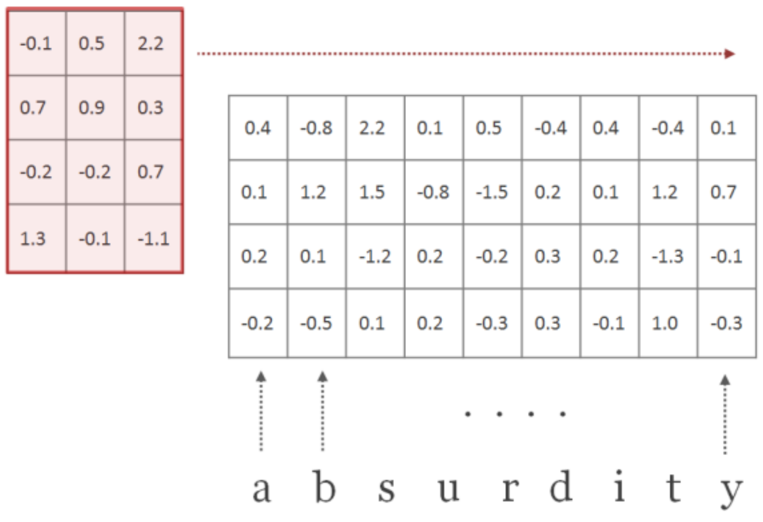
1. Let's say we want to apply 1D-CNN on the word "absurdity". The first thing we do is represent each character in that word as a vector of dimension **d**. These vectors are randomly initialized. Collectively, these vectors form a matrix **C.** ***d*** is the height of this



0.4	-0.8	2.2	0.1	0.5	-0.4	0.4	-0.4	0.1
0.1	1.2	1.5	-0.8	-1.5	0.2	0.1	1.2	0.7
0.2	0.1	-1.2	0.2	-0.2	0.3	0.2	-1.3	-0.1
-0.2	-0.5	0.1	0.2	-0.3	0.3	-0.1	1.0	-0.3

2. Next, we create a *convolutional filter* **H**. This convolutional filter (also known as "kernel") is a matrix with which we will "scan" the word. Its height, ***d***, is the same as the height of **C** but its width **w** is a number that is smaller than ***l***. The values within **H** are randomly initialized and will be adjusted during model training.

$$\mathbf{H} \in \mathbb{R}^{d \times w} : \text{Convolutional filter matrix of width } w = 3$$

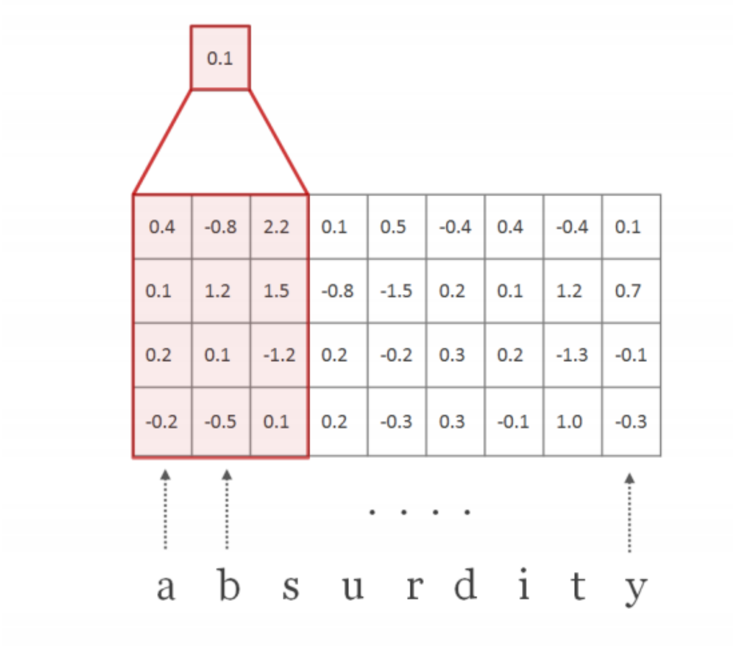


-0.1	0.5	2.2
0.7	0.9	0.3
-0.2	-0.2	0.7
1.3	-0.1	-1.1

0.4	-0.8	2.2	0.1	0.5	-0.4	0.4	-0.4	0.1
0.1	1.2	1.5	-0.8	-1.5	0.2	0.1	1.2	0.7
0.2	0.1	-1.2	0.2	-0.2	0.3	0.2	-1.3	-0.1
-0.2	-0.5	0.1	0.2	-0.3	0.3	-0.1	1.0	-0.3

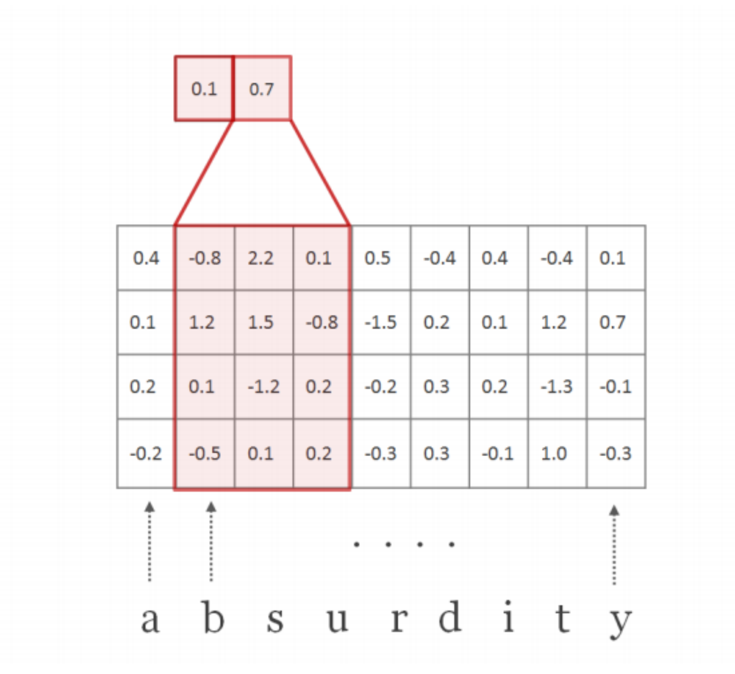
3. We overlay **H** on the leftmost corner of **C** and take an element-wise product of **H** and its projection on **C** (a fancy word to describe this process is taking a *Hadamard product* of **H** and its projection on **C**). This process outputs a matrix that has the same dimension as **H** — a ***d*** x ***l*** matrix. We then sum up all the numbers in this output matrix to get a scalar. In our example, the scalar is 0.1. This scalar is set as the first element of a new vector called **f.**

$$\mathbf{f}[1] = \langle \mathbf{C}[*, 1:3], \mathbf{H} \rangle$$

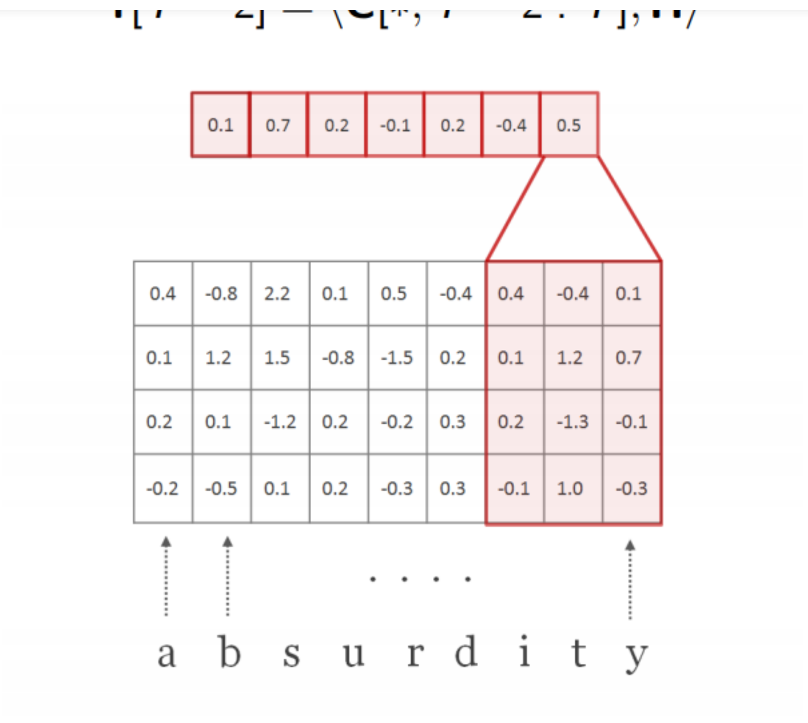


4. We then slide **H** one character to the right and perform the same operations (get the Hadamard product and sum up the numbers in the resulting matrix) to get another scalar, 0.7. This scalar is set as the second element of **f.**

$$\mathbf{f}[2] = \langle \mathbf{C}[*, 2:4], \mathbf{H} \rangle$$

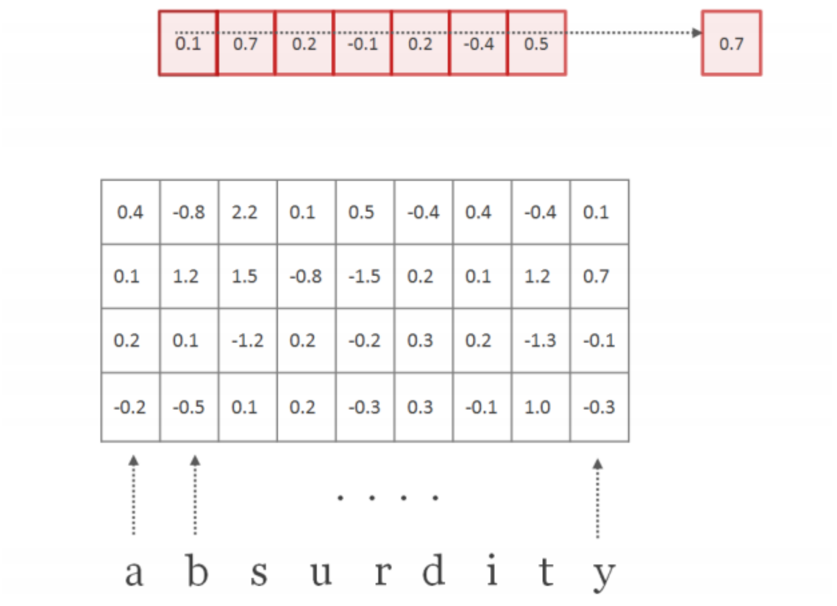


5. We repeat these operations character by character until we reach the end of the word. In each step, we add one more element to **f** and lengthen the vector until it reaches its maximum length which is **l - w+1.** The vector **f** is a numeric representation of the word "absurdity" obtained when we look at this word *three characters at a time*. One thing to note is that the values within the convolution filter **H** don't change as **H** slides through the word. In fancier terms, we call **H** "position invariant". **The position invariance of the convolutional filters enables us to capture the meaning of a certain letter combination no matter where in the word such combination appears.**

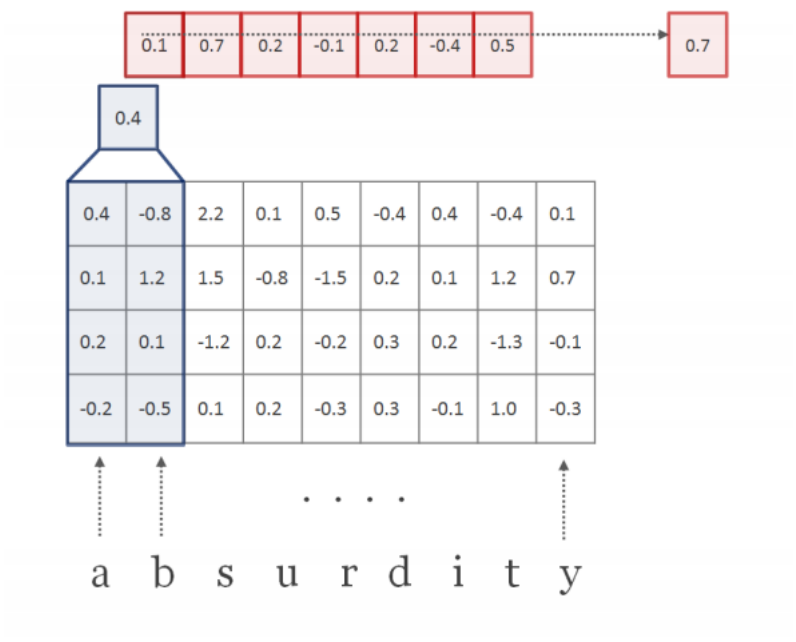


6. We record the maximum value in **f.** This maximum can be thought of as the "summary" of **f.** In our example, this number is 0.7. We shall refer to this number as the *"summary scalar"* of f. This process of taking a maximum value of the vector **f** is also referred to as *"max-pooling"*.

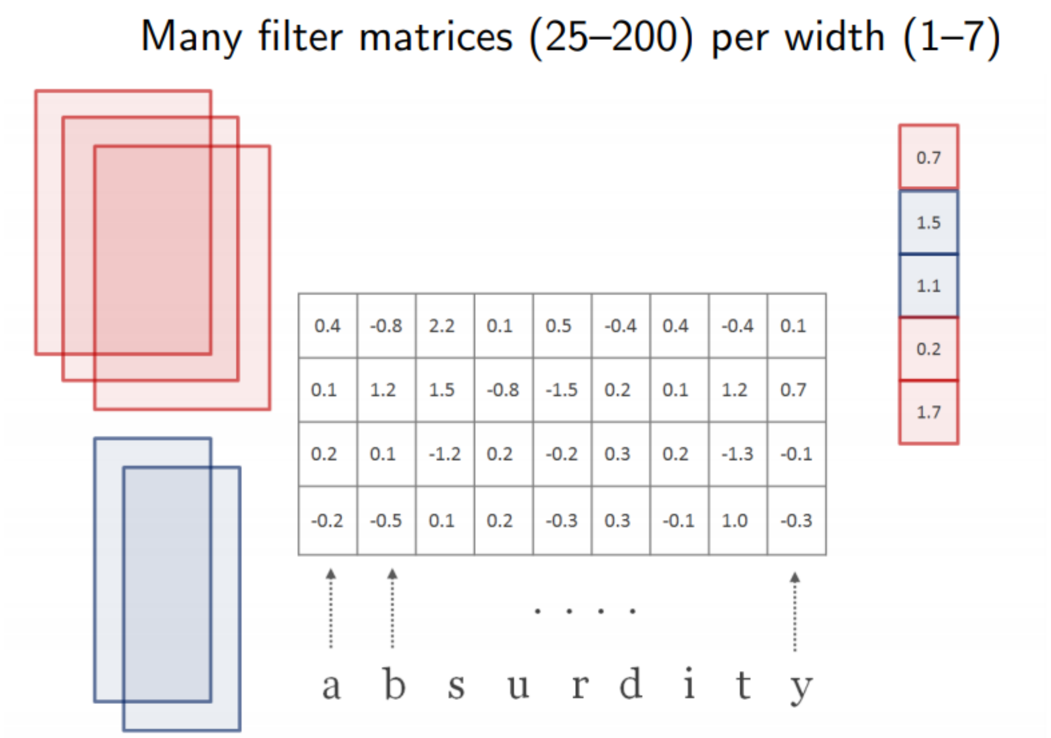
$$y[1] = \max_{i}\{\mathbf{f}[i]\}$$



7. We then repeat all of the above steps with yet another convolutional filter (yet another **H**!). This convolutional filter might have a different width. In our example below, our second **H**, denoted **H'**, has a width of 2. As with the first filter, we slide along **H'** across the word to get the vector **f** and then perform max-pooling on **f** (i.e. get its summary scalar).



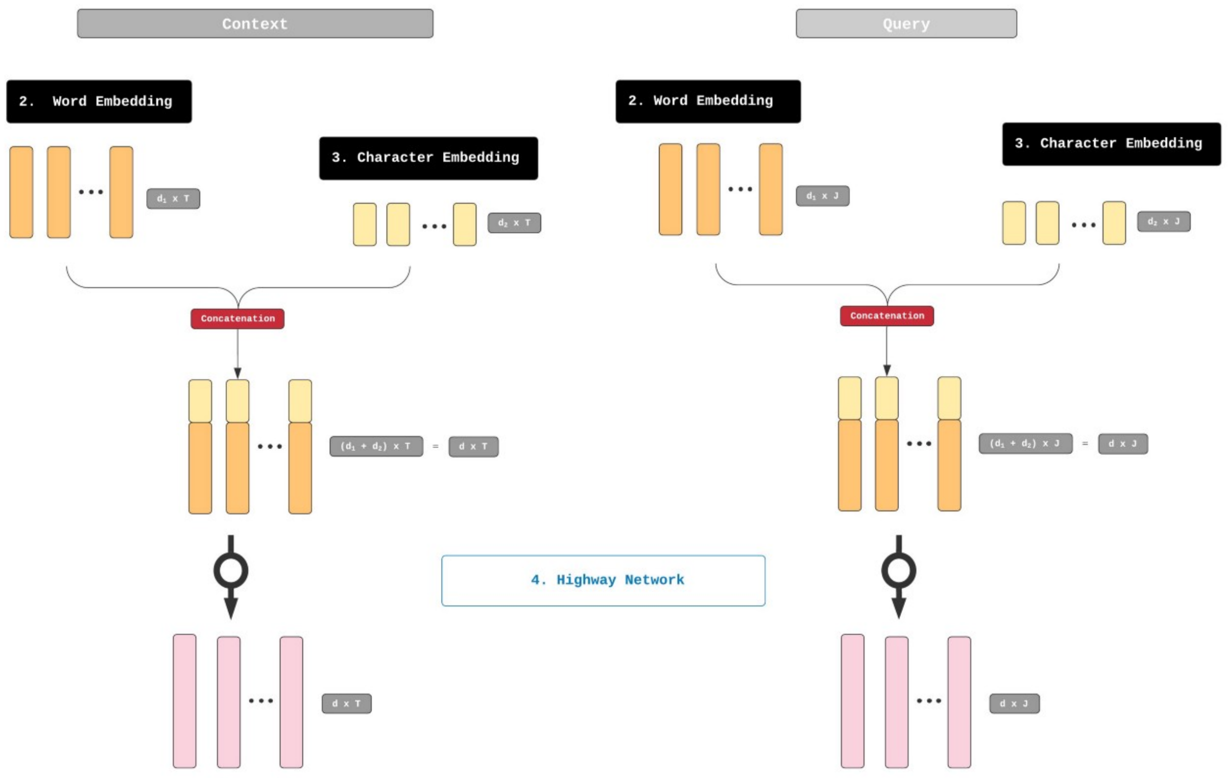
8. We repeat this scanning process several times with different convolutional filters, with each scanning process resulting in one summary scalar. **Finally, the summary scalars from these different scanning processes are collected to form the character embedding of the word.**



So that's it — now we've obtained a character-based representation of the word that can complement is word-based representation. That's the end of this little digression on 1D-CNN; now let's get back to talking about BiDAF.

Step 4. Highway Network

At this point, we have obtained two sets of vector representations for our words — one from the GloVe (word) embedding and the other from 1D-CNN (character) embedding. **The next step is to vertically concatenate these representations** matrix).



Step 4. The concatenated matrices from the word embedding and the character embedding steps are passed into a highway network

These matrices are then passed through a so-called highway network. A highway network is very similar to a feed forward neural network. You guys are probably familiar with feed forward neural network already. To remind you, if we insert an input vector **y** into a single layer of feed forward neural network, three things will happen before the output **z** is produced:

1. **y** will be multiplied with **W**, the weight matrix of the layer
2. A bias, **b**, will be added to **W*y**
3. A nonlinear function *g*, such as ReLU or Tanh will be applied to **W*y + b**

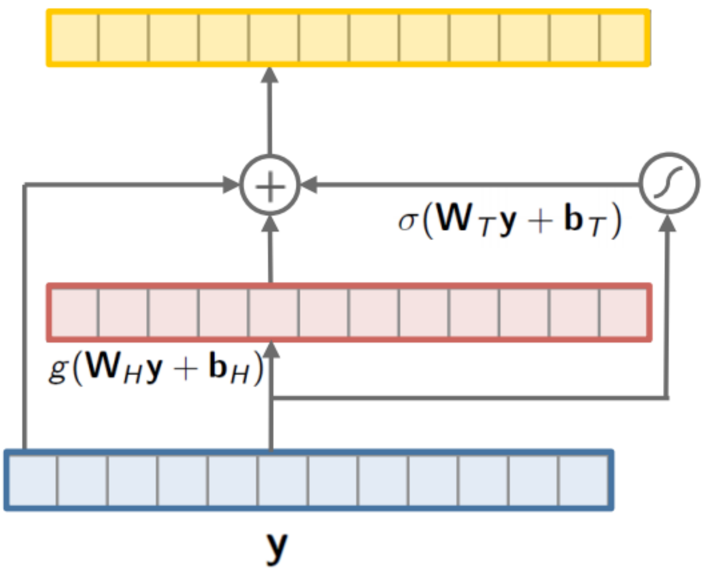
$$\mathbf{z} = g(\mathbf{W}\mathbf{y} + \mathbf{b})$$

In a highway network, only a fraction of the input will be subjected to the three aforementioned steps; the remaining fraction is permitted to pass through the network untransformed. 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. Now, our equation becomes as follows:

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

Upon exiting the network, the transformed fraction of the input is summed with its untransformed fraction.

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


$\sigma(\mathbf{W}_T\mathbf{y} + \mathbf{b}_T)$
$g(\mathbf{W}_H\mathbf{y} + \mathbf{b}_H)$
$\mathbf{y}$

The highway network's role is to adjust the relative contribution from the word embedding and the character embedding steps. The logic is that if we are dealing with an OOV word such as "misunderestimate", we would want to increase the relative importance of the word's 1D-CNN representation because we know that its GloVe representation is likely to be some random gibberish. On the other hand, when we are dealing with a common and unambiguous English word such as "table", we might want to have more equal contribution from GloVe and 1D-CNN.

The outputs of the highway network are again two matrices, one for the Context (a **d**-by-**T** matrix) and one for the Query (a **d**-by-**J** matrix). They represent the adjusted vector representations of words in the Query and the Context from word and character embedding steps.

Step 5. Contextual Embedding

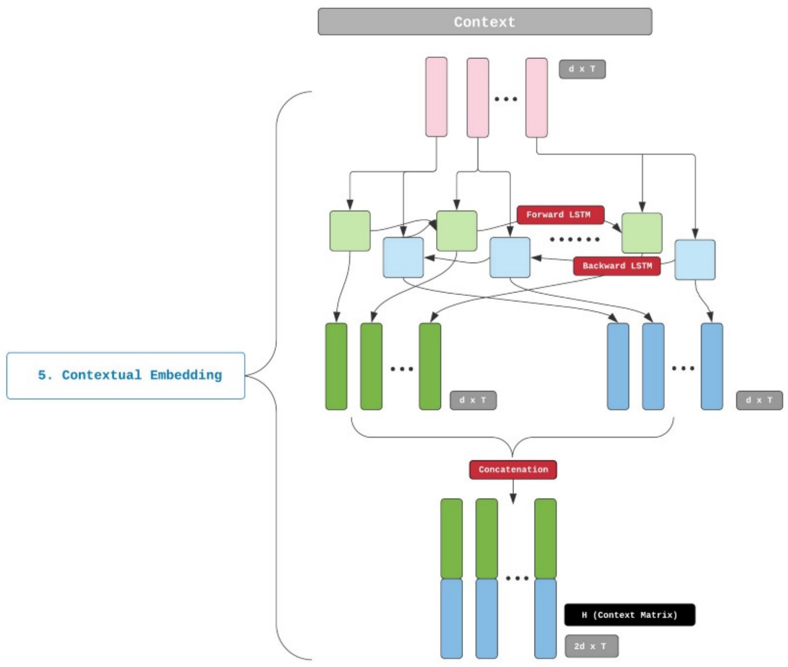
It turns out that these representations are still insufficient for our purpose! **The problem is that these word representations don't take into account the words' contextual meaning — the meaning derived from the words' surroundings.** When we rely on word and character embedding alone, a pair of homonyms such as the words "tear" (watery excretion from the eyes) and "tear" (rip apart) will be assigned the exact same vector representation although these are actually different words. This might confuse our model and reduce its accuracy.

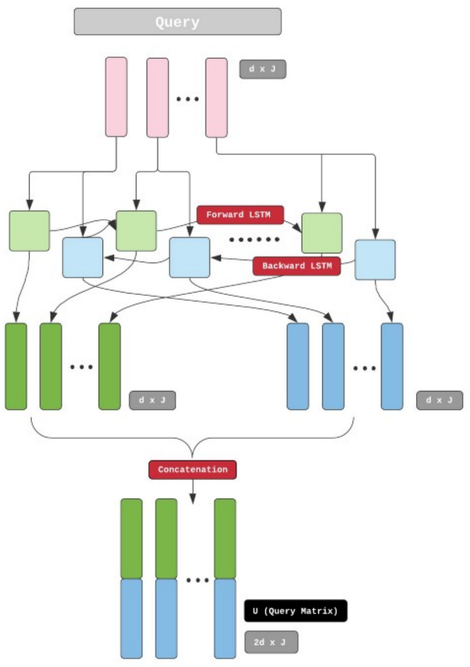
Thus, we need an embedding mechanism that can understand a word in its context. This is where the contextual embedding layer comes in. **The contextual embedding layer consists of Long-Short-Term-Memory (LSTM) sequences.** Here is a quick introduction to LSTM:

dependencies. When we enter an input sequence (such as a string of text) into a normal forward LSTM layer, the output sequence for each timestep will encode information from that timestep as well as past timesteps. In other words, the output embedding for each word will contain contextual information from words that came before it.

BiDAF employs a bidirectional-LSTM (bi-LSTM), which is composed of both forward- as well as backward-LSTM sequences. The combined output embeddings from the forward- and backward-LSTM simultaneously encode information from both past (backwards) and future (forward) states. Put differently, each word representation coming out of this layer now includes include contextual information about the surrounding phrases of the word.

The output of the contextual embedding step is two matrices — one from the Context and the other from the Query. The BiDAF paper refers to these matrices as **H** and **U**, respectively (terminologies alert — this matrix **H** is distinct from the convolutional matrix **H** mentioned earlier; it is an unfortunate coincidence that the two sources use the same symbol for two different concepts!). The Context matrix **H** is a **d**-by-**T** matrix while the Query matrix **U** is a **d**-by-**J** matrix.





Part 5. The contextual embedding step uses bi-LSTM to embed contextual information into the output matrices H and U.

So that's all there is to it about the embedding layers in BiDAF! Thanks to the contribution from the three embedding layers, the embedding outputs **H** and **U** carry within them the syntactic, semantic as well as contextual information from all words in the Query and the Context. We will use **H** and **U** in the next step — the attention step — in which we will fuse together the information from them. The attention step, which is the core technical innovation of BiDAF, will be the focus of the next article in the series — do check it out!

Glossary

- **Context :** the accompanying text to the Query that contains an answer to that Query
- **Query :** the question to which the model is supposed to give an answer
- **Answer :** a substring of the Context that contains information that can answer Query. This substring is to be extracted out by the

- **J** : the number of words/tokens in the Query
- **d1** : the dimension from the word embedding step (GloVe)
- **d2** : the dimension from the character embedding step
- **d** : the dimension of the matrices obtained by vertically concatenating word and character embeddings. **d** is equal to **d1** + **d2.**
- **H** : the Context matrix outputted by the contextual embedding step. **H** has a dimension of **2d**-by-**T**
- **U** : the Query matrix outputted by the contextual embedding step. **U** has a dimension of **2d**-by-**J**

References

[1] Bi-Directional Attention Flow for Machine Comprehension (Minjoon Seo *et. al*, 2017)

[2] Character-Aware Neural Language Models (Yoon Kim *et. al*, 2015)

If you have any questions/comments about the article or would like to reach out to me, feel free to do so either through LinkedIn or via email at meraldo.antonio AT gmail DOT com.