Short Intro to BERT

Notes for DTY - CentraleSupélec

Yassine Ouali

Introduction

Before the development of the Transformer architecture, and the subsequent NLP models that are based on such a structure, *e.g.*, BERT, RoBERTa, GPT. Start-of-the-art NLP models were based on Recurrent Neural Networks (RNNs), such as the LSTMs. A recurrent neural network takes as input a sequence of words, one at a time, and gradually builds up an understanding of the sentence. Unlike standard neural networks, the input to a recurrent neural network is both the current word as well as a hidden state (*i.e.*, a vector) from processing the previous word. This recursiveness complicates things a little bit, both when training the model, since we need to send the training signal back through multiple steps, and computationally, since we can not parallelize the computation given that to compute the output at given step, we need to wait for the previous ones to finish.

The Transformer comes to the rescue, with an architecture based fully on the attention mechanism without any recurrence, making the training of NLP models much faster. Additionally, and similar to computer vision models, *e.g.*, ResNet, where we can adapt a pre-trained model on some down-stream task (*e.g.*, document classification) without significantly altering the original model.

1 Preliminaries

One of the main benifits of BERT is its usage in Transfer Learning. Transfer Learning consists of two main step: pre-training and fine-tuning so first, before going into the inner workings of BERT, we start by clearing the notions of pre-training and fine-tuning

1.1 Pre-training

Before applying BERT to any given task, like sentence classification, we start by a pre-training task. This step is quite important, in which we teach the model about the structure of language, acquiring the necessary information that will be used across different tasks.

BERT pre-training consists of two tasks: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). Let's take for example this input with two sentences A and B:

BERT is a powerful model. It obtains new state-of-the-art results on eleven NLP tasks.

First, we mask a randomly given work, replacing it with [MASK], so the input becomes:

BERT is a powerful model. It obtains new state-of-the-art [MASK] on eleven NLP tasks.

And here's what BERT is supposed to do:

- MLM: Predict the masked out word. (Correct answer is "results").
- NSP: Does the sentence B (*i.e.*, BERT obtains new state-of-the-art [MASK] on eleven NLP tasks) comes immediately after sentence A (*i.e.*, BERT is a powerful model), or is the two switched or not related? (Correct answer is that they are consecutive).

MLM and NSP can be see as *fake tasks*, *i.e.*, BERT was trained to perform these two tasks purely as a way to force it to develop a high-level understanding of language. Once the pre-training is done, we discard the portion of the model that's specific to these tasks, *i.e.*, the last project layer going from

the embeddings to the vocabulary, and replace it with something more useful for out downstream task we are interested in, *e.g.*, classification layers.
What's great about MLM and NSP is that we don't need any labeled data (*i.e.*, we call such tasks *self-supervised* tasks, since the labels are generated directly from the data itself), and we can simply train with raw text, such as all of Wikipedia, or a crawl of the internet. However, there is a catch, the tasks are challenging, requiring BERT to develop strong language understanding skills. Which requires a very large amount of data, and subsequently a large amount of compute to train a model with enough capacity to solve the pre-training tasks. Fortunately, the team behind BERT trained it to perform MLM and NSP on a massive text dataset with many TPUs, and then made the trained model publicly available for us to use. So we can use them out-of-the box without any pre-training.

1.2 Fine-Tuning

As stated earlier, the end-goal is not to learn to predict missing words (MLM) and tell you whether two sentences are consecutive or not (NSP), but to use the pre-trained BERT for something more useful, *e.g.*, question answering or sentence classification. BERT is a 12-layer neural network that processes text. MLM and NSP each add a small, single-layer classifier to the output of BERT to perform their respective tasks. The magic of BERT is that we can replace this final classifier with our own task-specific model, and leverage all of BERT's learned knowledge.

For example, if we want to apply BERT to a particular text classification task, we would take the pre-trained BERT model, add an untrained layer on top, and train the new, combined model on your own dataset and task. This final training step is referred to as *fine-tuning*, because the amount of training required to adapt BERT to our own task is very small compared to what it took to pre-train it. The illustration below shows how the same pre-trained BERT model can be used to perform Sentiment Analysis or Named Entity Recognition, with the only difference being the final layer of the model.

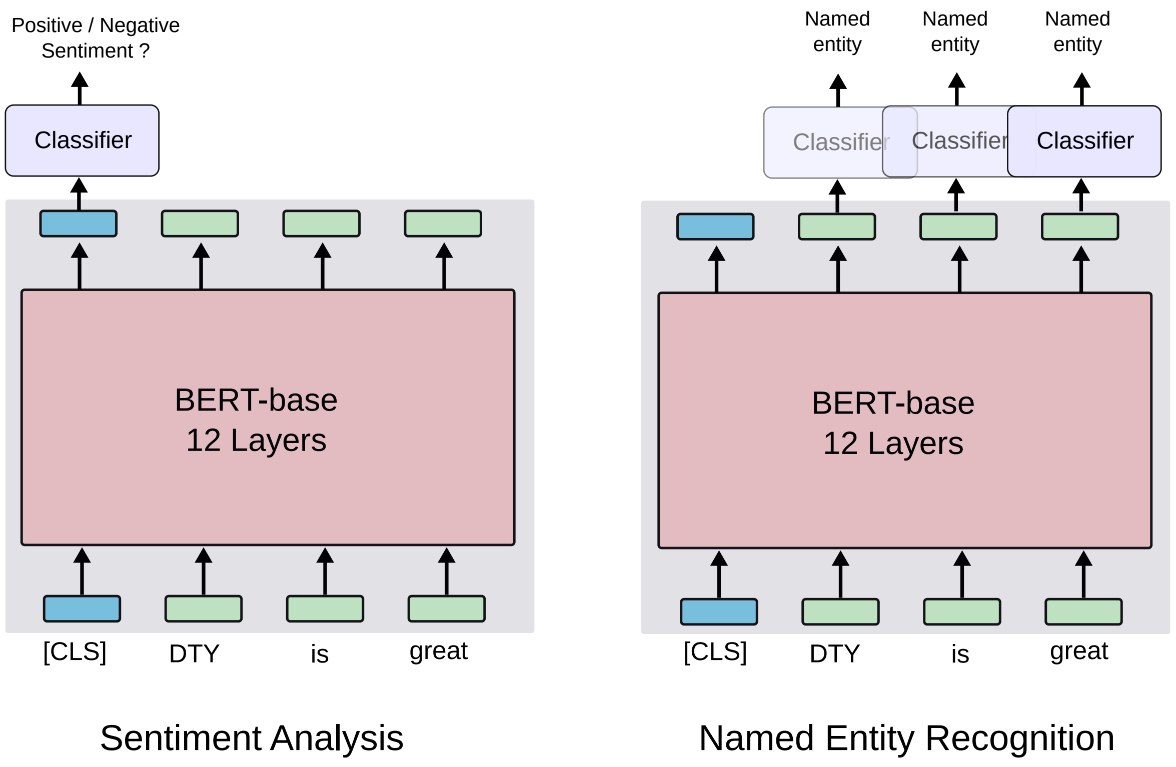

Figure 1: Examples of downstream tasks BERT can be used on.

1.3 Transfer Learning

This technique of adding a small task-specific component to the end of a large pre-trained model, and fine-tuning the result, is known as Transfer Learning. So, why use transfer learning rather than train a specific deep learning model (*e.g.*, a CNN, BiLSTM, etc.) that is well suited for the specific NLP task we need?

Quicker Development. First, the pre-trained BERT model weights already encode a lot of information about the language used during pre-training. As a result, it takes much less time to train our fine-tuned model, it is as if we have already trained the bottom layers of our network extensively and only need to gently tune them while using their output as features for our classification task. In fact, the authors recommend only 2-4 epochs for fine-tuning BERT on a specific NLP task, compared to the hundreds of GPU hours needed to train the original BERT model.

Less Data. In addition and perhaps just as important, because of the pre-trained weights this method allows us to fine-tune on our task with a much smaller dataset than would be required in a model that is built from scratch. A major drawback of NLP models built from scratch is that we often need a prohibitively large dataset in order to train the network to reasonable accuracy, meaning a lot of time and energy had to be put into dataset creation. By fine-tuning BERT, we are now able to get away with training a model to good performance on a much smaller amount of training data.

Better Results. Finally, this simple fine-tuning procedure (typically adding one fully-connected layer on top of BERT and training for a few epochs) was shown to achieve state of the art results with minimal task-specific adjustments for a wide variety of tasks, *e.g.*, classification, language inference, semantic similarity, question answering, etc. Rather than implementing custom and sometimes-obscure architectures shown to work well on a specific task, simply fine-tuning BERT is shown to be a better (or at least equal) alternative.

1.4 Tokenizer

Before diving into the inner working of BERT, it is necessary to have a clear understanding of how BERT processes the input text and what exactly BERT outputs. BERT has a way of processing, and attempting to make sense of, any text that we give it, even if the text contains words that never appeared during BERT's pre-training, thanks to the tokenizer used. BERT's performance will likely suffer if the text is highly domain-specific, such as a biomedical research paper, and if we fed BERT French, it wouldn't work at all, but thanks to BERT's tokenizer, we will at least be able to input the text in the correct format.

But, what is a tokenizer? A tokenizer is responsible of preprocessing the inputs for out model. Since the model doesn't know anything about the structure of language, we need to break the inputs into chunks, or tokens, before feeding it into the model, which will then be trained to learn the appropriate representation of each token. So, given a sequence of words, tokenization is the task of chopping it up into pieces, called tokens. Here, tokens can be either words, characters, or subwords. Hence, tokenization can be broadly classified into 3 types, *i.e.*, word, character, and subword tokenization.

Subword Tokenization splits the piece of text into subwords. For example, words like lower can be segmented as low-er, smartest as smart-est, and so on. Transformed based models, *e.g.*, BERT, rely on subword tokenization algorithms (*e.g.*, Word Piece, Bit Pair Encoding, Sentence Piece) for preparing vocabulary.

BERT tokenizer A pre-trained BERT model comes with its own built-in BERT Tokenizer which will take our raw text string and break it into tokens that BERT can process. The reason BERT provides its own tokenizer is that it has its own fixed vocabulary of tokens, of about 30k tokens, with an embedding associated with each one. 30k tokens is fairly small compared to pre-trained word embedding like fastText with a vocabulary of 200k. But as it turns out, a smaller vocabulary can be a very good thing, since a larger vocabulary will contain rarer words, which are, by definition, harder to train on given their rarity. However, a drawback of having such a fixed and small vocabulary is that we cannot introduce any new words, because all of BERT's impressive knowledge about language is based on the embeddings that it learned during pre-training.

Luckily, thanks to the way the tokenizer works, this might not be as a big of problem, since it is capable of handling words that aren't in the vocabulary. Additionally, there are many pre-trained BERT models out there besides the ones provided by the authors, and there may be one available that better fits our application domain (for example, SciBERT for biomedical literature).

Handling Out-of-Vocab Words BERT's tokenizer uses a *WordPiece* model to tokenize the text. BERT has a fixed vocabulary size of about 30k tokens in total. Somewhere between 60-80% of these are whole words, and the rest are subwords or *word pieces*. If a word isn't in BERT's vocabulary, then it simply breaks it down into subwords. And if it's missing a subword, then it can break it down further into individual characters. So no matter the word we are presented, WordPiece tokenizer will be able to represent it using the 30,000 existing tokens. When an such unknown word is split into subwords and/or characters, each of these pieces becomes a separate token, represented by its own embedding. For example if we want to encode *Centralesupélec*, but the word is not in the vocabulary if will be tokenized as follows:

Centralesupélec → *[central, ##es, ##up, ##ele, ##c]*

And as we see, the word *Centralesupélec* was devided in a whole word: *central*, subwords: *es, up, ele* and a character *c*. Note that subwords and character are always proceeded with ## as a way of differentiating them.

2 BERT: A black box view

When we apply BERT to a piece of text, the tokenizer will split the text into tokens, and then look up the embeddings for these tokens, and those are the actual inputs into BERT. Each embedding is a vector with 768 features. If we view the BERT model simply as a *black box*, then what it does is take in a set of embeddings on its input (*i.e.*, all at once, not in sequence), and then outputs an enhanced version of each of those representations, which are of the same dimension as the input, *i.e.*, 768-d vectors.

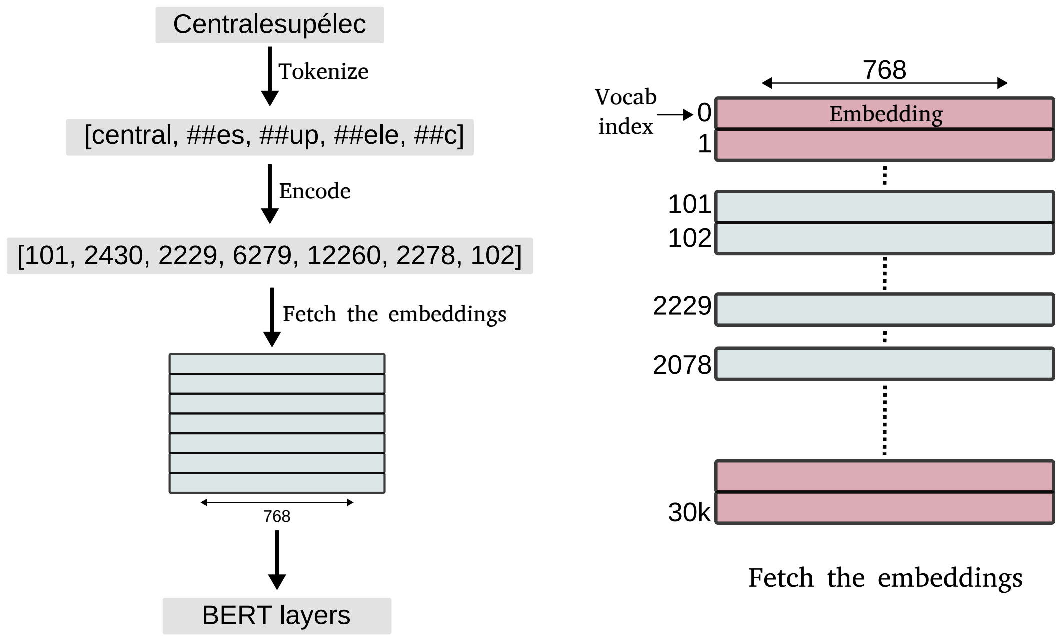

Figure 2: The preprocessing steps.

How are the outputs are enhanced? BERT will process all of the words in the input text in parallel, but it's helpful to look at the model from the perspective of how it creates the enhanced and contextualized output features for a single word within the input sentence. In order to enhance the features for "like" in the illustration below, we look at the input embedding for "like", but we also feed in all of that word's *context*, *i.e.*, the embeddings for all of the other words in the sentence. To be more precise, this context aggregation is done by repeating many weighted averaging steps with self-attention mechanism, where the representations of each work are updated as a weighted average of all the inputs (*i.e.*, all other words in the sequence), where the weights are themselves learned.

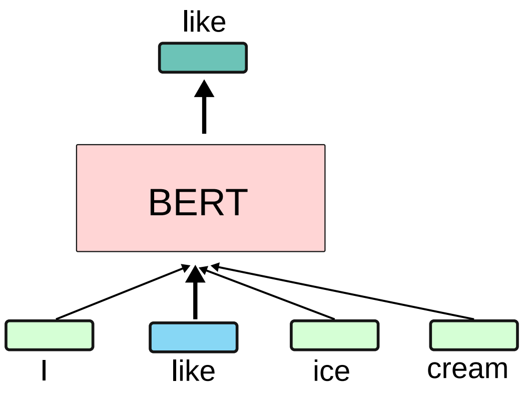

Figure 3: A illustration of context aggregation for a single token.

Out of the box, the features on the output are optimized for the two tasks that BERT was pre-trained on. Those two pre-training tasks (*i.e.*, MLM and NSP) aren't intended to be practical or interesting, though, so fine-tuning the model to adapt the features for our task is important.

2.1 Parallelization

Because BERT processes each word independently, it's possible to parallelize the process and run all of the input words through BERT at once. Where the process of taking in a set of features and enhancing them is actually repeated 12 times in BERT. Each of BERT's 12 layers takes in the output embeddings from the previous layer, and further enhances them. The bellow illustration reflects this by that the aggregation of context information is done for each input token in parallel. And the webbing shows that all words are incorporated when processing each word. This process of taking in a set of embeddings and enhancing them is actually repeated 12 times in BERT.

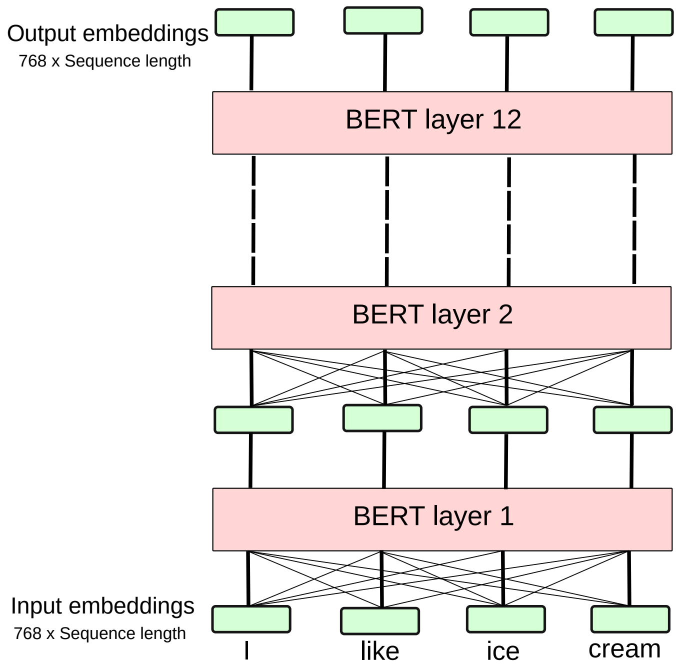

Figure 4: Illustration of 12 layers of BERT-base.

Until now, we are processing a single input sequence, so given a sentence, we first tokenize it, encode it and then fetch the embedding corresponding to each token and pass the embedding throught the 12 layers of BERT. So for an input sequence of length M, the output will be of size is $M \times 768$, since each input token is represented by a 768-d embedding vector. But why not process many sequence at a time. So for N sequences, where each sequence is of the same length M, we want to get the outputs for all of them at one, *i.e.*, $N \times M \times 768$, in this case, we can pass this directly to our model and we will get the correct outputs. However, it is very unlikely that all the sequences have the have length. The solutions is simple, pad the small sequences to the correct length. For example for two sequences of length M_1 and M_2 with $M_1 > M_2$, we need to pad the second sequence to the first one, and this is done by adding zeros to it up to length bM_1.

2.2 Encoding Word Position

In the previous illustrations, the words and their embeddings were placed in the order that they occur in a sentence. This arrangement is actually somewhat misleading, the words could have been reordered in any manner, and the output feature will remain the same because BERT doesn't actually have any explicit knowledge of the word order. In recurrent neural networks, word order is fundamental to the architecture, because the words are fed in one at a time in sequence, and processing each word requires both the current word and some results from the previous word. Since BERT process all input tokens in parallel, and given how important the order of the words in a sentence is, we need another way to encode the position of each word.

Transformers solve this by incorporating the positions of the words in the embeddings themselves. BERT for instance has a set of 512 Positional Encoding (PE) embedding vectors, each vector is of the same size as the embedding for the words, *i.e.*, 768-d vectors. But why 512 PE vectors? the maximum sequence length BERT accepts if 512, so at worst, we'll need 512 ways to differentiate between the input tokens. So, in order to incorporate the position i of a word x in a sequence, we first fetch i-th PE vector of size 768, and then we simply add it to the corresponding embedding if word x. See figure bellow on how the embedding for work "like" is computed. Note that for BERT, there is also an additional embedding for the sentences level to differentiate between sentence A and sentence B.

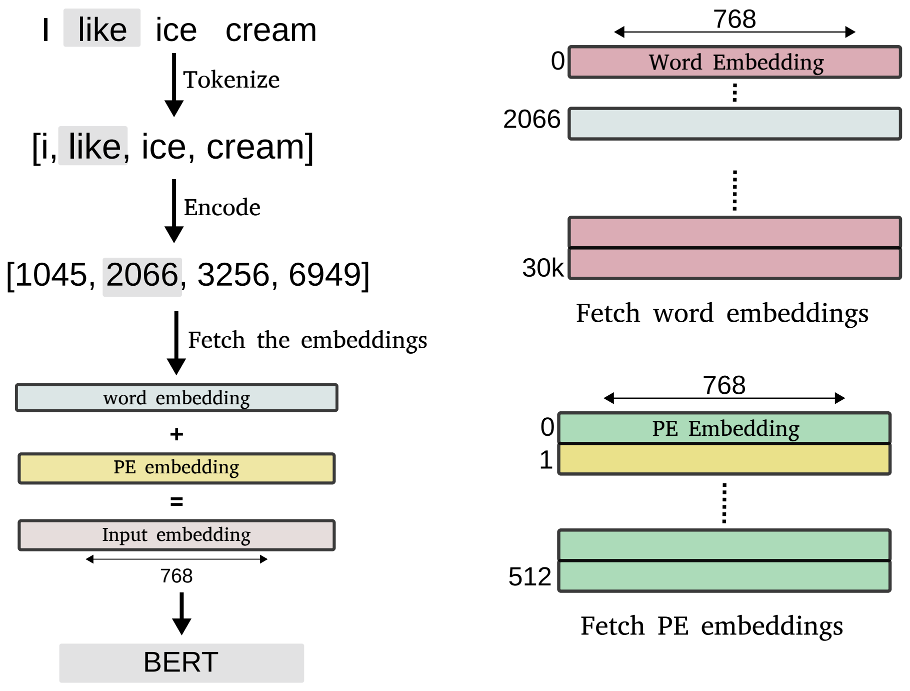

Figure 5: The way the word position is incorporated into the word embeddings.

The PE vectors create an upper limit on the input sequence length to BERT. BERT can receive at most 512 input tokens, since this is the number of positions that it has learned PE vectors for. 512 tokens is a large amount of text, though, and if we go over this limit, the most common workaround is to simply truncate the input to 512 tokens.

2.3 Special Tokens

There are still a few details of the input and output format which we haven't covered yet:

- The special [CLS] token for sentence-level classification tasks.
- The [SEP] token and the Segment Embeddings for handling two-sentence tasks.

[CLS] token. Whenever we feed in a piece of text to BERT, the first token, which we've omitted until now, will always be the special [CLS] token. Then, when applying BERT to a classification task, we always use the final embedding for the [CLS] token as the input to our classifier, and ignore the individual token embeddings. See the Figure 1 for an illustration. But why is this token works for sentences classification? When pre-training BERT, one of the two pre-training tasks is Next Sentence Prediction (NSP), and it's a text classification task. To predict the label, we add a simple linear classifier to the end of BERT, and only the final embedding for the [CLS] token is fed into it.

[SEP] token. BERT also has the ability to receive two separate pieces of text at once, and to make a prediction based on both pieces of text. A practical use of this might be de-duplication. For example, in 2017 Quora released a *Question Pairs* dataset, where the task is to detect whether two questions posted on Quora are essentially the same. BERT includes two mechanisms which help it to differentiate the two pieces of text on its input. The first is another special token, "[SEP]", which is placed in between the two pieces of text. The second is the addition of two Segment Embeddings, one for sentence A and one for sentence B. Along with the Positional Encoding vectors, the two segment embeddings are simply added to the word embeddings before they are sent into the model.

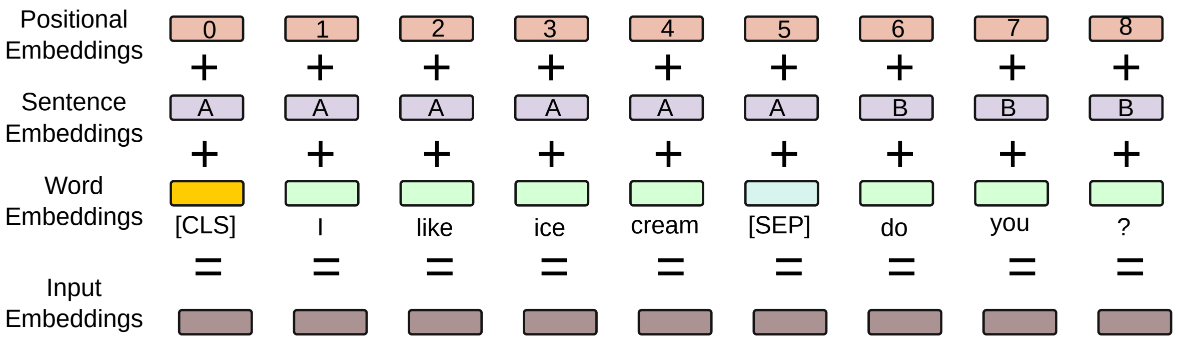

Figure 6: The final embeddings.

To the lab

Now, with this high-level overview, you have the necessary knowledge to use BERT for you application of choice. Granted, we did not go through the inner working of the Transformer architecture. But that it is not the end goal for this brief introduction, and if you wish, you can always check the original paper and other recourses that present the inner working of BERT.