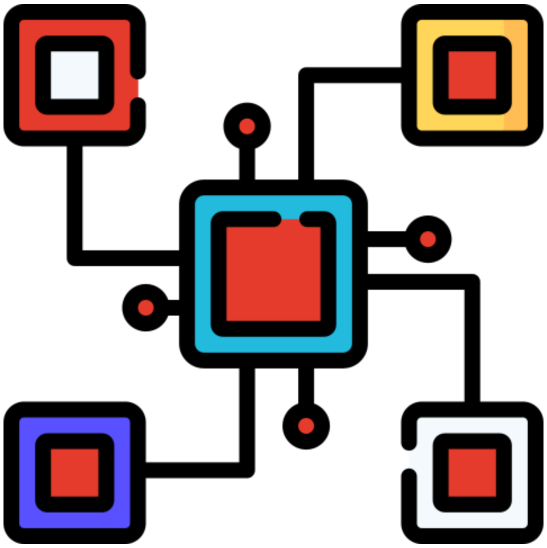
The Inner Workings

- of -

BERT



By Chris McCormick

It is my earnest desire that the information in this book be as correct as possible; however, I cannot make any guarantees. This is an evolving book about an evolving technology in an evolving field--there are going to be mistakes! So here's my disclaimer: The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Edition: v1.0.0

Contents

Introduction **4**

i. Organization 6

ii. Pre-Requisites 9

iii. Transfer Learning 11

Pre-Training 11

Fine-Tuning 13

Transfer Learning 15

Part 1 - BERT Basics & Applications **18**

1. BERT Input & Output Format 19

1.1 BERT Tokenizer 19

1.2. Black Box View 23

1.3. Single Embedding View 25

Variable Sentence Length 26

1.4. Parallelization 27

Twelve Layers 28

1.5. Encoding Word Position 31

1.6. Special Tokens 33

2. Ways to Apply BERT 34

2.1. Token Classification 35

2.2. Text Classification 35

Example Code 39

2.3. Text-Pair Classification 40

2.4. What BERT Can’t Do 43

Part 2 - BERT Architecture 47

3. Self-Attention 47

3.1. Weighted Average of Embeddings 48

3.2. Calculating the Attention Weights 52

3.2. Projecting the Embeddings 53

4. Feed-Forward Neural Network 59

5. Multi-Headed Attention 62

5.1. Re-Combining Attention Head Embeddings 63

Part 3 - Pre-Training Tasks 66

6. BERT’s Pre-Training 67

6.1. Training Set 67

6.2. Masked Language Model (MLM) 69

6.3. Next Sentence Prediction (NSP) 74

7. Training Task Improvements 77

7.1. Whole-Word Masking 78

7.2. Replacing NSP 79

7.3. Plausible Substitutions 80

Appendix 81

A.1. Revision History 81

Introduction

BERT (Bidirectional Encoder Representations from Transformers) was first introduced towards the end of 2018, and quickly became a hot topic in NLP. This is because:

1. It demonstrates a very sophisticated knowledge of language, achieving *human-level performance* on certain tasks.
2. It can be applied to a variety of tasks.
3. It offers the benefits of **pre-training + fine-tuning**: BERT has been *pre-trained* by Google on a very large text corpus, and you can leverage its understanding of language by taking the pre-trained model and *fine-tuning* it on your own application (classification, entity recognition, question answering, etc.). This can allow you to achieve **highly accurate** results on your task with minimal design work.

You can imagine, though, how it's easy to misconstrue the above qualities as "BERT has a human-level understanding of language and you can throw it at any problem and expect to get great results". We're not there yet!

The goal of this eBook is to give you a detailed understanding of how the BERT model works, as well as an honest look at what it can and cannot do.

i. Organization

I've divided this book into three parts.

Part 1 - BERT Basics & Applications

The input and output format of the BERT model--how it takes in your text, and what it ultimately gives you back--is likely different than any other machine learning model that you are previously familiar with. In Part 1, we'll start by treating the BERT model as a kind of "black box", and we'll focus first on understanding the format of the input and output.

One of the most exciting aspects of BERT is that it can be applied to so many different NLP problems. Once you feel comfortable with how BERT handles text, we can explore some of these different applications. And, lest you think that BERT is the ultimate solution to all NLP problems, we'll also look at the applications that BERT *can't* solve (due to the details of BERT's input and output!).

Part 2 - BERT Architecture

In Part 2, we'll open up that black box, and work through the details of BERT's architecture and implementation. The topics in this part will be the most difficult to understand, but I’ve done my best to illustrate the concepts and explain them plainly and with examples.

Part 3 - Pre-Training Tasks

You’ve probably heard of some related models which have come out since BERT. Names like RoBERTa, XLNet, and ALBERT, which have all managed to achieve higher benchmark scores than the original BERT.

I’ve definitely felt some dread over these developments--”Oh no! Is BERT outdated now? Do I need to learn an entirely new architecture?” Definitely not. You’ll be surprised to learn that these advances still use *the same basic architecture* discussed in part 2 of this book! A common theme to these newer models is that they have all tried to modify and improve upon *the tasks used to pre-train BERT*.

In Part 3, we’ll look first at BERT’s original pre-training tasks, and then I’ll also cover some of the various modifications to these tasks proposed in more recent papers.

Appendices

To make this book a less-daunting read, I’ve moved some of the topics which, either, you may already be familiar with, or you may not be particularly interested in, to the appendix.

ii. Pre-Requisites

*You **do not** need to understand Recurrence!*

Prior to BERT (and the "Transformer" architecture it's built from), the most sophisticated NLP models relied on "Recurrent" Neural Networks (RNNs), such as the LSTM architecture.

A recurrent neural network looks at the words in a sentence 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 some "hidden state" (i.e., a vector) from processing the previous word. This recursiveness complicates the model (think about backprop--you have to go backwards through the layers *and* backwards through time…), and makes RNNs their own (substantial) area of study.

The good news about learning BERT and the Transformer architecture is that the *Transformer does away with recurrence*!

> *The 2017 paper by a team at Google which originally introduced the Transformer architecture (on which BERT is built) was called "Attention is all you need…" meaning, you don't need recurrence.*

If you are familiar with the fundamentals of Machine Learning, then you can understand BERT. Specifically, you should already understand the following:

- Basics of Neural Networks
    - For example, it's enough if you've seen a neural network applied to the MNIST handwritten digit dataset. (Or - For example, what you'd learn in an introductory Machine Learning course).
    - I'll be using terms like **vectors** and **weight matrices**, and we'll be performing **dot products** and **matrix multiplication**.
- The concept of a "word embedding"
    - Just the *concept* of a word embedding is enough--you *don't* need any detailed knowledge of algorithms like word2vec or GloVe.
    - If you're unfamiliar with word embeddings, I cover the basics at the beginning of this YouTube video.

iii. Transfer Learning

Before we can dig into how BERT works, it's critical that we're clear on the notions of **Pre-Training** and **Fine-Tuning** (which, together form the technique of **Transfer Learning**).

Pre-Training

If you read the BERT paper, or many of the BERT tutorials out on the web, you'll find that the key contribution of BERT are these two (rather strange) tasks named **Masked Language Model (MLM)** and **Next Sentence Prediction (NSP)**.

Here is a simple example of these two tasks. For example text, I'm using a passage from the intro of the BERT paper itself:

> **"BERT is conceptually ~~simple~~ and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks..."**

And here's what BERT is supposed to do:

1. *MLM* - Predict the crossed out word. (Correct answer is "simple").

2. *NSP* - Was **sentence B** found immediately after **sentence A**, or from somewhere else? (Correct answer is that they are consecutive).

If you are not familiar with the body of work that led up to BERT, then seeing so much emphasis on the above two tasks can be very confusing. When I first set out to understand BERT, I had a couple responses to these tasks.

First, they don't seem particularly *useful*--why is BERT so exciting if these are the only two things that it's capable of doing?!

Second, I can't imagine that BERT can actually perform these tasks with high accuracy--it seems too easy to come up with examples of both of these where even a human would fail!

Here's what I was missing. MLM and NSP are just BERT's "Fake Tasks", or **Pre-Training Tasks**. BERT was trained to perform these two tasks purely as a way to force it to develop a sophisticated understanding of language. Once the pre-training is done, we actually *discard* the (tiny) portion of the model that's specific to these tasks, and replace it with something more useful (more on that in a moment…).

What's great about MLM and NSP is that:

1. We don't need any labeled data. We can simply *train with raw text*, such as all of Wikipedia, or a crawl of the internet.

2. The tasks are *challenging*, requiring BERT to develop strong language understanding skills.

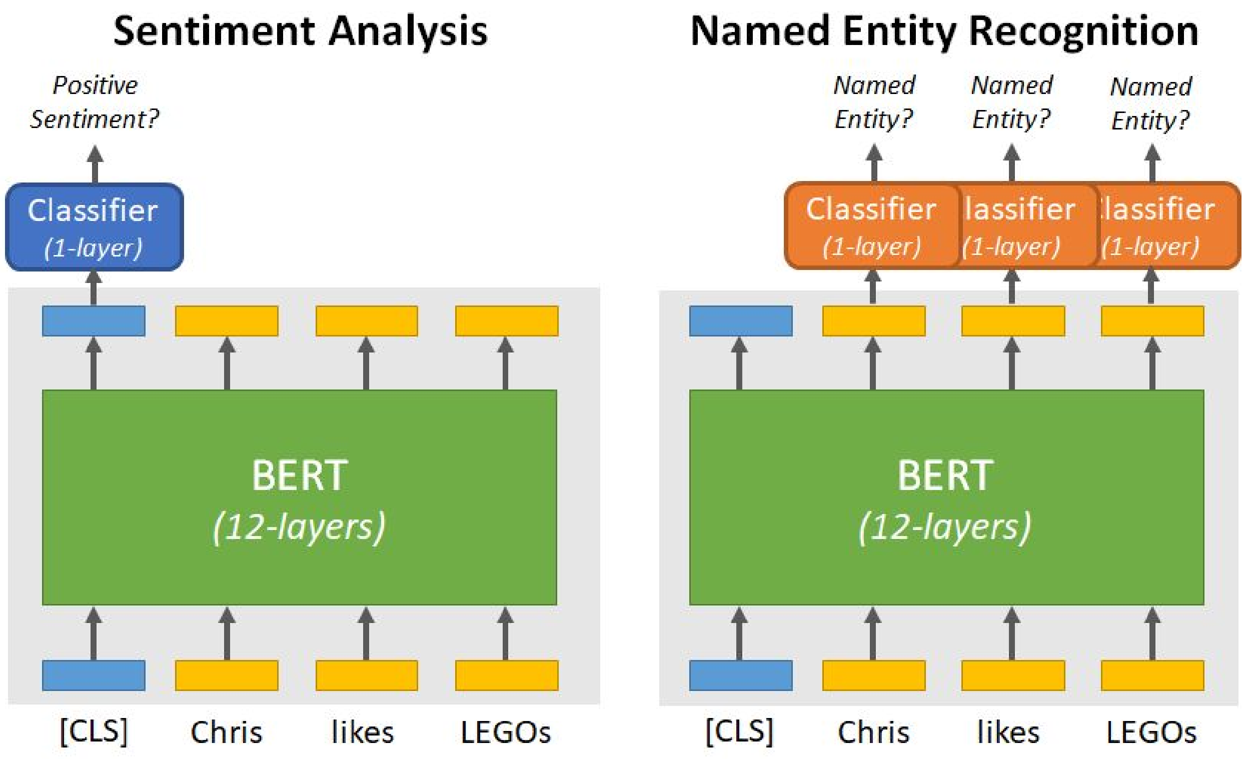
The team at Google trained BERT to perform MLM and NSP on a massive text dataset, using *tens to hundreds of thousands of dollars* worth of compute time on TPUs, and then made the trained model publicly available for us to use. Thanks, Google!

Fine-Tuning

Of course, no matter how intelligent BERT is, it would be much less interesting if all it could do were predict missing words (MLM) and tell you whether two sentences are consecutive or not (NSP).

The main portion of BERT is a very large 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 you can replace this final classifier with your own task-specific model, and leverage all of BERT's knowledge!

The below illustration shows how the same pre-trained BERT model (inside the grey boxes) can be used to perform Sentiment Analysis or Named Entity Recognition, with the only difference being the final layer of the model.



For example, if you want to apply BERT to a particular text classification task, you would take the pre-trained BERT model, add an untrained layer of neurons on the end, 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 your own task is very small compared to what it took Google to *pre-train* BERT.

Side Note: Though the fine-tuning step is small compared to what Google had to do, it’s worth noting that fine-tuning BERT is still a fairly computationally expensive task…

Transfer Learning

This technique of adding a small task-specifc 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 (a CNN, BiLSTM, etc.) that is well suited for the specific NLP task you need?

1. **Quicker Development**
    - First, the pre-trained BERT model weights already encode a lot of information about our language. 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 of training for fine-tuning BERT on a specific NLP task (compared to the hundreds of GPU hours needed to train the original BERT model or an LSTM from scratch!).
2. **Less Data**
    - In addition and perhaps just as important, because of the pre-trained weights this method allows us to fine-tune our

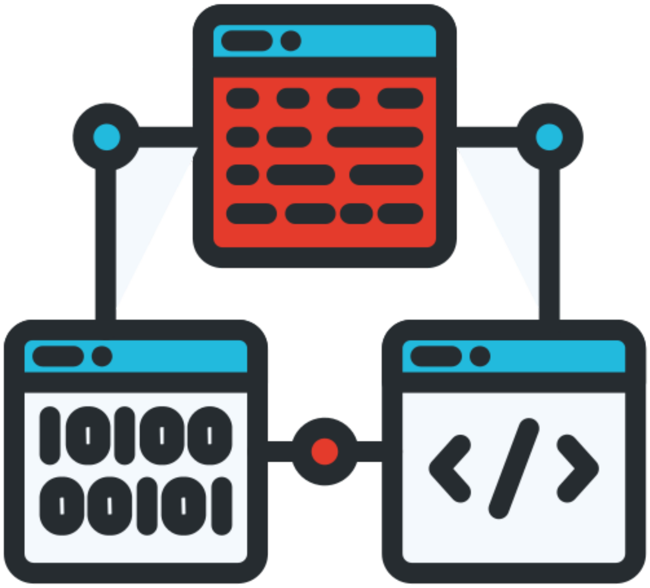
task on a much smaller dataset than would be required in a model that is built from scratch. A major drawback of NLP models built from scratch is that we often need a prohibitively large dataset in order to train our network to reasonable accuracy, meaning a lot of time and energy had to be put into 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.

3. **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: 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.

This shift to transfer learning parallels the same shift that took place in computer vision a few years ago. Creating a good deep learning network for computer vision tasks can take millions of parameters and be very expensive to train. Researchers discovered that deep networks learn hierarchical feature representations (simple features like edges at the lowest layers with gradually more complex features at higher layers). Rather than training a new network from scratch each time, the lower layers of a trained network with generalized image features could be copied and transferred for use in another network with a different task. It soon became common practice to download a pre-trained deep network and quickly retrain it for the new task or add additional layers on top - vastly preferable to the expensive process of training a network from scratch.

For many, the introduction of BERT (a deep, pre-trained language model) signals the same shift to transfer learning in NLP that computer vision saw.

Part 1 - BERT Basics & Applications



1. BERT Input & Output Format

Before diving into the details of BERT's internal architecture, it's good to have a clear picture of how BERT ingests your text and what exactly BERT gives you back in return.

BERT can actually be run against any arbitrary text (even if the vocabulary is different than what BERT was trained on!), and it all begins with the tokenizer.

1.1 BERT Tokenizer

BERT Takes Raw Text

A pre-trained BERT model comes with its own built-in **BERT Tokenizer** which will take your raw text string and break it into tokens that BERT can ingest.

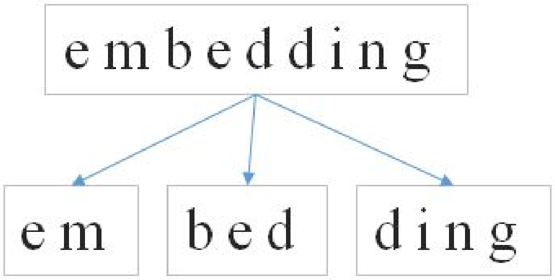
The reason BERT provides its own tokenizer is that it has its own **fixed vocabulary** of tokens, with an embedding associated with each of these. A drawback of BERT's pre-training is that you cannot use your own embeddings--all of BERT's impressive knowledge about language is based on the embeddings that it was pre-trained with. If you gave it a different vocabulary, it wouldn't know what to do with it!

How big of a problem is this? *Less than you might think*. First, BERT has an intelligent way of handling words that aren't in its vocabulary, which we'll cover here. Second, there are many pre-trained BERT models out there besides Google's original, and there may be one available that better fits your application domain (for example, BioBERT for medical text and SciBERT for scientific literature).

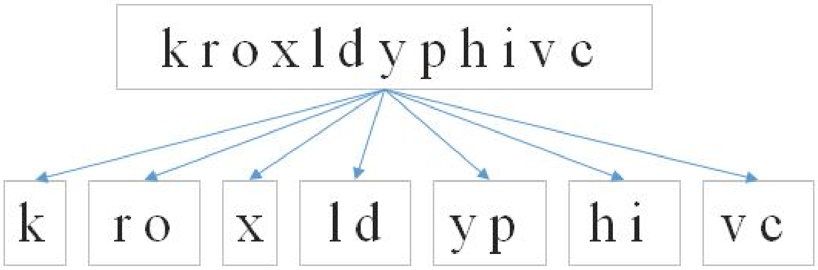
Handling Out-of-Vocab Words

BERT's tokenizer uses a "WordPiece" model to tokenize your text. The original BERT model has a fixed vocabulary size of about 30,000 tokens 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 how strange of a word you throw at it, BERT has a way of representing it.

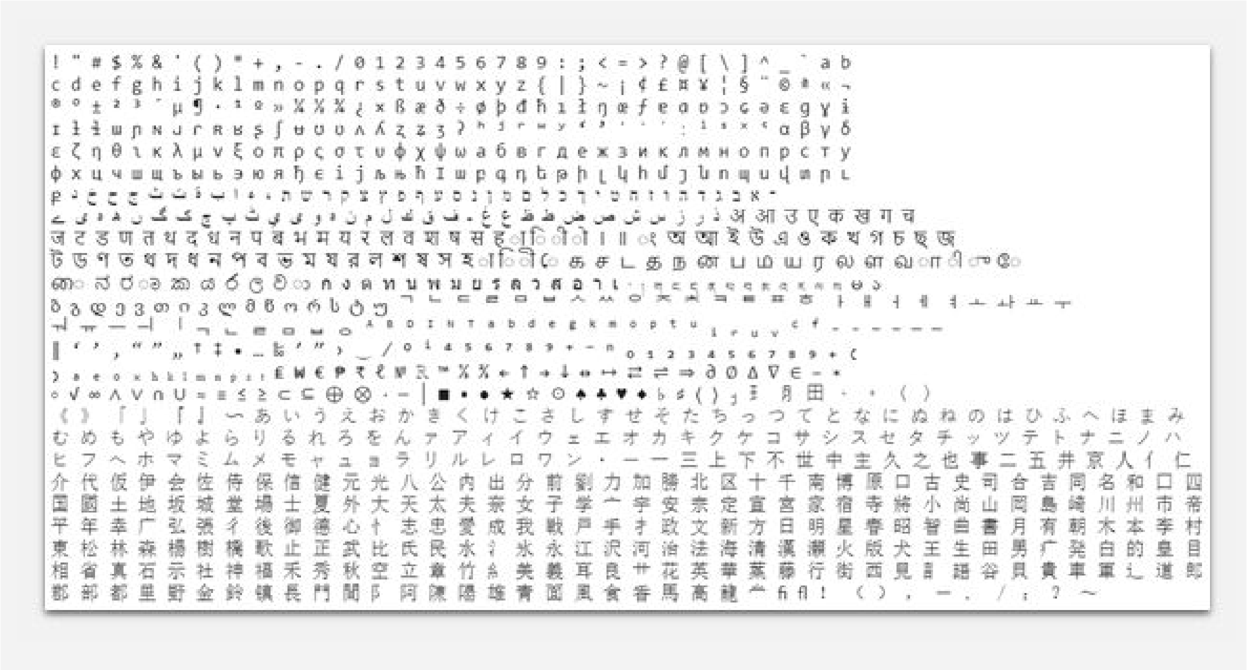


In S03E12 of South Park, Kyle loses a spelling bee on this word, which means, of course, “Something with a kroxldyph-like quality”… actually it’s nonsense ;-)

When an “unknown” word is split into subwords and/or characters, each of these pieces becomes a separate token, represented by its own embedding.

BERT even has tokens for punctuation characters, which, if you think about it, can be important for understanding a sentence! The only thing BERT discards when tokenizing your text is the whitespace.

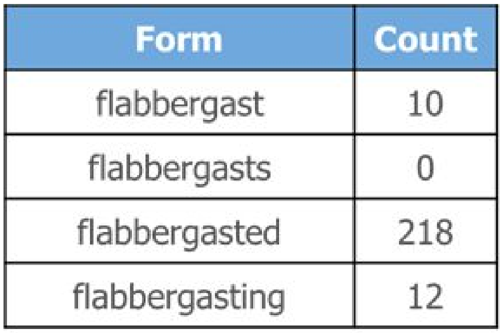
It looks like the authors of BERT simply dumped a whole character set in the vocabulary--here are the 997 individual characters included in its vocabulary:



Benefits of Smaller Vocabularies

It’s great that BERT has a way of handling unknown words, but couldn’t they have trained it on a larger vocabulary? 30,000 tokens is fairly small compared to pre-trained word embedding vocabularies that we’ve seen in the past…

As it turns out, a smaller vocabulary can be a very good thing. The problem with increasing the vocabulary to capture rarer words is that, by definition, there is less training data for these words! For example, here is the number of times different conjugations of the verb “flabbergast” appear in Wikipedia:



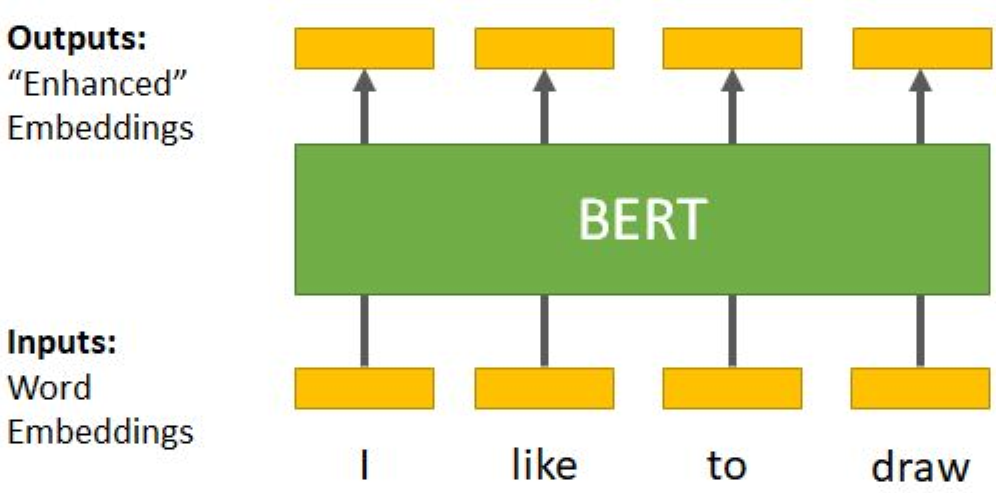
Form	Count
flabbergast	10
flabbergasts	0
flabbergasted	218
flabbergasting	12

If BERT breaks this verb down into pieces, then it can re-use whatever it learns about “flabbergasted” (the most common form) to help it better interpret the rarer forms like “flabbergasting”. If BERT instead had separate embeddings for each form, then they would be “siloed”, with no information shared between them.

1.2. Black Box View

When you 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 (all at once, not in sequence!), and then spits out an “enhanced” version of each of those embeddings on its output.



The output embeddings are the same dimension as the inputs, with 768 features each.

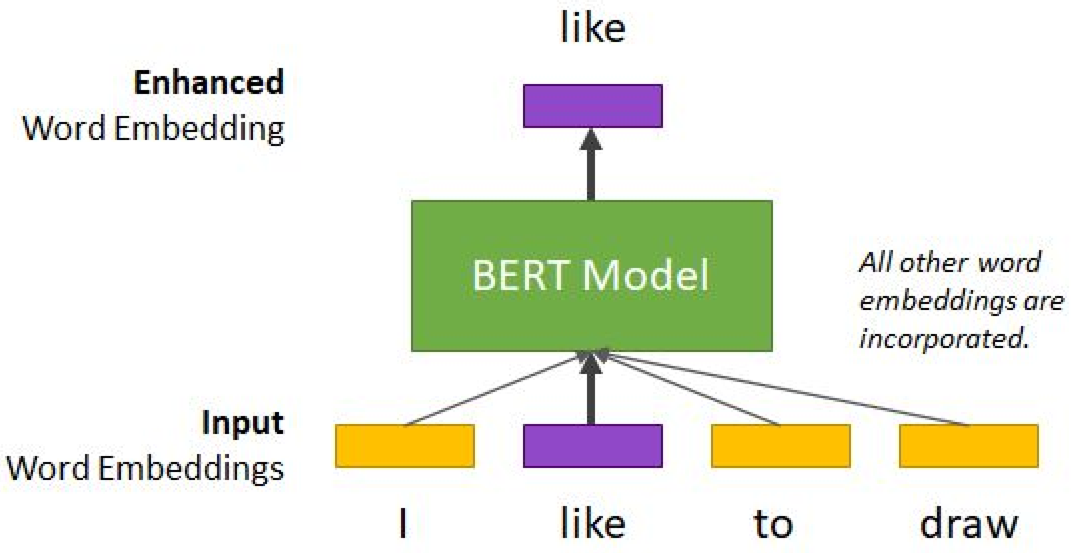
To apply BERT to a specific application, we feed those enhanced output embeddings into some other task-specific (and usually very simple) model.

What it means for BERT to "enhance" the embeddings depends on what application you fine-tune it for. Out of the box, the embeddings on the output are optimized for the two tasks that Google pre-trained BERT on. Those two pre-training tasks (MLM and NSP, from the introduction) aren't intended to be practical or interesting, though, so fine-tuning the model for your task is important.

1.3. Single Embedding View

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 embedding *for a single word* within the input sentence.

In order to enhance the embedding for “like” (see the illustration below) we look at the input embedding for “like”, but we also feed in all of that word’s “context”--the embeddings for all of the other words in the sentence.



You could say that the other words are provided as “supplemental material” for helping BERT interpret the input word.

Variable Sentence Length

Something that troubled me when first learning about BERT was the problem created by sentences being completely variable in length. Neural Networks have fixed input dimensions and fixed output dimensions--it's fundamental to their architecture. Recurrent Neural Networks get around this, of course, by ingesting the words one at a time in sequence. But if BERT isn't recurrent, how does it handle this?

The first key is to recognize that, if you take the previous illustration and ignore the yellow vectors for a moment, looking only at the purple vectors, BERT *does* have fixed input and output dimensions--it has one input vector, and one output vector. The same model is simply run in parallel on all of the words.

The part that *is variable* in length is the set of "supplemental" vectors, colored yellow, which are the rest of the words in the sentence. How does BERT handle those?

Inside BERT, there is a step where we will take the *weighted average* of all of these input vectors. This addresses the variable length problem in two ways:

1. No matter how many input vectors there are, averaging them will always produce one output vector.

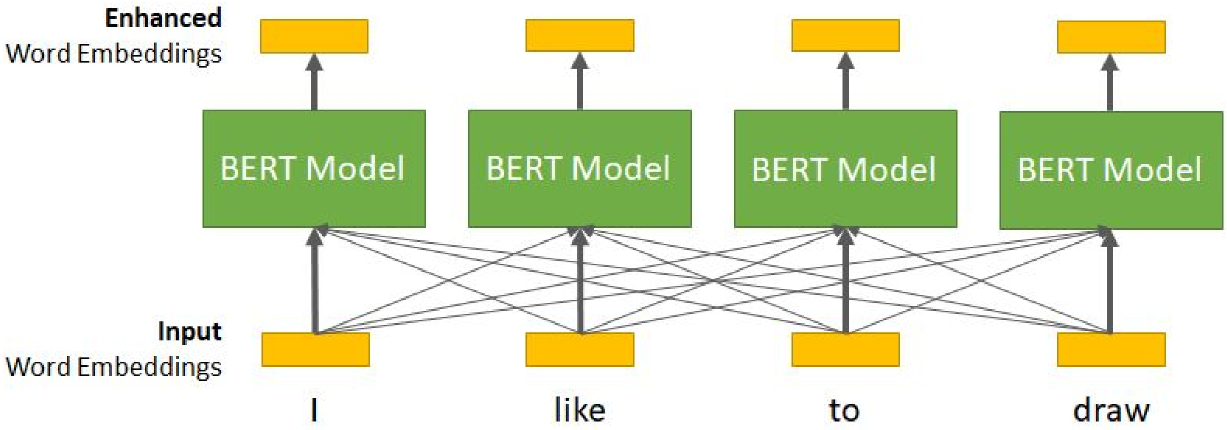
2. If we simply summed the vectors together, then the magnitude would vary depending on the number of vectors. By instead taking their average, the magnitude of the result should be more consistent.

This “weighted average” step occurs at the end of the “Self-Attention” mechanism. We’ll get there!

1.4. Parallelization

Because BERT processes each word indepently, it’s possible to parallelize the process and run all of the input words through BERT “at once”.

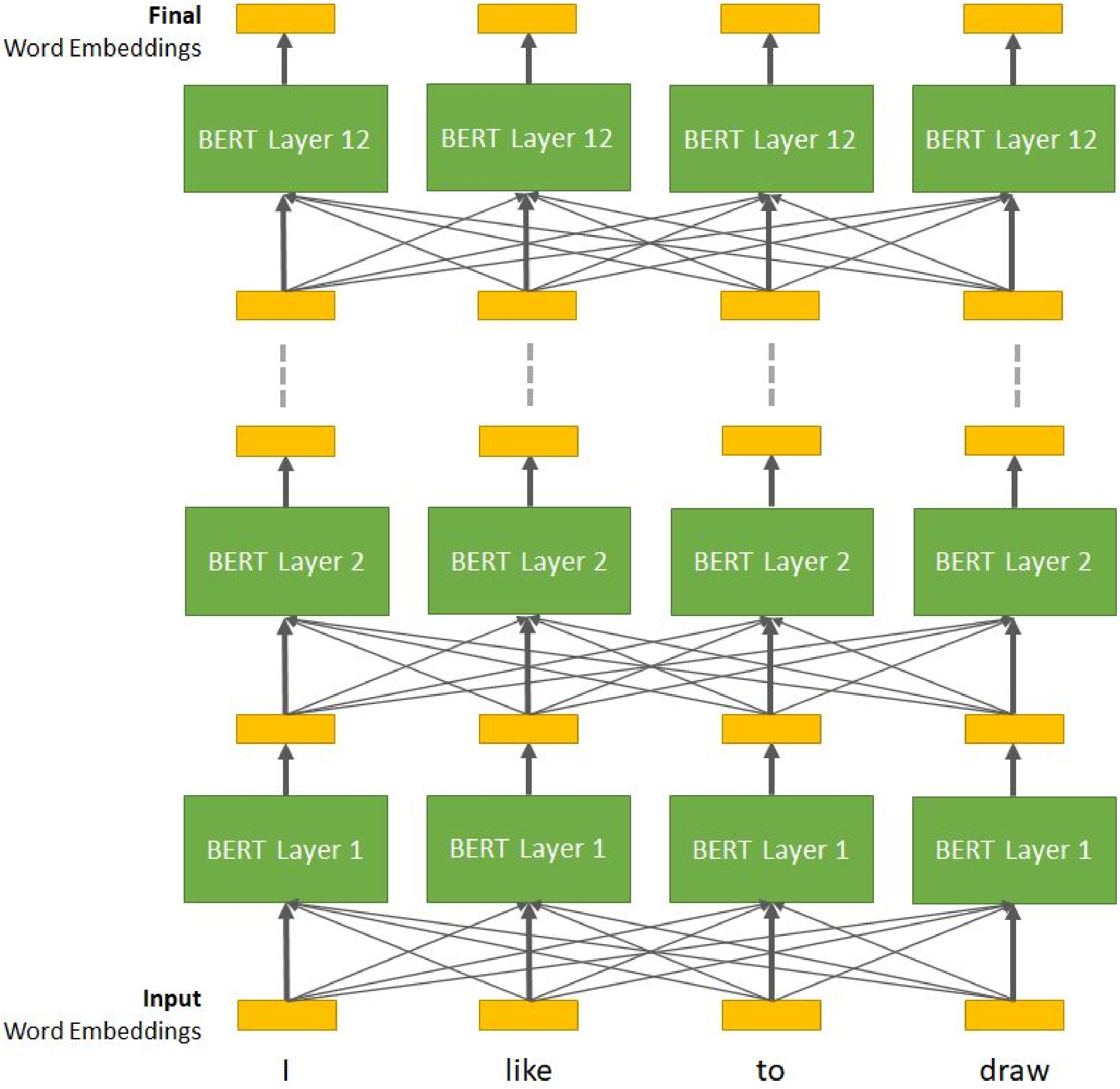
The following illustration reflects this by showing four copies of the BERT Model, each working on a different input word in parallel.



The web of connections on the input illustrates that, for each input word, we will incorporate all of the other word embeddings as well.

Twelve Layers

This process of taking in a set of embeddings and enhancing them is actually repeated *12 times* in BERT. Each of BERT's 12 layers takes in the enhanced embeddings from the previous layer, and further enhances them.

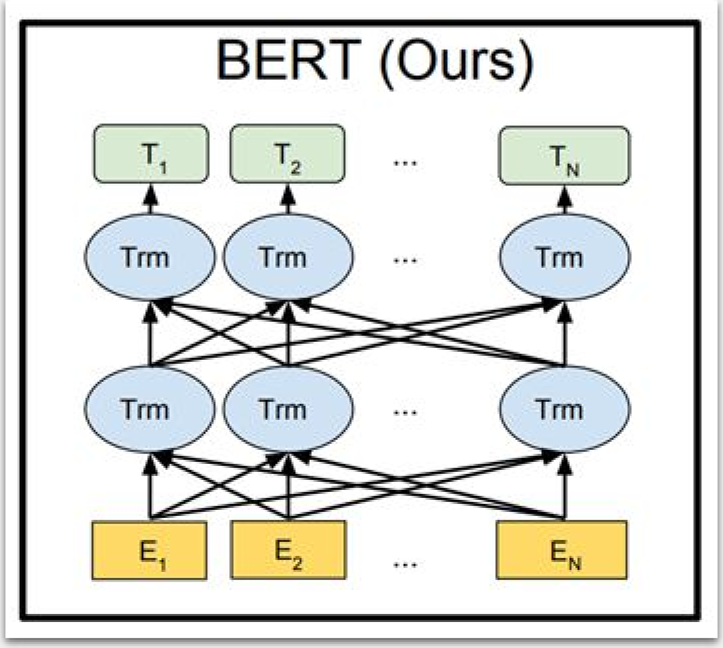


This illustration is meant to convey several aspects of BERT:

1. *The "webbing" shows that all words are incorporated in processing each word.*
2. *The same architecture is applied to each word in parallel.*
3. *There are 12 layers to the model.*

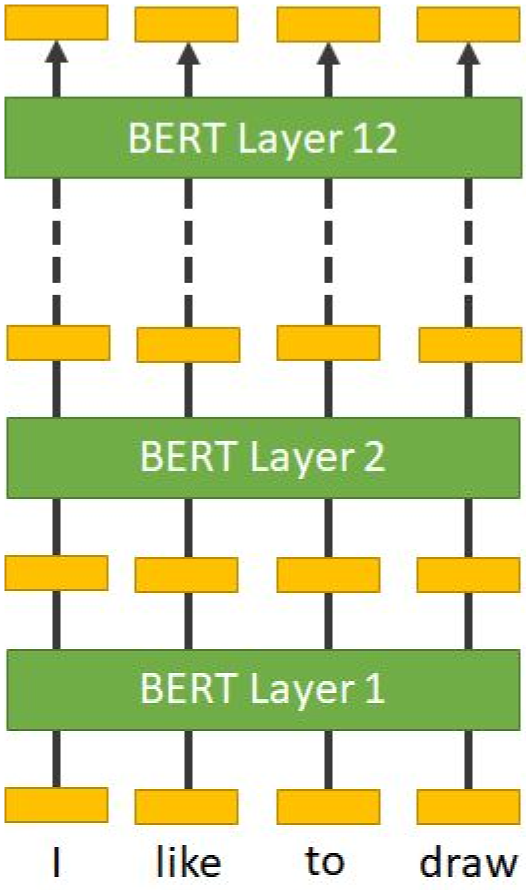
Note that this illustration reveals some dependency between the words--we have to get back all of the updated embeddings from layer 1 before we can start executing layer 2. By the same token, BERT is intended to be applied to tasks where the entire input text can be presented to it at once. This means it may not be suitable in tasks where you want to operate on the words as you receive them, such as in speech recognition.

The above illustration matches this one from Figure 3 of the BERT paper:



“Trm” is short for “Transformer”, which is the architecture used by each BERT layer

From here on out, to simplify the illustrations, I won’t include the web of connections at the input of each layer, they’ll be implied:



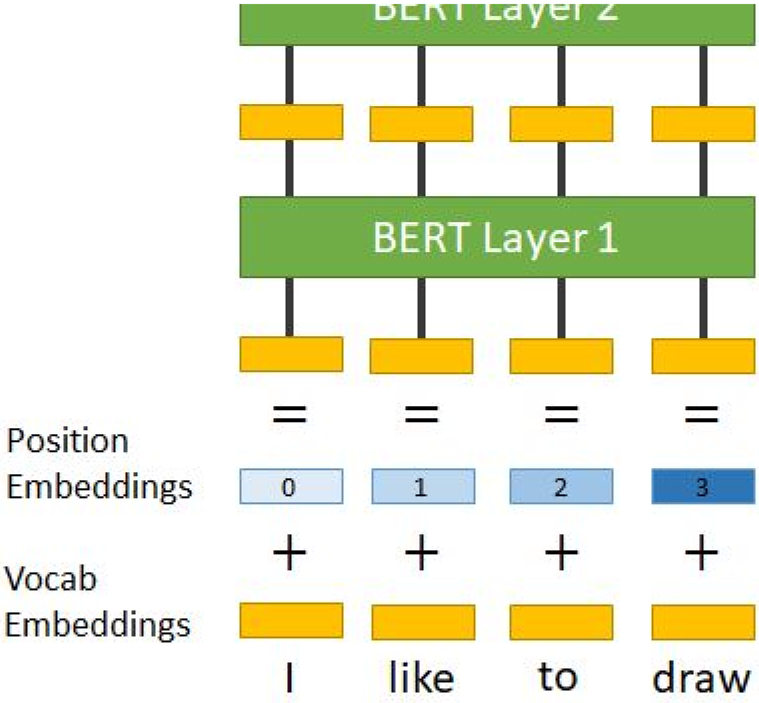
1.5. Encoding Word Position

In the previous illustrations, I placed the words and their embeddings in the order that they occur in a sentence. This arrangement is actually somewhat misleading… Really, I could have arranged the words in any order 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.

BERT does not have this dependence--it can be applied to the words in any order (or all at once in parallel, as we do in practice).

To perform as intelligently on NLP tasks as it does, BERT requires some notion of the word order. The way BERT incorporates the relative position of the words is fairly peculiar… BERT has a set of 512 "**Positional Encoding (PE)**" embeddings. We simply add the PE vector to the corresponding word.



For each token in the sentence, the model will take its embedding from the vocabulary, then add the PE vector to it

> *corresponding to its position in the sentence. The result is fed into the first layer of the model.*

Note that these PE vectors are only added before layer 1, and *not* between any of the other layers.

Also, 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; if you go over this limit, the most common workaround is to simply truncate the input to 512 tokens.

1.6. 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.

Because these details are specific to allowing BERT to support certain applications, I've left their explanation for the next section.

2. Ways to Apply BERT

So far we have seen what BERT ultimately "does"--it takes a set of input embeddings, and produces a set of better ones, which have been "contextualized" (their meaning and content has been changed to reflect how they are used in the input sequence).

Before diving into the details of *how* BERT does this, I think it's helpful to clarify what tasks BERT can be applied to (and which ones it can't!).

The types of tasks that BERT can handle can be grouped into 3 main categories:

- Token Classification
- Text Classification
- Text-Pair Classification

Note: I used the term "Classification" for simplicity and because that's the most common operation, but you can just as easily apply BERT to "Regression" versions of these tasks. Regression just means that you produce a floating point value (e.g., a predicted stock price) rather than a class label.

2.1. Token Classification

Token-level tasks involve making a prediction on each token in the input. We've seen that BERT takes in a set of word embeddings and produces a set of better word embeddings, so token-level tasks are the most "natural" application.

For **Named Entity Recognition (NER)**, you can add a simple classifier to the output of BERT which looks at each final embedding and predicts whether they are, e.g., the name of a person or not.

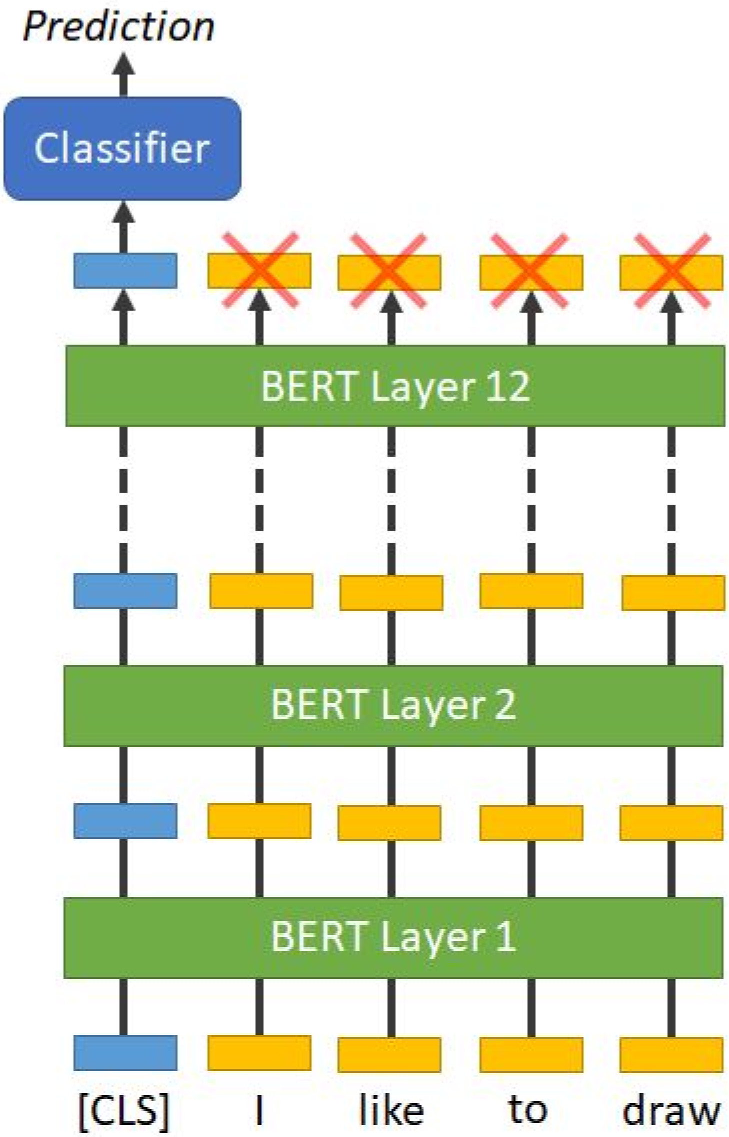
Question Answering is a more complicated token-level task. We give BERT a question and *a passage of text* containing the answer, then apply a classifier to each token in the answer text to identify the tokens marking the start and end of the answer. (If you're curious to learn more about this, I've given a YouTube tutorial on the topic).

2.2. Text Classification

BERT can be applied to traditional NLP tasks such as text classification and sentiment analysis (assessing whether, e.g., a user review expresses negative or positive sentiment about a product).

Given that BERT produces a collection of embeddings, the natural next step would be to perform some kind of pooling on these output embeddings (e.g., averaging them), then sending the result into a classifier. BERT has a better way, though…

Whenever you feed in a piece of text to BERT, the first token (which I’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.



"[CLS]" is a special token that is always placed at position 0.

If you look in BERT's vocabulary, there is literally an entry with the string "[CLS]" (not including the quotes). Presumably CLS is short for classification.

When Google performed their extensive pre-training of BERT, one of the two pre-training tasks is called “Next Sentence Prediction (NSP)”, and it’s a text classification task. They added a simple linear classifier to the end of BERT, and only the final embedding for the [CLS] token was fed into it.

Because of this, BERT learned that, in order to perform well on this task, it needed to *enrich the [CLS] embedding with all of the information it would need in order to make the classification decision*. So BERT has this built-in behavior of expecting [CLS] as the first token, and enriching the [CLS] embedding with information about the whole input text.

So, you *could* try your own pooling scheme on the output embeddings, but it’s likely that BERT can find a better one for you, and pool whatever knowledge is needed into that single [CLS] embedding.

Note that in text classification tasks, although we “discard” all of the *final* token embeddings except for [CLS], that doesn’t mean the other embeddings are “wasted”. Keep in mind that the token embeddings were being enhanced by each layer of BERT, and the [CLS] token was in turn being enhanced by looking at all of the token embeddings. So *we do need* those other token embeddings all the way up to the final layer.

BERT expects the [CLS] token to be first no matter what task you are *doing, so it should always be included at position 0, whether you actually need it or not :)*

Padding

As previously stated, BERT can be applied to any sentence length up to 512 tokens. However, in order to support parallel processing of multiple text samples at once, implementations of BERT require that you pick a single fixed length for all input text in your application.

For the sake of parallelization, all input text sequences are padded out or truncated to a single *fixed length* of your choosing. You'll pick a length that suits your application. But note that:

- A shorter fixed-length will make training and evaluating BERT faster.
- BERT does have a hard upper limit of 512 tokens.

Example Code

BERT Fine-Tuning Sentence Classification v3.ipynb

- I've made the above Colab Notebook freely available; it will take you through a sentence classification task using BERT in PyTorch.

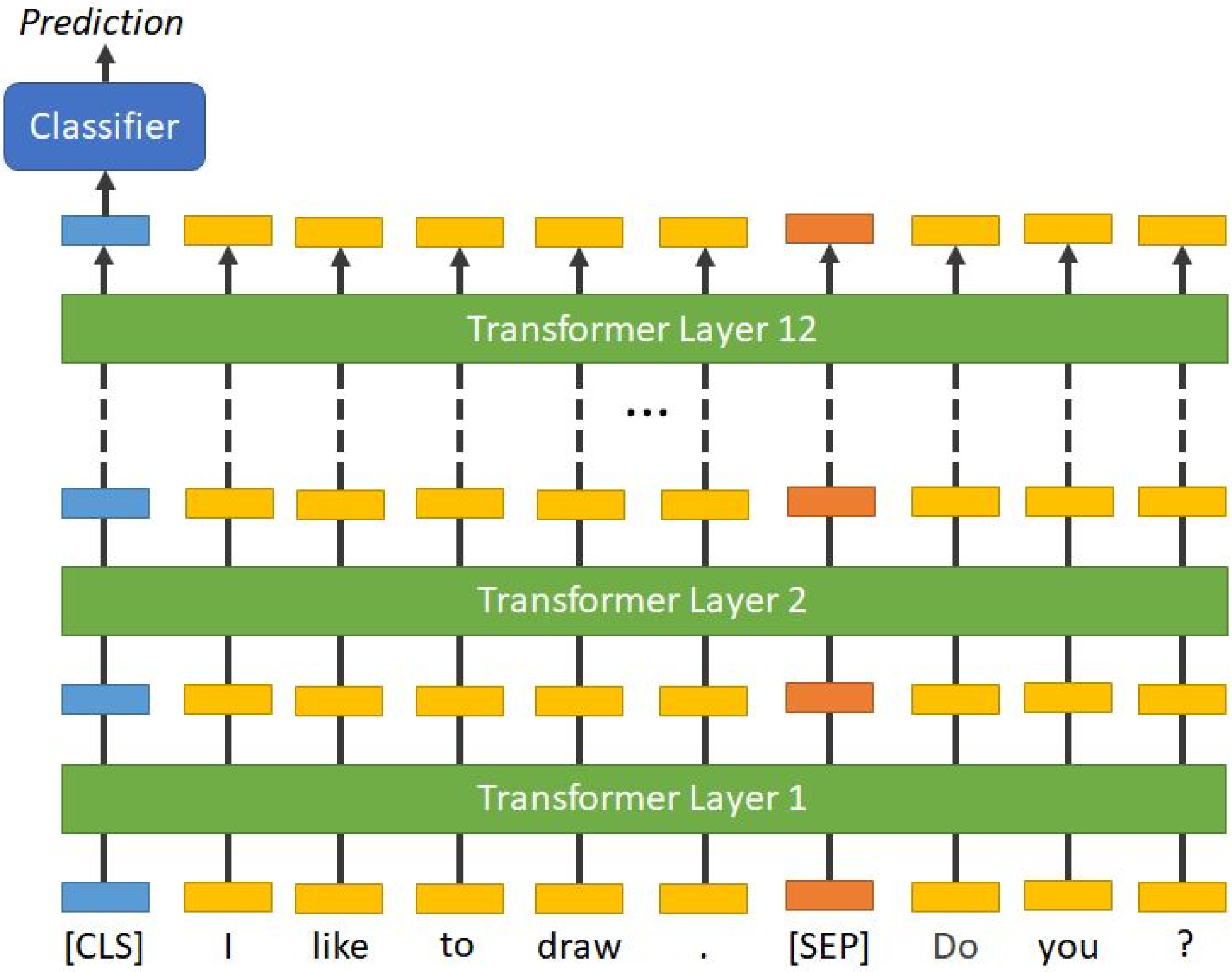
2.3. Text-Pair Classification

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.

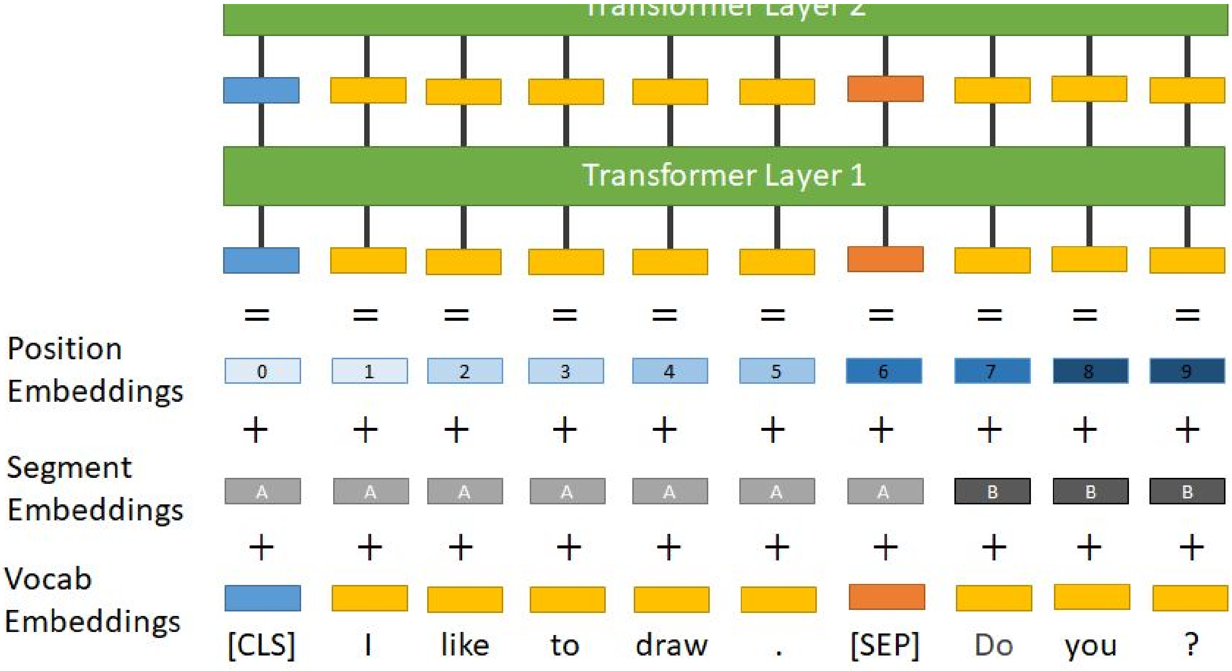
A practical use of this might be de-duplication. For example, in 2017 Quora released a “Question Pairs” dataset, where the task is to detect whether two questions posted on Quora are essentially the same.

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.



Note that the convention (established by BERT's pre-training) is for the [CLS] and [SEP] tokens to receive the "Segment A" embedding.

One of the benchmark tasks on which BERT competes is called **Natural Language Inferencing**, and this a text-pair classification task. Given two sentences, the task is to determine how they logically relate to one another. Do they either (1) contradict each other, (2) "entail" one another (meaning the second follows logically from the first), or (3) are they "neutral" in their relationship?

To my knowledge, NLI is more of a "very difficult task on which to benchmark and compare models", rather than something with direct practical application.

Finally, although Question Answering is really a token-classification task, it does make use of this two-segment input feature in that the model needs to be given both a question and a passage containing the answer.

2.4. What BERT Can’t Do

Is BERT the answer to all of NLP? Sadly, no.

Simply put, BERT cannot perform **Text Generation.** It cannot translate text from one language to another, or generate a written response to a question.

You might be surprised to learn that the authors of BERT did not come up with BERT’s architecture--it is taken directly from an earlier model called the **Transformer**. The Transformer included an additional component that allowed it to generate text, but this had to be removed in order to create BERT.

For those who are curious, the remainder of this chapter provides some of the history and context to BERT, and explains why it was necessary to remove the text generation portion of the Transformer.

BERT vs. The Transformer

The Transformer was designed to perform **Machine Translation** (translating text from one language to another). It consisted of two major parts--an “Encoder” and a “Decoder”. The **Encoder** would look at the whole input sentence in one language and encode it into embeddings, and then the **Decoder** would use those embeddings to generate the output sentence in a different language.

> The Transformer was introduced in a paper called Attention is all you need, by Vaswani et. al., which came out of Google in 2017.

Text Generation

BERT is literally just the *Encoder* portion of that model, scaled up a bit, and trained on MLM and NSP instead of translation.

While the original Transformer needed to be trained on (man-made) translation examples, BERT is pre-trained on raw, unlabeled text--a *much* larger dataset!

The result is an *incredibly powerful Encoder* (BERT), but with no Decoder portion. The loss of the Decoder means that BERT is *not applicable* to tasks where we want to *generate text* (such as translation).

“All-at-once” Input

Ready for another surprise about BERT? The authors weren’t the first ones to try stripping away the Decoder and performing unsupervised pre-training. That distinction goes to the OpenAI-GPT model, and the GPT was the inspiration for BERT.

Here’s what makes BERT special. OpenAI trained their model on a common unsupervised learning task called **Language Modeling**, which is simply to predict the next word in a sentence. So, using our example sentence, you could feed in “I like to”, and the model has to predict “draw”.

Language Modeling has that same wonderful property of only requiring raw, unlabeled text. But BERT’s MLM and NSP tasks have a very important advantage…

In MLM and NSP, you can give BERT *the entire input sequence at once*, and BERT gets to factor-in all of the words in the text into its predictions. In Language Modeling, the OpenAI-GPT only gets to see the words *to the left* of the word that you’re currently predicting. This is the significance behind the ‘B’ in BERT’s name--it stands for “Bidirectional”, as opposed to left-to-right.

The difference between the traditional “Language Modeling” task and BERT’s “Masked Language Model” is very subtle, but it makes BERT the champion!

Because BERT works best by looking at the entire input all at once, it may not be as helpful in applications where you want to process the text in real-time as it comes in, such as in auto-completion or speech recognition.

Training Sample Efficiency

Even the MLM task was not novel to BERT! It’s also known as the “Cloze task”, and others have experimented with it as an NLP training task in the past, just not with the Transformer.

One reason MLM was dismissed in the past is that it is much less training-sample-efficient--you can only mask out a percentage of the words (BERT masks 15%), whereas Language Modeling gives you a training sample for every single word in the text! This, in theory, might make ~6x slower to converge, but the reality was not as bad.

> The insights in this section came primarily from an interview by Kaggle with BERT’s primary author, Jacob Devlin, which you can watch on YouTube here.

Part 2 - BERT Architecture

3. Self-Attention

At this point, we know how BERT tokenizes text, and we know that it ultimately produces a set of "enhanced" embeddings for each token. What's the mechanism behind this "enhancing", though? How does BERT actually make sense of language?

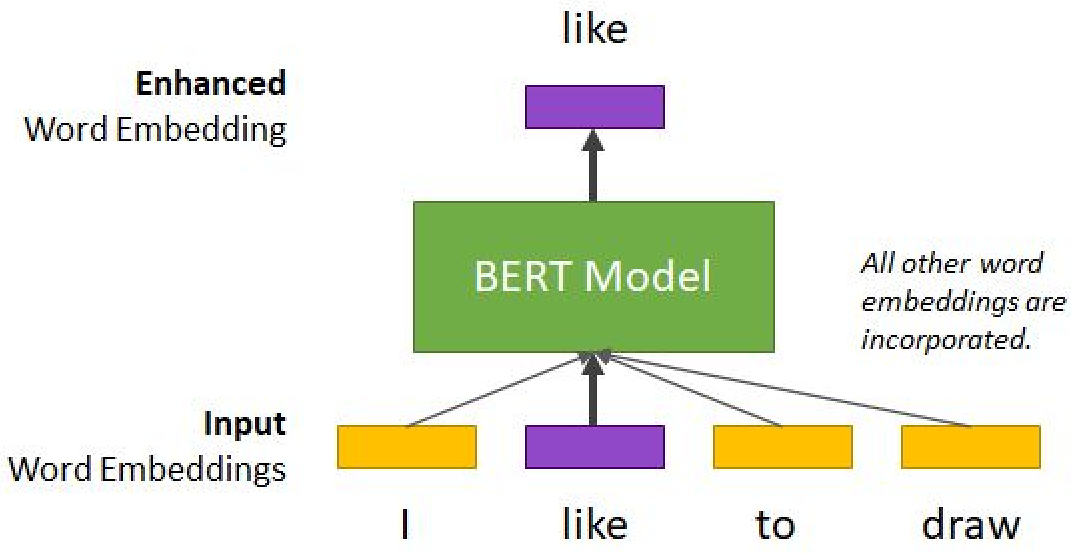
Again, BERT's architecture comes directly from the Transformer model--it's just the 'Encoder' portion of that model. And the most important piece of the Transformer's / BERT's internal architecture is a technique called **Self-Attention**.

> Because BERT just uses the Encoder from the Transformer, you won't find any explanation of BERT's actual architecture in the BERT paper--the authors refer you elsewhere, making the BERT paper a pretty unhelpful place to start if you're trying to understand its architecture :)

Acknowledgment: This chapter would not have been possible without Jay Allamar's "Illustrated Transformer" blog post here! The explanation and illustrations in this chapter are my own, but Jay's post was key in helping me make sense of the Transformer architecture. Thank you, Jay!

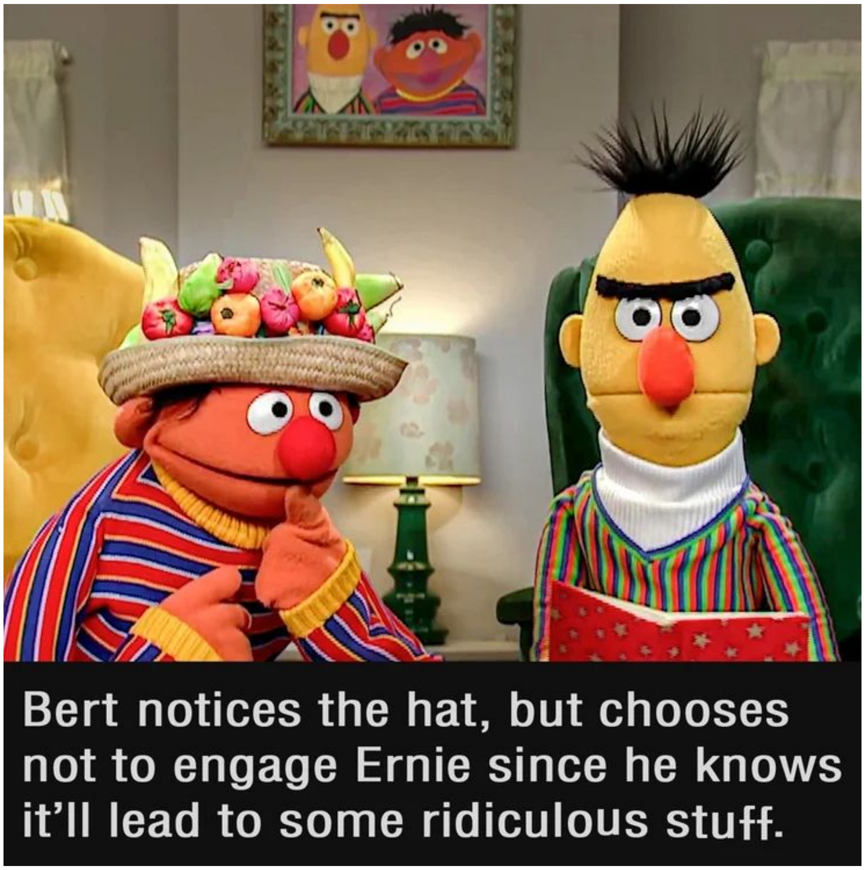
3.1. Weighted Average of Embeddings

We previously looked at the below example, where I explained that, when enhancing the embedding for the word "like", we also incorporate the embeddings for the other words in the sentence:



Self-attention is the piece that actually "incorporates the other embeddings".

To see what it does, let's look at a more interesting sentence:

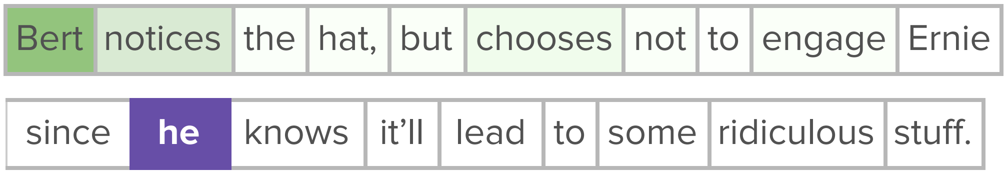


Bert notices the hat, but chooses not to engage Ernie since he knows it'll lead to some ridiculous stuff.

Midway through this sentence is the pronoun “he”. What does “he” refer to? It could be either Bert or Ernie… or maybe there's even a character named “the hat”?!

When producing the enhanced embedding for the word “he”, self-attention will take a **weighted-average of the embeddings** of the other context words. The weight self-attention assigns to each context word is, you could say, how much *attention* to give to that context word.

So, when processing the input word “he” (highlighted in purple), self-attention might give the following weights to the different words in the sentence (a darker shade of green means give this word more weight):



Bert notices the hat, but chooses not to engage Ernie

since **he** knows it'll lead to some ridiculous stuff.

The background color of the other words reflects how much attention BERT is giving to each word--in this case BERT is focusing primarily on the name of the character that 'he' refers to--"Bert" .

(Note that, for the sake of illustration, I didn't break the sentence down into real tokens, or split out punctuation).

Some details that you may be wondering about:

- Every word will be given *some* weight (none are given zero weight)
- The embedding for the input word ("he" in this example) *is included* in the weighted average.
- The weights are calculated with a SoftMax function (we'll look at this soon), which means:
    - They are fractions between 0 - 1.0 (not inclusive).
    - They will all add up to exactly 1.0.

How do you calculate a "weighted average"? Instead of simply summing the values and dividing by the count, you first multiply each value by its weight, then divide the sum by the sum of the weights (instead of the count).

This is expressed by the standard formula for calculating a weighted average, shown below. In our context, the variables have the following meanings:

- n is the number of the words in the sentence.
- i is the position of a word in the sentence.
- x_i is the embedding for the word at position i.
- w_i is the weight value (between 0 - 1.0) given to the word at position i.

$$\overline{x} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

The result, $\overline{x}$, is the enhanced embedding for the current input word!

We'll see that all of the weights are guaranteed to sum to 1.0 (they're calculated using a Softmax function), so we can actually simplify this function and remove the denominator:

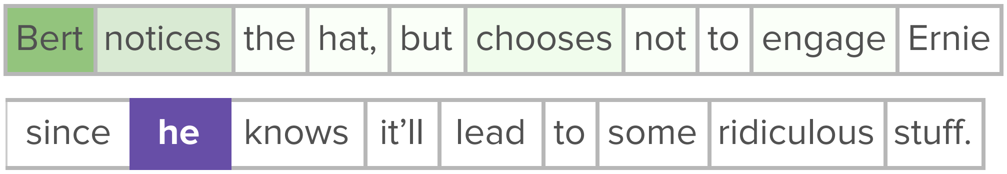
$$\overline{x} = \sum_{i=1}^{n} w_i x_i$$

Note: Before taking the weighted average of the embeddings, we'll perform an important transformation to each of them--I'll cover this down in section 3.3.

3.2. Calculating the Attention Weights

For a given input word, how does Self-Attention determine the weights to assign to each word in the sentence?

We'll continue to use the example sentence from the previous section:

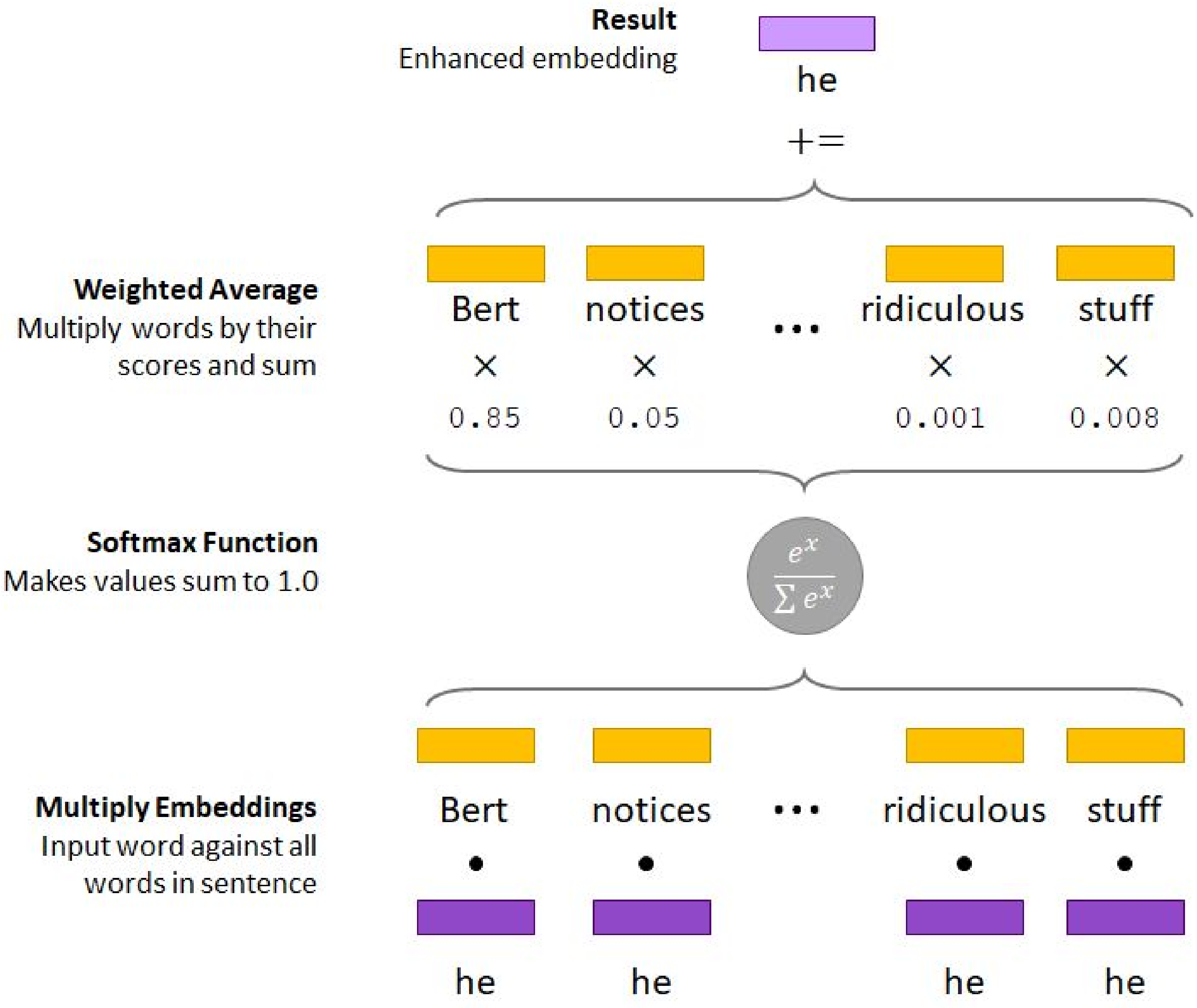


Bert notices the hat, but chooses not to engage Ernie

since he knows it'll lead to some ridiculous stuff.

For the input word "he", let's say we want to calculate the weight to give the context word "Ernie". Essentially, we are going to take the dot product between the embedding for "he" and embedding for "Ernie" to calculate the weight to give "Ernie". We'll do this for all of our context words, and feed them through the Softmax function, and we have our weights!

Below is an illustration of this process (starting from the bottom and moving upward to the result at the top)



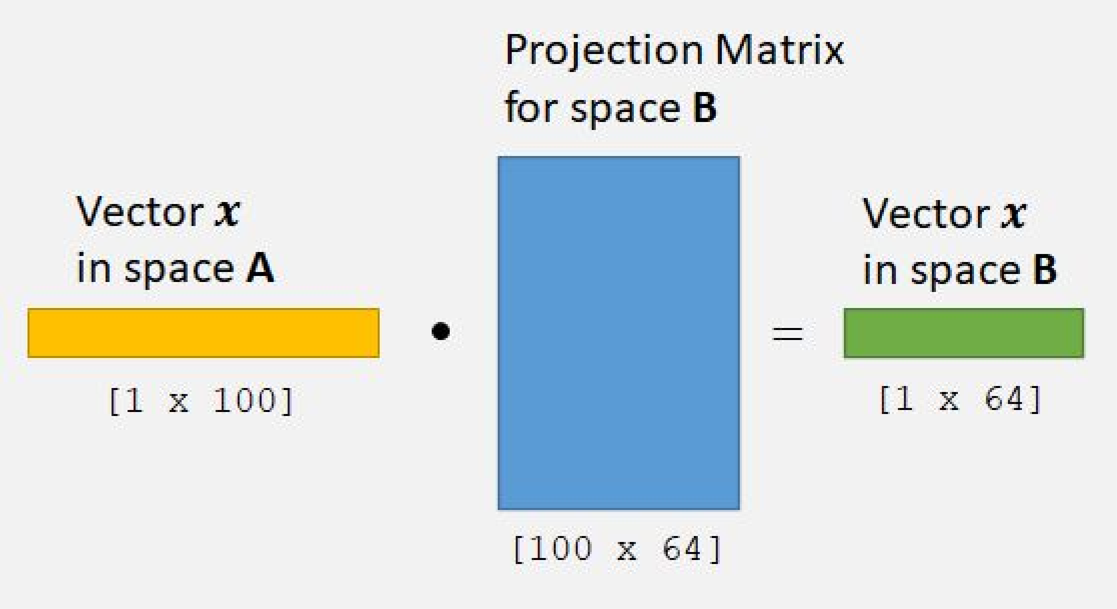
3.2. Projecting the Embeddings

The above illustration captures the arithmetic for self-attention, but is missing a very important pre-processing step that we'll apply to the vectors first.

We are going to **project** the embeddings onto several different **vector spaces.**

What does it mean to project onto a vector space?

Projection is a powerful technique in machine learning where we take the vector representation of an object and multiply it against a **projection matrix** in order to place it into a different **vector space**.

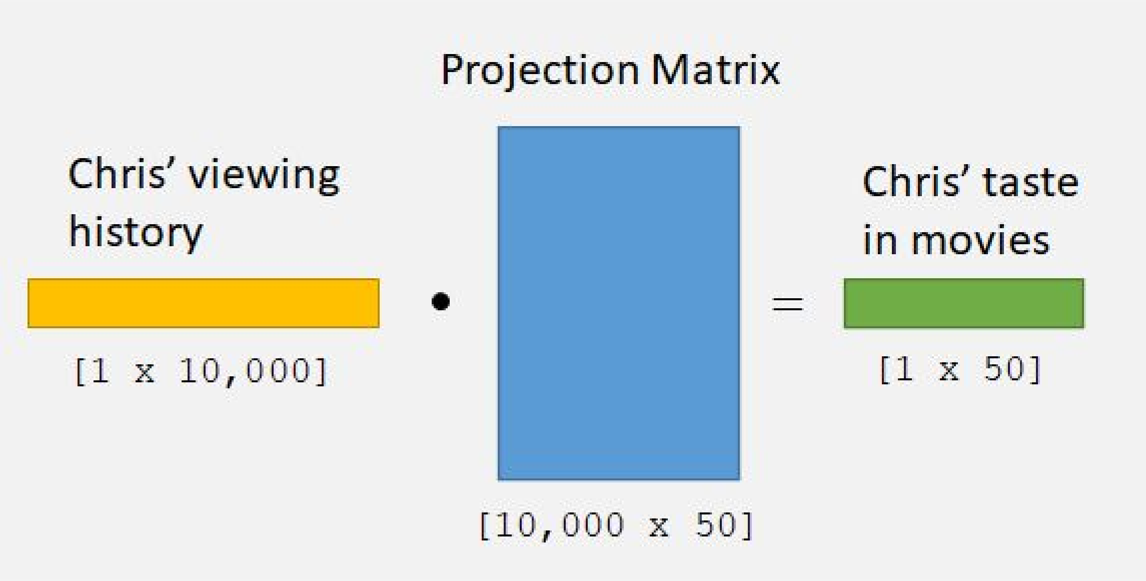


Both the **yellow** and the **green** vectors are representations of ‘x', but the two spaces (A and B) expose different aspects of ‘x’.

Example

Let’s say you are Disney and you have a vector for the user “Chris” that has 10,000 dimensions. Each position in the vector corresponds to a specific Disney movie or show on Disney Plus.

For every movie or show that Chris has watched, there is a “1” in that position, and 0s for everything he hasn’t seen.



After multiplying Chris’ viewing history vector against a particular projection matrix, we get a 50 dimensional vector of floating values that is now an abstraction of Chris’ taste in movies!

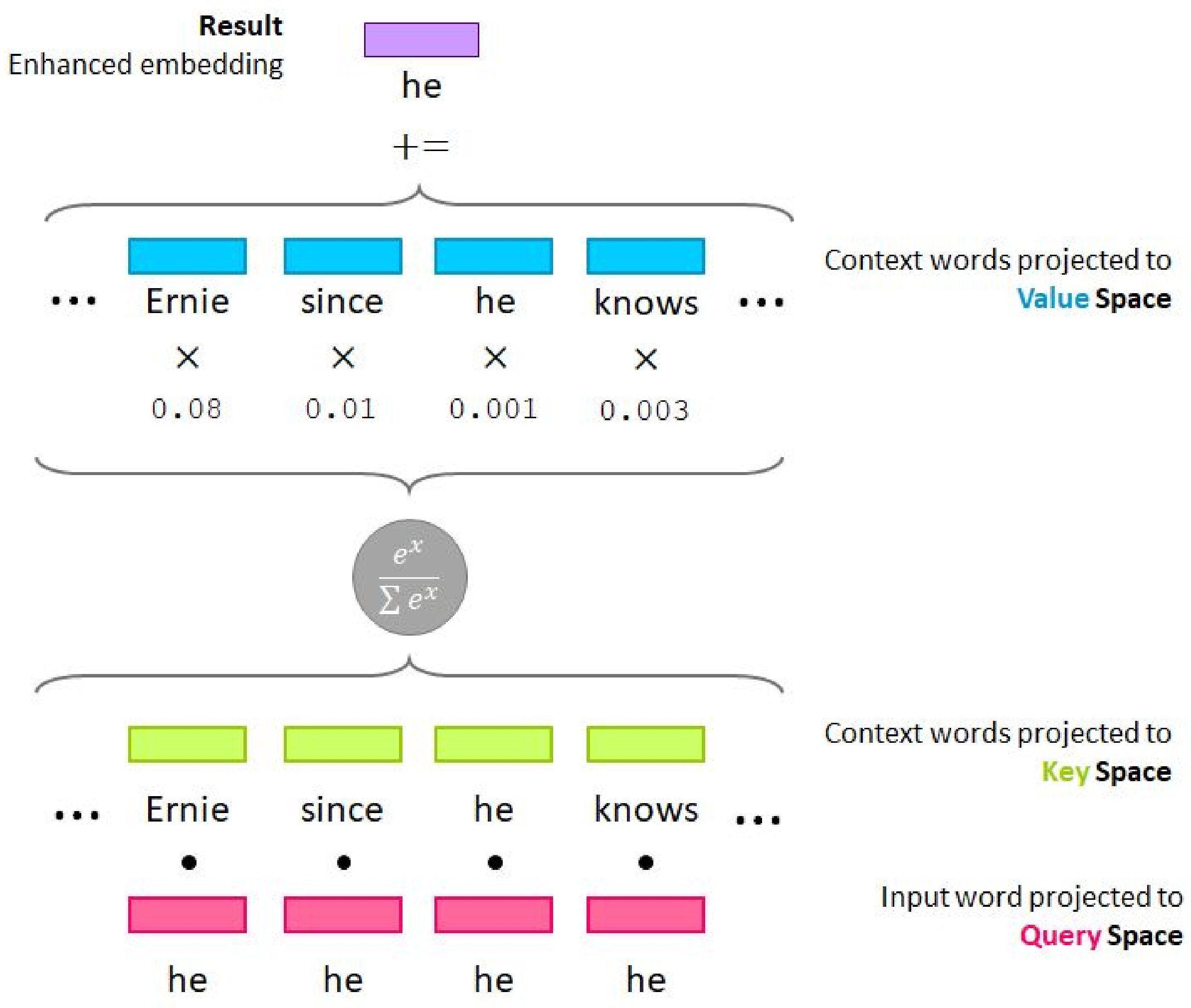
We are going to project our word embeddings onto three different vector spaces. The authors of the original Transformer architecture used the names “Query”, “Key” and “Value” for these spaces.

Here’s how these three spaces will be used.

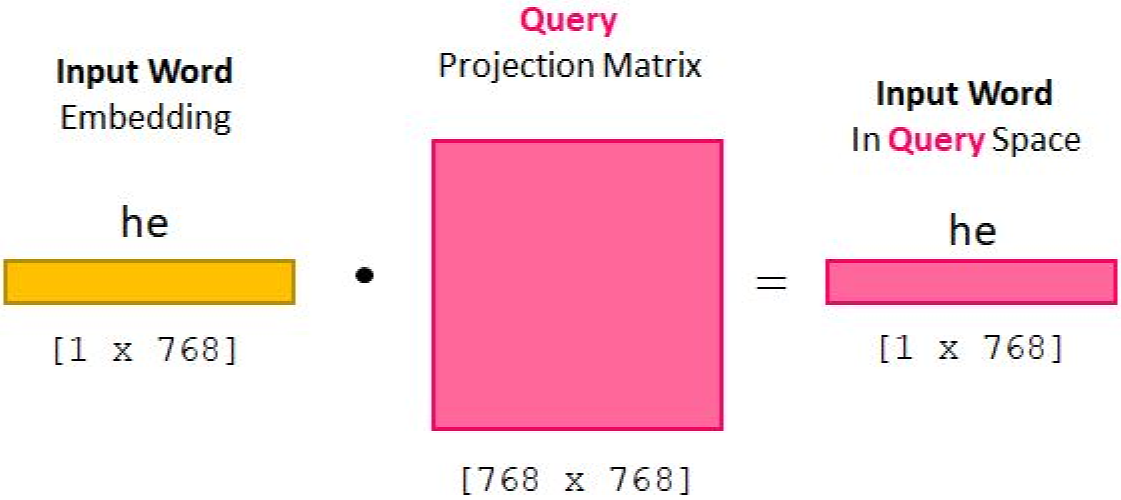
1. We will project our input word “he” onto the **Query Space**.
2. For the dot-product step, we will project every context word onto the **Key Space**.

3. For the weighted average step, we will project every context word onto the **Value Space**.

The below illustration reflects these changes--notice that it's still the same process, we've just applied some transformations (specifically, projections) to the vectors.



The projections, again, are done simply by multiplying the embeddings with the Projection Matrix for each space, as illustrated below for each of the three spaces.



Query Projection Matrix

Input Word Embedding

he

[1 x 768]

•

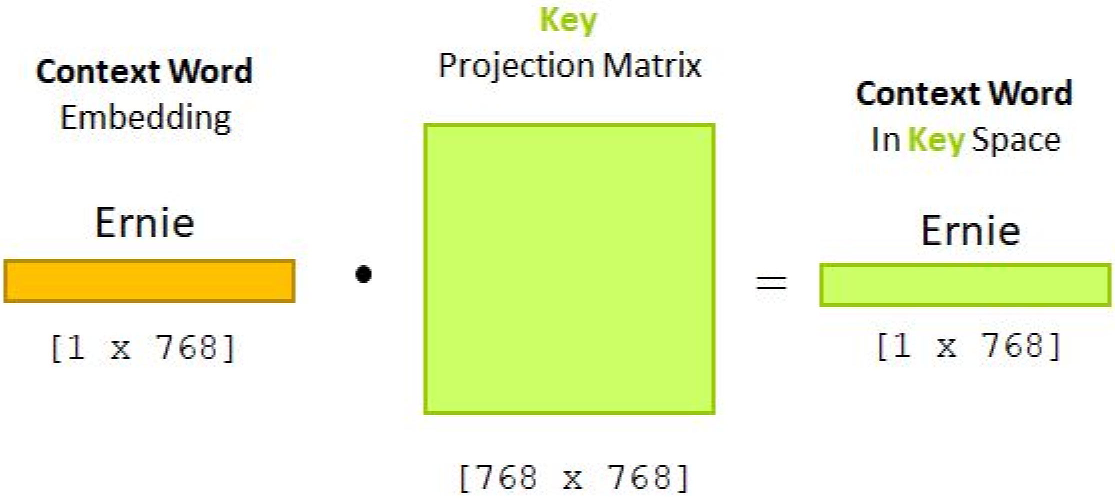
[768 x 768]

=

Input Word In **Query** Space

he

[1 x 768]



Key Projection Matrix

Context Word Embedding

Ernie

[1 x 768]

•

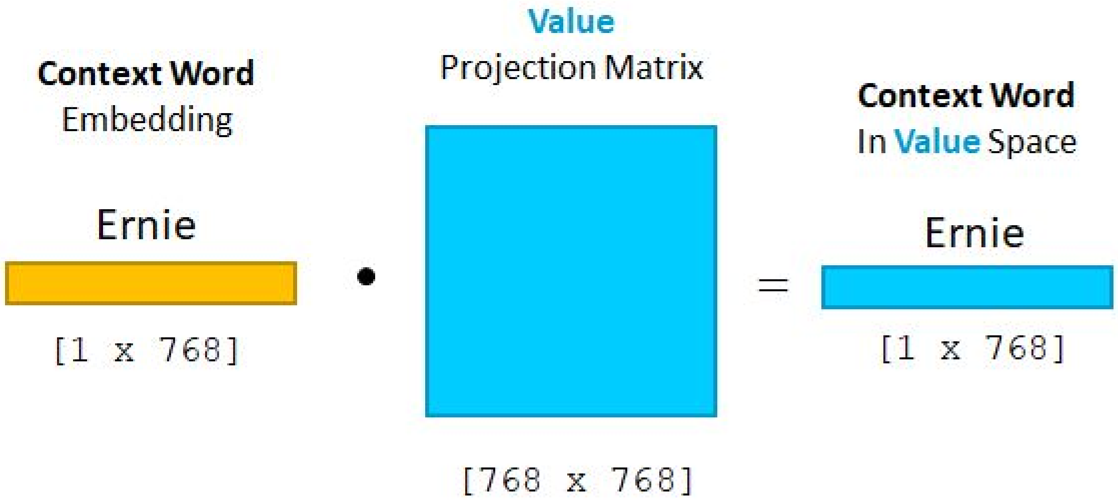
[768 x 768]

=

Context Word In **Key** Space

Ernie

[1 x 768]



Context Word Embedding

Ernie

[1 x 768]

●

Value Projection Matrix

[768 x 768]

=

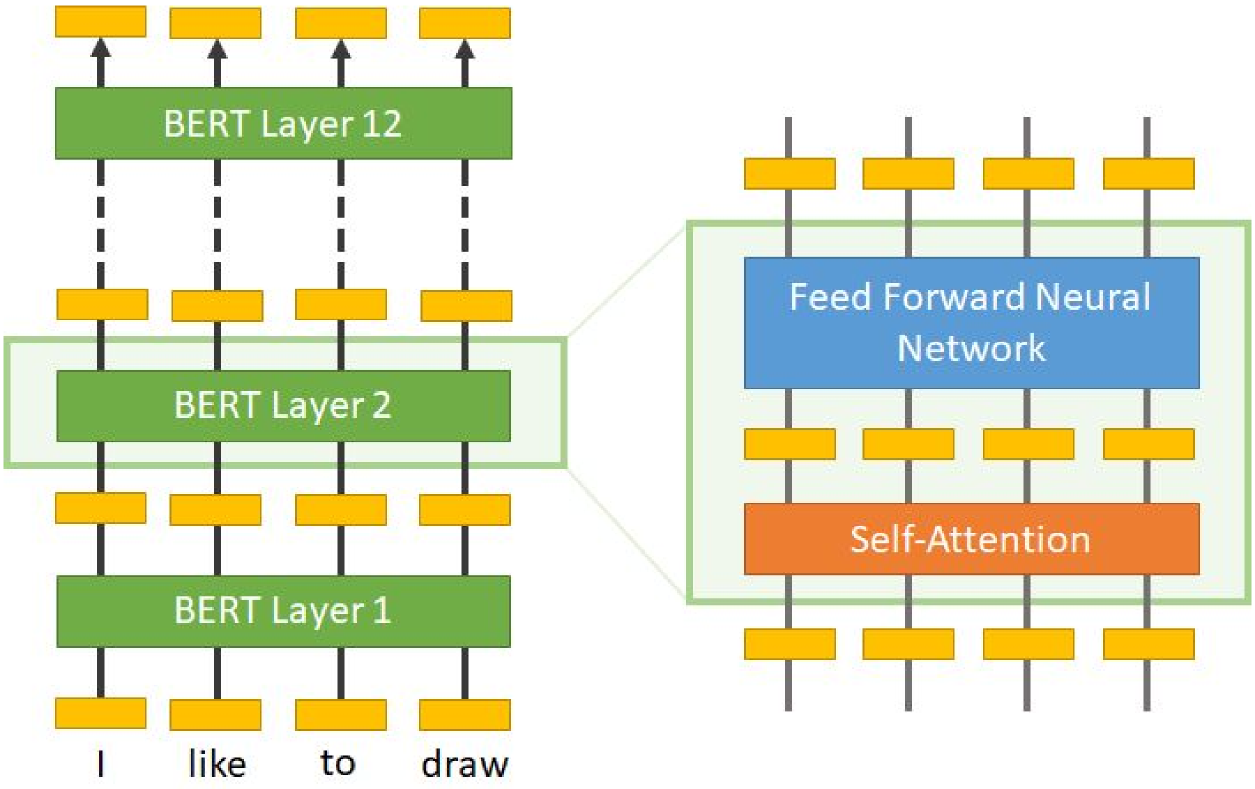
Context Word In **Value** Space

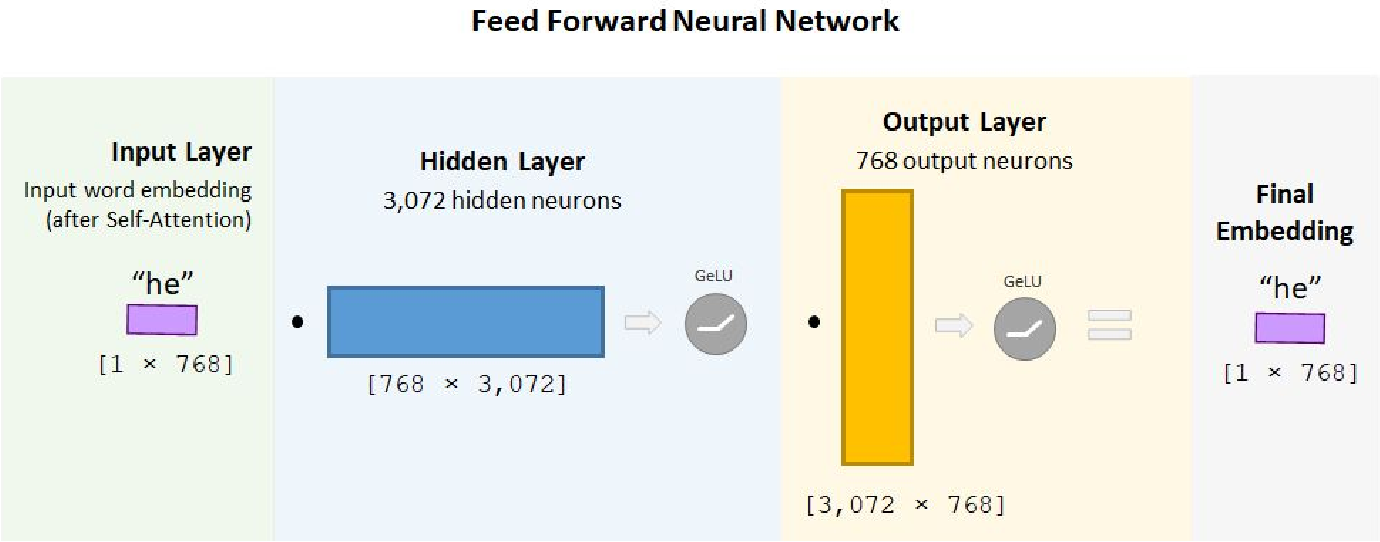
Ernie

[1 x 768]

4. Feed-Forward Neural Network

Inside of a single BERT layer, there is one more major component after self-attention--but it’s a simple one! After we apply Self-Attention to a word embedding, we send it through a standard 3-layer neural network (referred to as the “Feed Forward Neural Network” in the original paper).





This neural network is sized such that you give it a 768-dim input vector (the word embedding after self-attention), and you get back another 768-dim vector on the output. Specifically, the network has 3,072 hidden layer neurons and 768 output neurons.

Here's what this neural net looks like in terms of the weight matrices.

The convention in the original Transformer and in BERT is to set the number of hidden layer neurons in this network to be 4x the embedding size, so 4 x 768 = 3,072 hidden neurons.

> *I'm unfamiliar with the 'GeLU' activation function, but section A.2 of the BERT paper states: "We use a gelu activation (Hendrycks and Gimpel, 2016) rather than the standard relu, following OpenAI GPT."*

I don’t currently have any great insight into what this component accomplishes for BERT; however, I think it’s interesting to note that this “FFN” contains a substantial portion of the total weights in BERT:

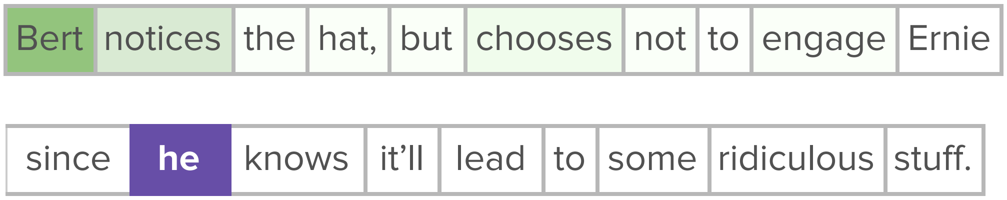
- The Self-Attention component consists primarily of the three projection matrices, each 768 x 768, for a total of `3 × 768 × 768 = 1,769,472` weights.
- The neural net consists of the hidden-layer and output-layer weights, both 768 x 3,072, for a total of `2 × 768 × 3072 = 4,718,592` weights.
- This means that the neural network is roughly **2.66x** larger than the self-attention piece!

Component	# of Weights	% of Total
Self-Attention	~1.8M	27%
Feed Forward Neural Network	~4.7M	73%
TOTAL	*~6.5M*	*100%*

Note that the above calculations are only for a single one of the 12 BERT layers!

5. Multi-Headed Attention

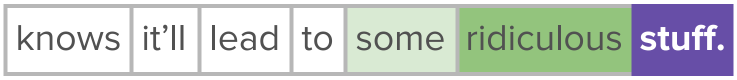
In our example sentence, I’ve illustrated self-attention paying attention to the person that “he” refers to:



Bert notices the hat, but chooses not to engage Ernie

since **he** knows it’ll lead to some ridiculous stuff.

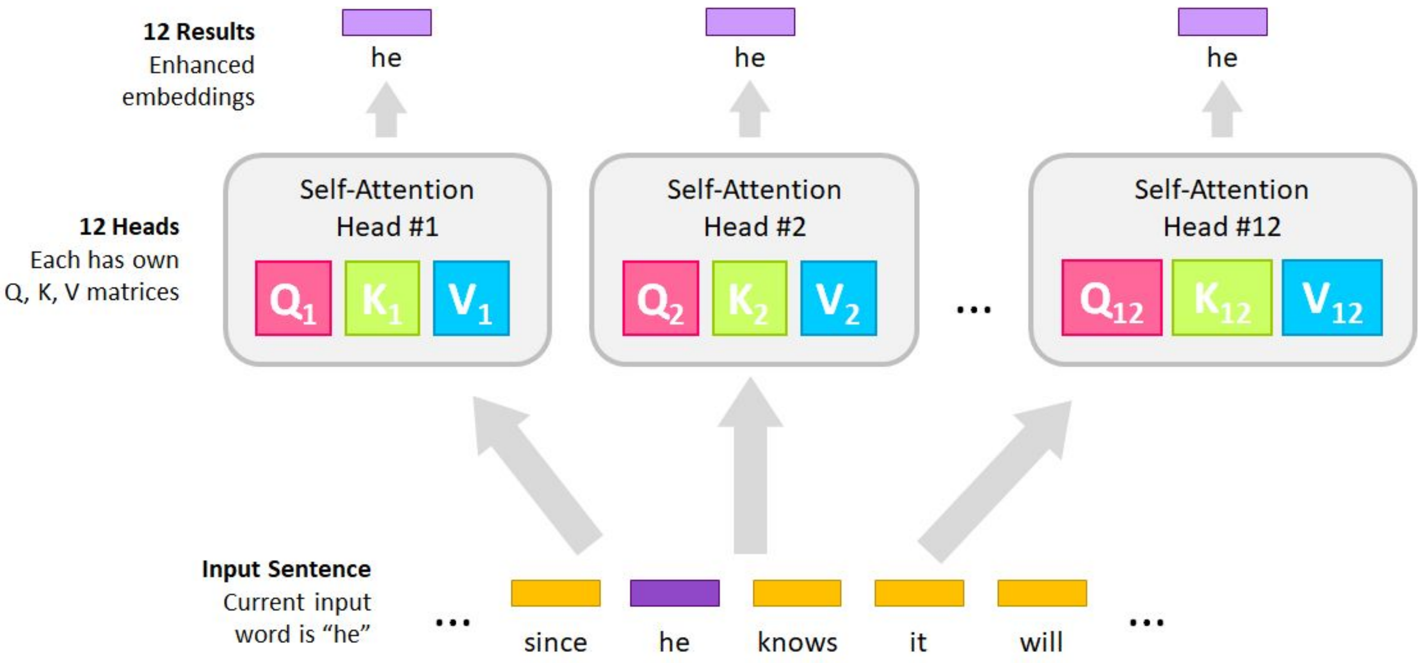
Determining who a pronoun refers to is really just one example of what self-attention may do. For instance, when processing the noun ‘stuff’ at the end of the sentence, it might be helpful for the network to pay attention to any adjectives that are used to describe the ‘stuff’--in this case, the adjective “ridiculous”:



knows it’ll lead to some ridiculous **stuff.**

Ideally, we would like to have multiple instances of the self-attention mechanism, each of which is trained to perform a different function--e.g., one to connect pronouns to their subjects, one to connect nouns to their adjectives, and still more to perform other functions.

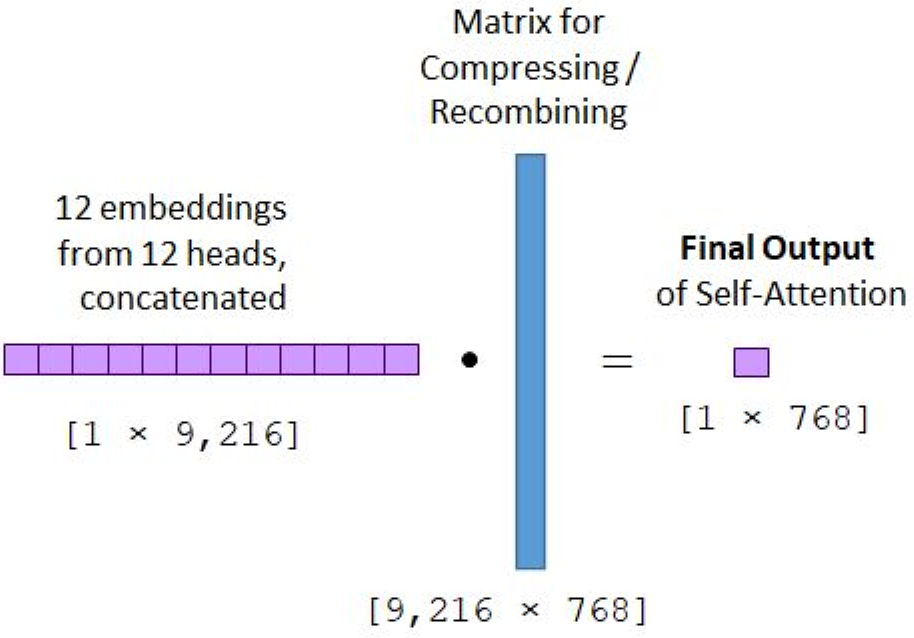
This is called “**multi-headed attention**”. Each instance of self-attention is referred to as an **attention head**, and BERT has 12 heads. What makes these different heads independent is that they each have their own unique instance of the Q, K, and V projection matrices, as illustrated below.



5.1. Re-Combining Attention Head Embeddings

For a single input word, we’ll run it (and its context words) through 12 unique instances of self-attention, but this also means that we’ll end up with 12 unique “enhanced embeddings”!

Each BERT layer produces only a single embedding per input word, so we need to recombine these 12 results into one. This is machine learning, so naturally we accomplish this by adding yet another neural network layer to learn how best to combine and compress these 12 embeddings back down to a single 768-dimension embedding.


Matrix for Compressing / Recombining
12 embeddings from 12 heads, concatenated
•
=
Final Output of Self-Attention
[1 × 9,216]
[1 × 768]
[9,216 × 768]

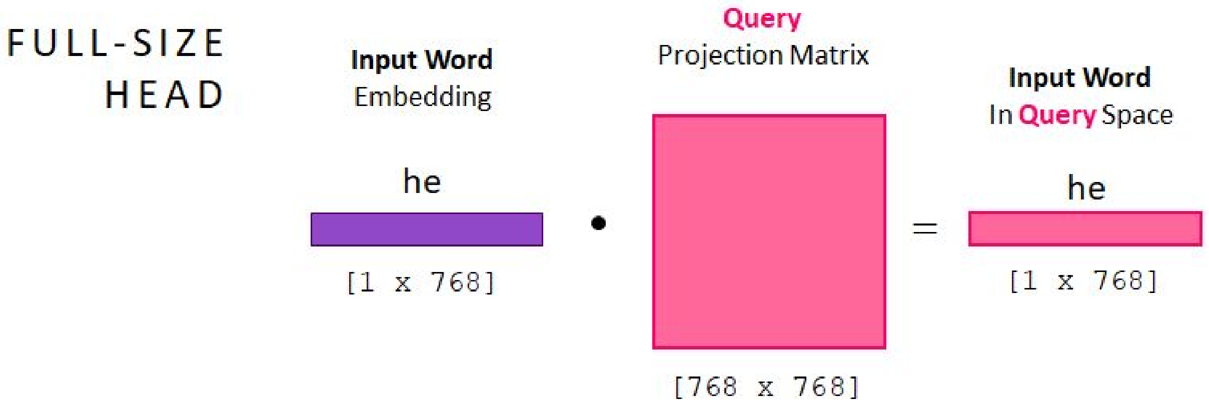
Reducing Embedding Size

By replicating our self-attention unit 12 times, we've also increased the number of weights in our model 12x! And on top of that, we've added the above "recombination" step with its own giant weight matrix.

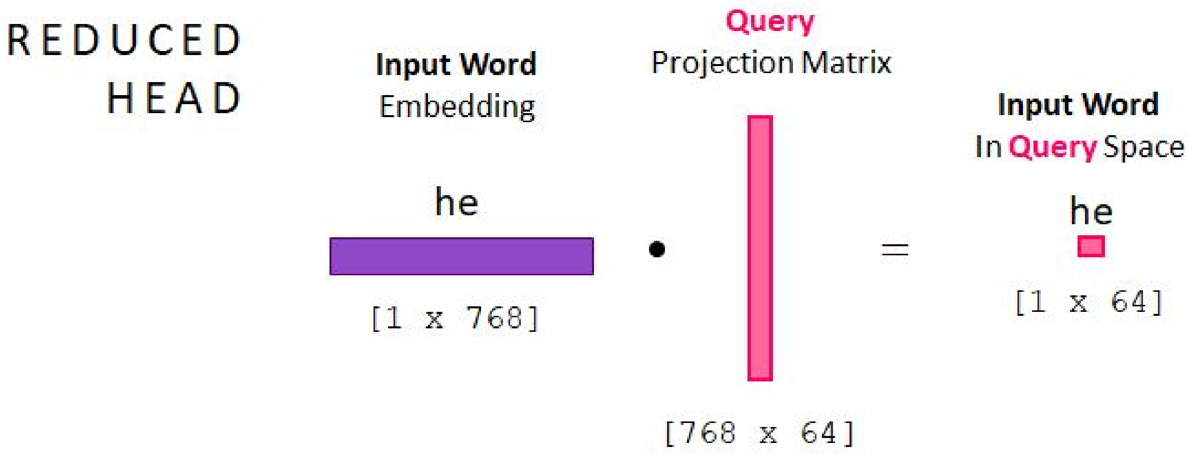
Altogether, this is too many different weights for BERT to learn. To address this, the authors actually scaled down the attention heads to an embedding size of 64 (which, we'll see, is exactly 1/12th of 768…)

Let's see how this works in detail.

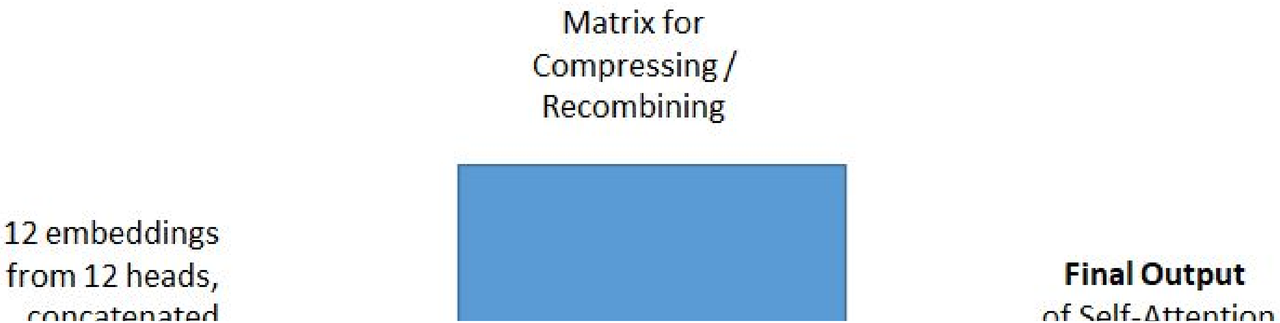
Here's what the projection matrix looked like in our full size attention head:



And now here's what it looks like at the reduced size of 64:



Each of the 12 heads will now produce an enhanced embedding of size 64. We'll still do that recombination step, but now the dimensions are more reasonable:



5.2. Additional Details

If you were to try and re-implement BERT exactly from scratch yourself, then there are a handful of tweaks which I haven't covered, so I'll list them here for completeness.

Deep Neural Network Tricks

BERT employs several common deep neural network techniques:

1. Layer Normalization
2. Residual Connections
3. Drop out

Jay Alammar has a nice illustration of the residuals and layer normalization steps in the section titled The Residuals of his post. You can find tutorials around the web on the purpose and implementation of these techniques, but none of them seem particularly important to me for understanding BERT.

Divide by 8

There is another, similar trick employed during the "scoring" of the context words. We divide each context word's score by 8 before feeding them into the SoftMax function. This apparently produces "more stable gradients" during training. The value 8 is calculated as the square root of the reduced embedding size of 64.

Part 3 - Pre-Training Tasks

The pre-training tasks are both an important part of BERT's contribution and helpful in understanding how BERT works. Perhaps most valuable, though, is that understanding these tasks will make it much easier to read and comprehend the various follow-on projects that have attempted to improve on the original model and training procedure.

In this portion of the book, we'll look at the specifics of these training tasks, and I'll end by attempting to summarize some of the modifications made by more recent models.

6. BERT’s Pre-Training

We’ve already explored the concept of a “Pre-Training Task” in the introduction, and I shared a simple example of the Masked Language Model (MLM) and Next Sentence Prediction (NSP) tasks. In this section, we’ll go into more depth on these tasks, and then review some ways in which researchers have tried to improve upon them.

6.1. Training Set

From the Appendix of the original BERT paper:

> “BERT is trained on the BooksCorpus (800M words) and Wikipedia (2,500M words)”.

With BERT coming from Google, I always just assumed that “BookCorpus” referred to training on Google’s massive “Google Books” library (which you can browse from https://books.google.com). It turns out that’s not the case. **BookCorpus** (the authors of BERT misspelled it as “BooksCorpus”) comes from the following paper:

> *Aligning Books and Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books* (pdf)
>
> First Author: Yukun Zhu, University of Toronto

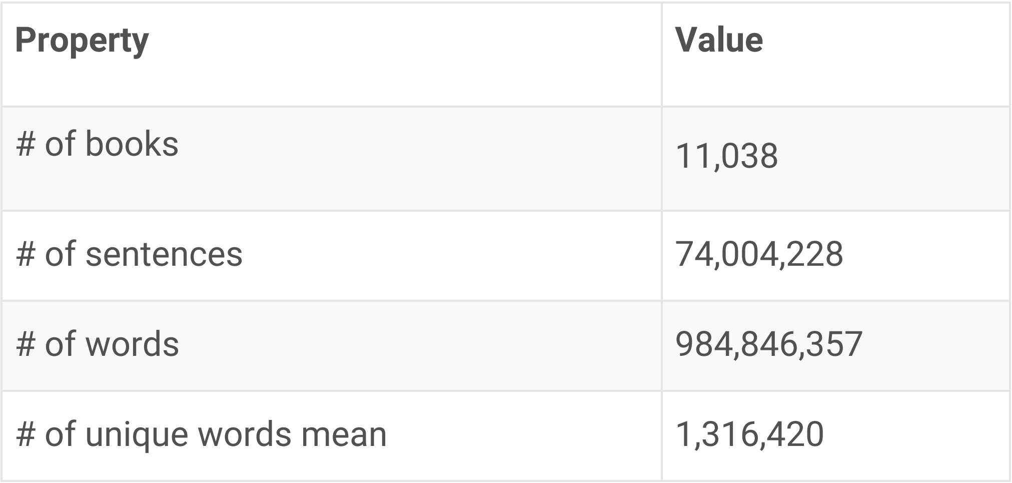
Published: ~2015

Here’s the description of the dataset given by the above paper (emphasis added):

> “**BookCorpus**. In order to train our sentence similarity model we collected a corpus of 11,038 books from the web. These are **free books written by yet unpublished authors.** We only included books that had more than 20K words in order to filter out perhaps noisier shorter stories. The dataset has books in 16 different genres, e.g., Romance (2,865 books), Fantasy (1,479), Science fiction (786), etc. Table 2 highlights the summary statistics of our corpus.”

Again, I was surprised by this, so I did a little more digging and I’ve included my findings below (in case you’re similarly curious).

And here are some stats from Table 2 of that paper.



Property	Value
# of books	11,038
# of sentences	74,004,228
# of words	984,846,357
# of unique words mean	1,316,420

# of words per sentence median	13
# of words per sentence	11

There is a parallel paper by the same authors, *Skip-Thought Vectors* (pdf). It contains a couple small extra details:

- They offer one more category: “Teen (430)”
- “Along with narratives, books contain dialogue, emotion and a wide range of interaction between characters”.

The website for the BookCorpus project is here, but they no longer host or distribute this dataset.

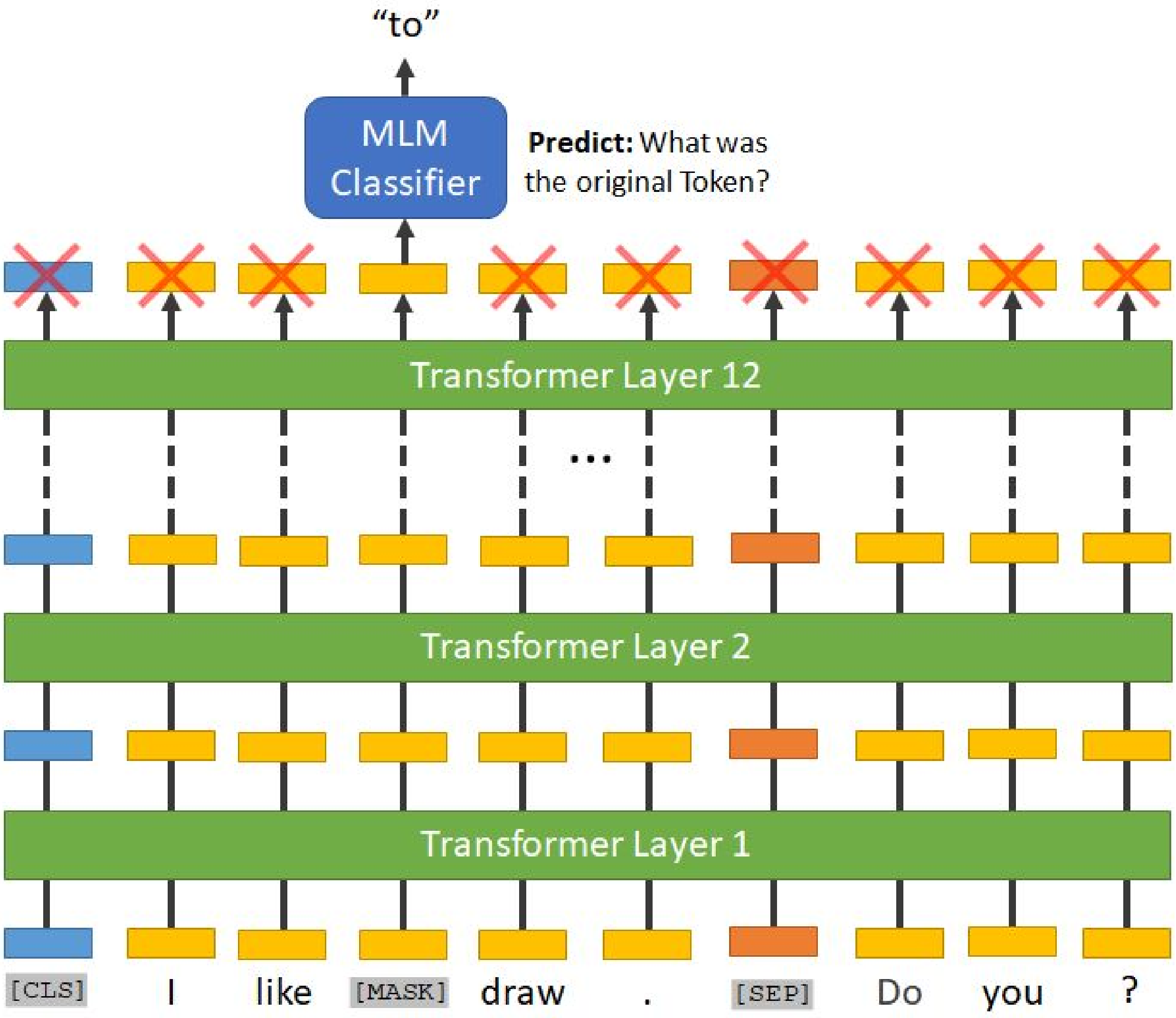
Instead, they say that the text was gathered from this site: https://www.smashwords.com/, and suggest that you gather your own dataset from there. I found a GitHub repo for doing just that here–not a lot of activity, though.

6.2. Masked Language Model (MLM)

To illustrate these pre-training tasks (both MLM and NSP), we’ll use these two short sentences as an example: “I like to draw. Do you?”

In the MLM pre-training task, BERT was fed all of the text from Wikipedia and the BookCorpus, but 12% of the tokens were randomly chosen to be “masked out” (replaced by the special [MASK] token), and BERT was trained to “fill-in-the-blanks” (predict the missing tokens).

Let’s say we masked out the word “to” from our example sentence, as shown below.

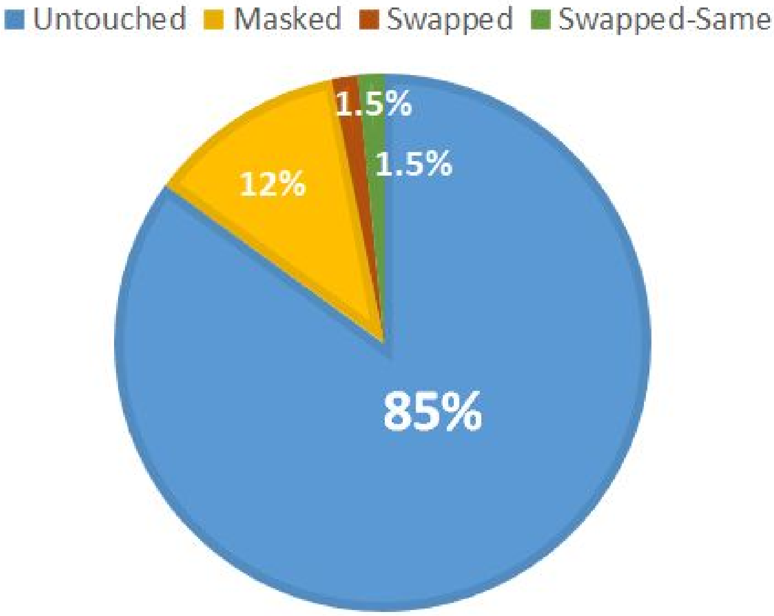


The whole input sequence is run through BERT, but notice on the output that we don’t do anything with the output embeddings except for the masked one. The masked token is fed into a simple one-layer

SoftMax Regression classifier, which outputs a probability for every token in BERT's vocabulary. Whichever word has the highest probability is the one that BERT would pick.

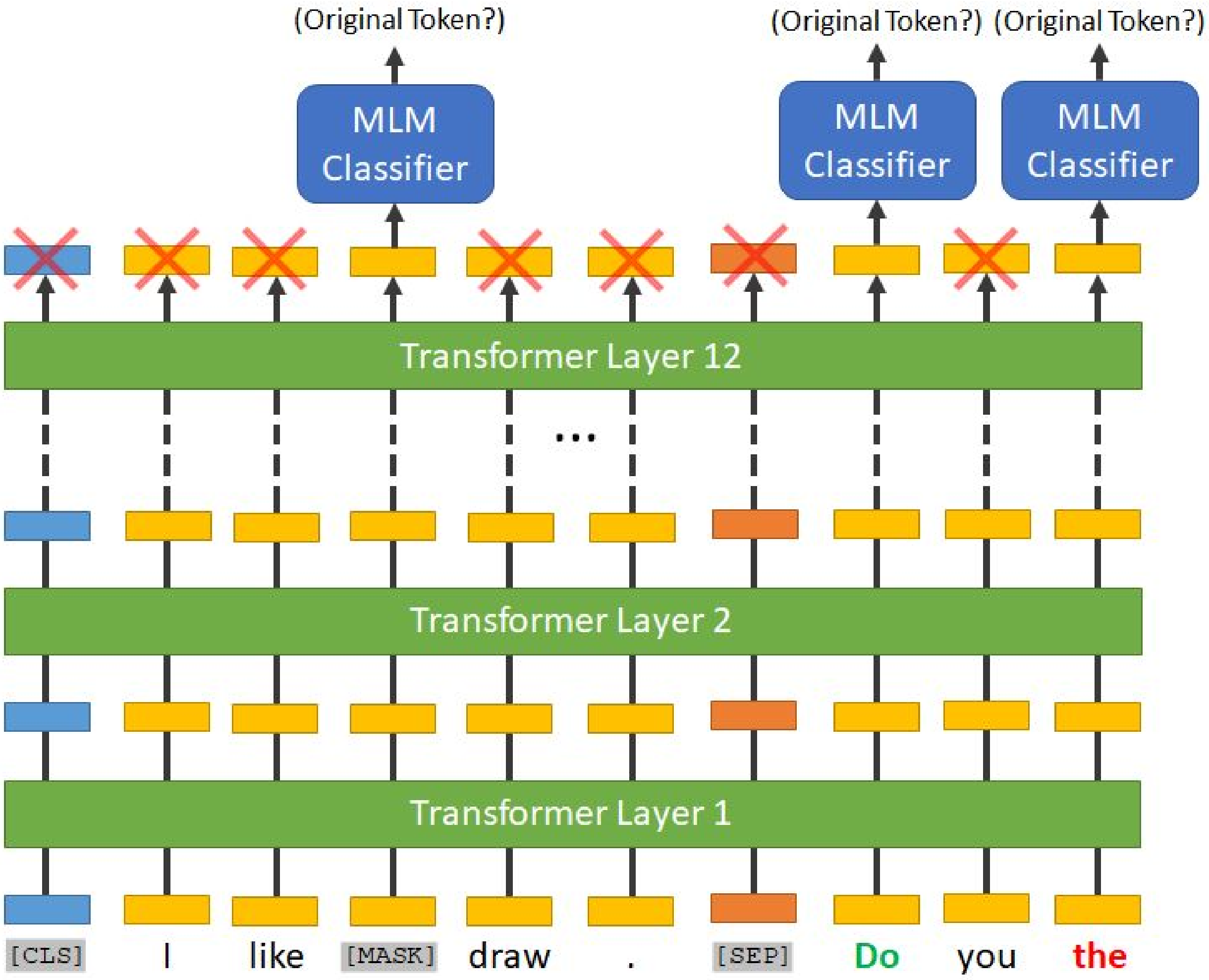
In addition to masking 12% of the tokens, they:

- Replaced another 1.5% of the tokens with a random token.
- Marked another 1.5% of the tokens for prediction, but didn't change these tokens.



Let's add these other two modifications to our example. Let's say that

1. The token for '?' was selected to be replaced by a random token, so it's been replaced by 'the'.
2. The token for 'Do' was selected to be predicted but unchanged.



Even though the word “Do” hasn’t been changed, BERT doesn’t know that (it could have been swapped!), and it still has to decide what token most likely belongs in that spot.

Why did they make these other two types of substitutions? The [MASK] token tells us *nothing* about the meaning of the word it replaced--BERT has to infer what word was there *purely by looking at the context*. This might give BERT a tendency to ignore the input embedding and infer everything from its context, which we wouldn’t want it to do in any of our *real* applications.

The authors landed on these proportions (12%, 1.5%, 1.5%) through experimentation, and the results of their experiments are in the appendix of their paper, section C.2.

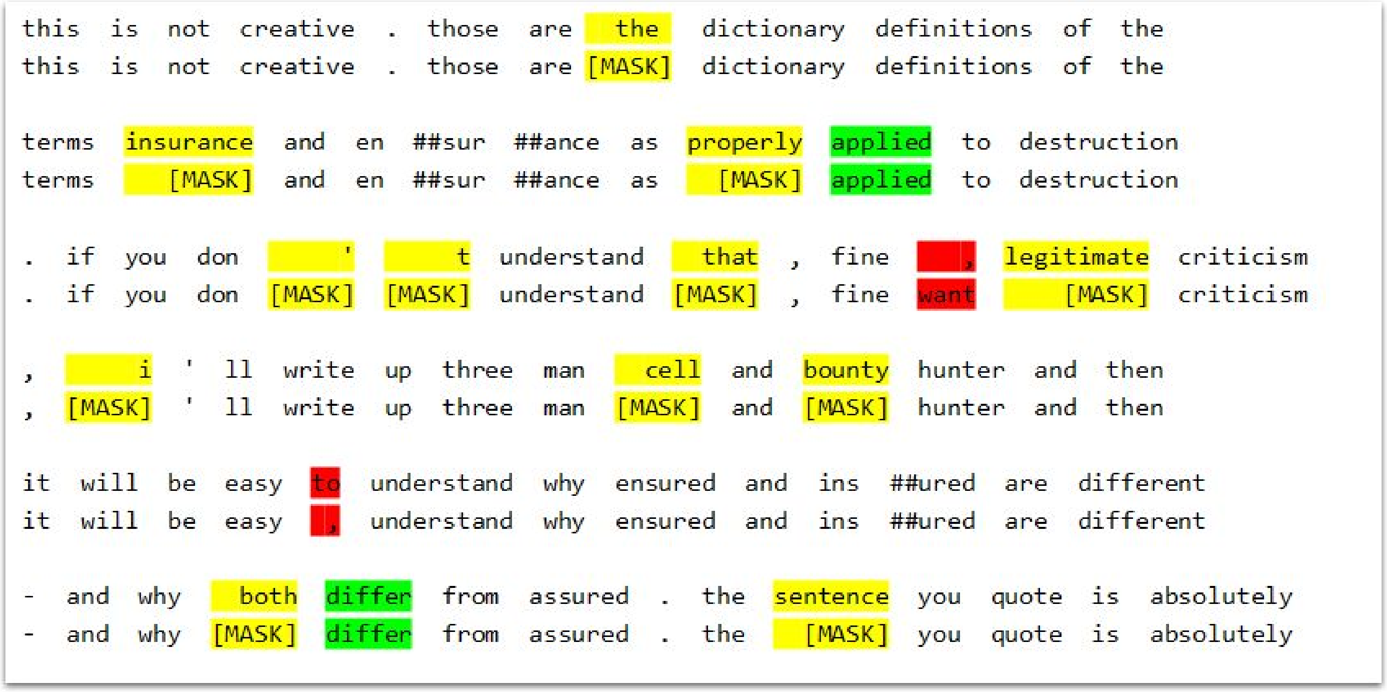
Side Note: The authors present the percentages as:

- *15% of all tokens masked, with:*
    - *80% replaced by [MASK]*
    - *20% replaced by a random token*
    - *20% replaced by the same token*

If you multiply these out, you get the overall percentages of 12%, 1.5%, and 1.5% that I showed above.

Example of Mangled Text

To better visualize the masking, here is a piece of text with the substitutions made in the correct proportions. For each pair of lines, the first is the original text, and the second is the modified version.

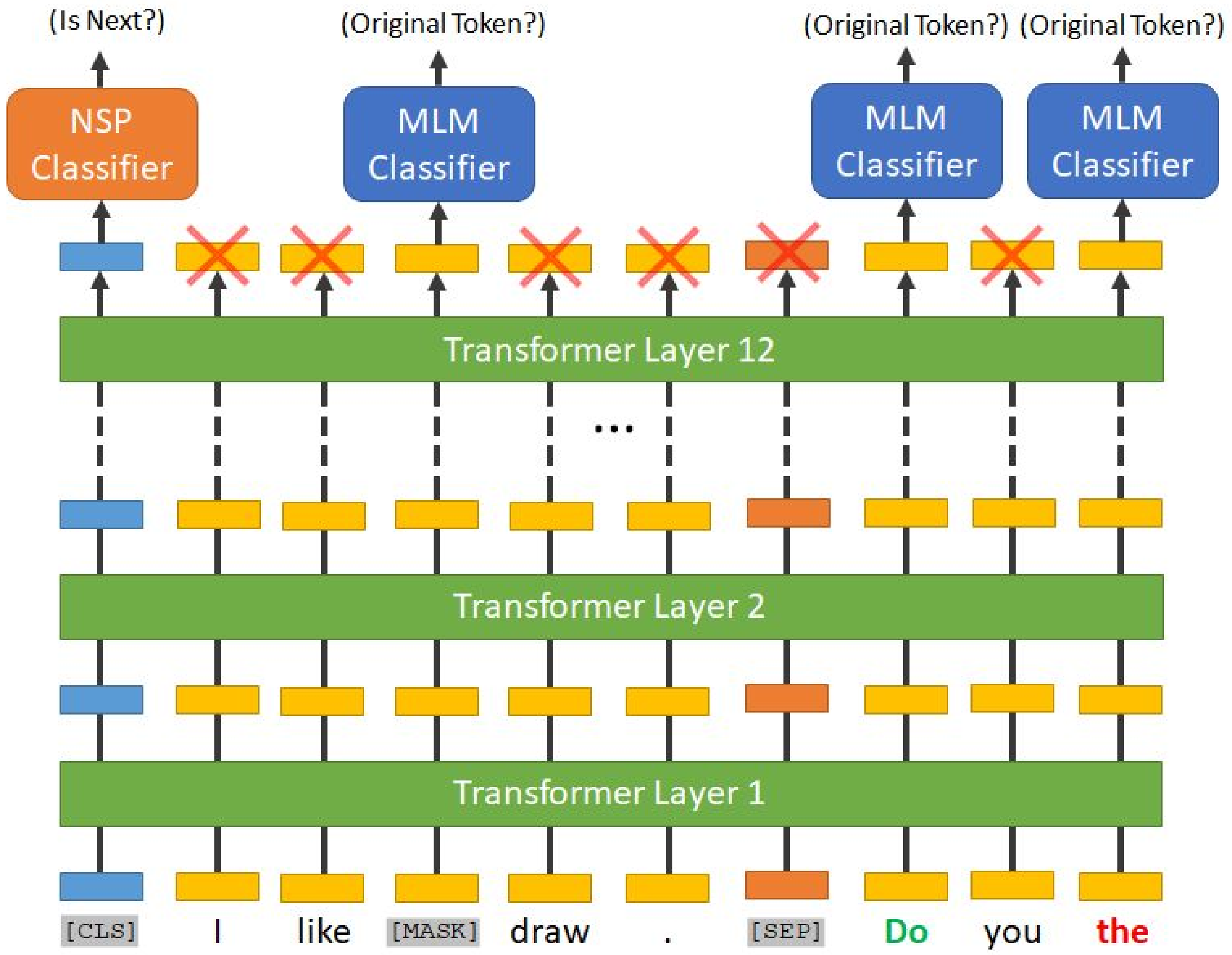


```
this   is   not   creative  .  those   are   the    dictionary  definitions  of   the
this   is   not   creative  .  those   are  [MASK]  dictionary  definitions  of   the

terms  insurance  and  en  ##sur  ##ance  as  properly  applied  to  destruction
terms     [MASK]  and  en  ##sur  ##ance  as    [MASK]  applied  to  destruction

.  if  you  don       '        t  understand    that  ,  fine     ,  legitimate  criticism
.  if  you  don  [MASK]  [MASK]  understand  [MASK]  ,  fine  want      [MASK]  criticism

,       i  '  ll  write  up  three  man    cell  and  bounty  hunter  and  then
,  [MASK]  '  ll  write  up  three  man  [MASK]  and  [MASK]  hunter  and  then

it  will  be  easy  to  understand  why  ensured  and  ins  ##ured  are  different
it  will  be  easy  ,   understand  why  ensured  and  ins  ##ured  are  different

-  and  why    both  differ  from  assured  .  the  sentence  you  quote  is  absolutely
-  and  why  [MASK]  differ  from  assured  .  the    [MASK]  you  quote  is  absolutely
```

6.3. Next Sentence Prediction (NSP)

In the “Next Sentence Prediction” task, BERT was fed *pairs* of text passages. Half of the time, the second passage immediately follows the first, and the other half of the time, the second passage is just a randomly selected piece of text from elsewhere in the training text.

These two passages are separated by the special [SEP] token, and BERT uses the [CLS] token to do the prediction. The only change to the illustration below is the addition of a binary classifier (in orange) to the CLS token.



NSP Uses Text Spans

The name "Next Sentence Prediction" is misleading, because the training samples aren't actually pairs of *sentences*. From section A.2. of the paper's appendix:

> "To generate each training input sequence, we sample two spans of text from the corpus, which we refer to as "sentences" even though they are typically much longer than single

> sentences (but can be shorter also)…They are sampled such that the combined length is ≤ 512 tokens.”

MLM and NSP are Trained Concurrently

MLM and NSP are actually trained *concurrently*, as illustrated above. One training sample consists of a chunk of text, with all of these various substitutions made, and this results in a number of different predictions that need to be made (1 NSP prediction, and a variable number of MLM predictions). To combine all of these predictions into a single training sample, you simply combine the losses of all of the predictions.

From the paper, Appendix A.2: “The training loss is the sum of the mean masked LM likelihood and the mean next sentence prediction likelihood.”

7. Training Task Improvements

Projects attempting to improve upon BERT have had a number of objectives, such as:

1. Reducing the size of the BERT model (for faster and cheaper inferencing, with reduced accuracy).
2. Improving support for non-English languages.
3. Reducing the cost of pre-training BERT.
4. Out-performing BERT in accuracy on NLP benchmarks.

In this chapter, I wanted to share an overview of some of the innovations targeted at number 4.

Number 4 seems the most pertinent to this eBook, since you might assume that if a more recent model achieves higher accuracy than the original BERT, that you might as well switch to it. This reasoning highlights an important problem with BERT research, though…

The Problem with Accuracy-Based Benchmarks

Many papers have come out since BERT which have modified the MLM and NSP tasks in various ways, including XLNet, RoBERTa, and ALBERT.

A challenge with these developments, however, is that the objectives of these research projects may not align with your own more practical interests. A primary motivation in these projects is to *outperform BERT's accuracy on the benchmarks*. This typically means increasing the size of the model, and making modifications to the training regimen to "scale successfully"--that is, to allow the model to increase in accuracy without over-fitting.

ALBERT, which also came out of Google, achieved higher benchmark scores than BERT, XLNet, and RoBERTa, *but only at its largest size*, "ALBERT-xxlarge". ALBERT-base, for instance, generally performs *the same or worse* than BERT-base!

I only mention this to caution you against automatically assuming that the latest and greatest model will be better for your application than the original BERT.

Here is a survey of some of the developments.

7.1. Whole-Word Masking

The original MLM masking procedure did not give any special consideration to sub word pieces. For example, BERT breaks the word “embeddings” into the pieces ['em', '##bed', '##ding', '##s'] and the masking procedure might only mask out one of these subwords. The remaining subwords can make it easier for BERT to predict the missing token--and that’s a bad thing, we want the task to be harder to force BERT to get smarter.

After their paper was published, the BERT authors trained and uploaded a variant of BERT with “whole-word masking”, where, if one of the subwords from “embeddings” was selected, then *all* of the subword tokens would be replaced.

You can read more in the README file of the original BERT repository on GitHub here.

N-gram Masking

ALBERT took whole-word masking a step further, and would mask out between 1 to 3 whole words to further increase the difficulty.

> ALBERT was first submitted September 26th, 2019

7.2. Replacing NSP

The authors of RoBERTa and XLNet found the NSP task to be unreliable, and actually chose to remove it completely from their pre-training. Table 5 of the BERT paper shows the results of their own experiments with removing NSP.

XLNet (Carnegie Melon & Google) submitted June 19th, 2019

RoBERTa (Univ. of Washington & Facebook) submitted July 26th, 2019

Sentence Order Prediction (SOP)

The authors of ALBERT replaced NSP with a subtle variation called Sentence Order Prediction. In SOP, the two passages of text are always consecutive, but sometimes they are presented in reverse order. BERT / ALBERT is tasked with predicting whether the two passages are in the correct order or not.

The argument for SOP is that NSP can often be solved by comparing the topic of the two passages (if they're on completely different subjects, then it's unlikely that the second passage followed the first). SOP, instead, requires a deeper understanding of what's being communicated.

7.3. Plausible Substitutions

A recent model from Google called ELECTRA made a substantial change to MLM. Instead of masking tokens, they used a "Generator Network" to replace a portion of the words with a "plausible alternative", then tasked BERT with predicting, *for every input token*, whether it had been swapped or not.

ELECTRA is particularly notable for claiming to outperform BERT in its smaller configurations (i.e., BERT-base), and not just at the larger scales.

Appendix

A.1. Revision History

Revision	Date	Notes
v1.0.0	May 13, 2020	● Initial release!