BERT for Sequence Classification from Scratch — Code and Theory

AR · Following
16 min read · Aug 3

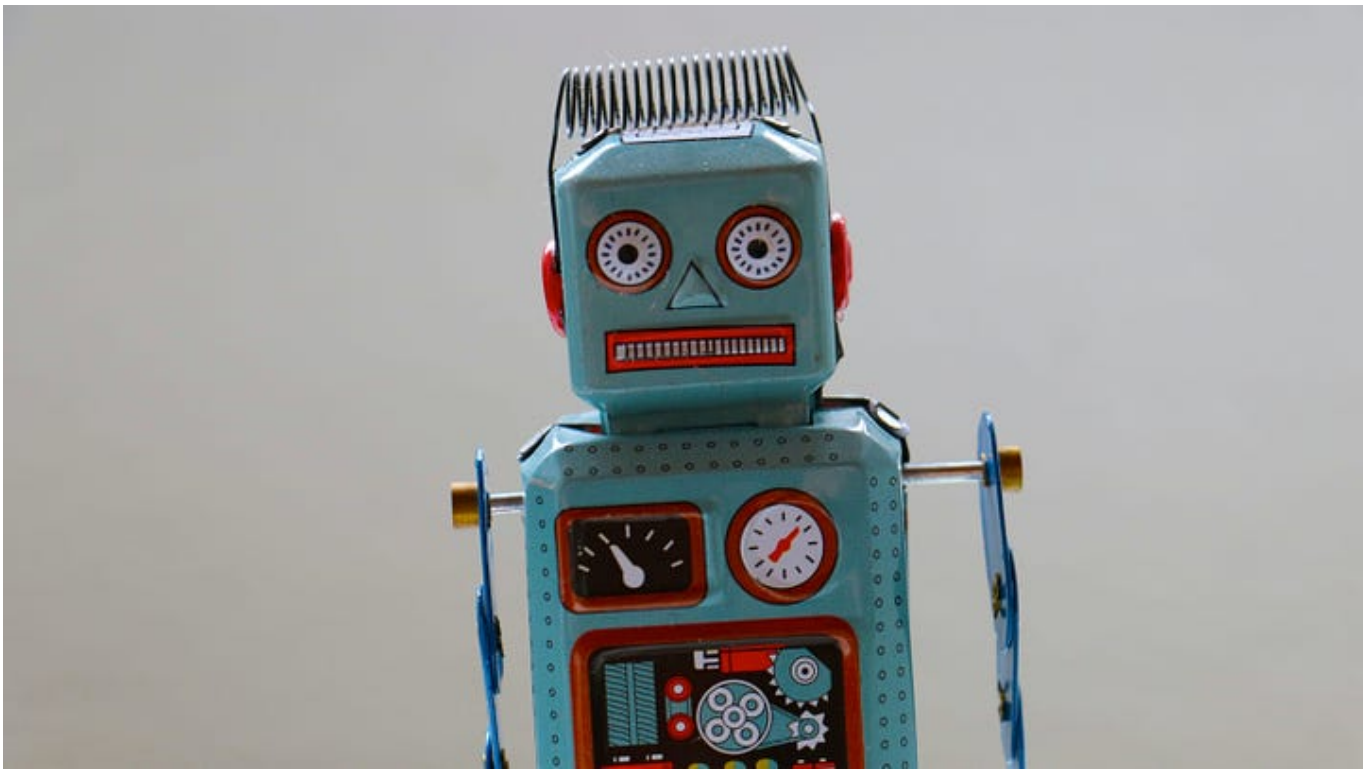


Photo by Rock'n Roll Monkey on Unsplash

Coding BERT for Sequence Classification from scratch serves as an exercise to better understand the transformer architecture in general and the Hugging Face (HF) implementation in specific.

Some of the non-critical features found in the HF repository have been left out for simplicity and readability. However, our model should function just the same as the HF core model and the weights will be compatible between the two.

This means that after fine-tuning our model (in Part 3 — Coming Soon), we will be able to copy our weights into a HF instance to take advantage of everything the HF ecosystem has to offer.

There will be lots of fine details explaining the reasoning behind design choices made in BERT.

Before proceeding I suggest opening this diagram in another tab for convenient reference. It was created by George Mihaila and outlines the specific architecture found in the HF version of BERT for Sequence Classification.

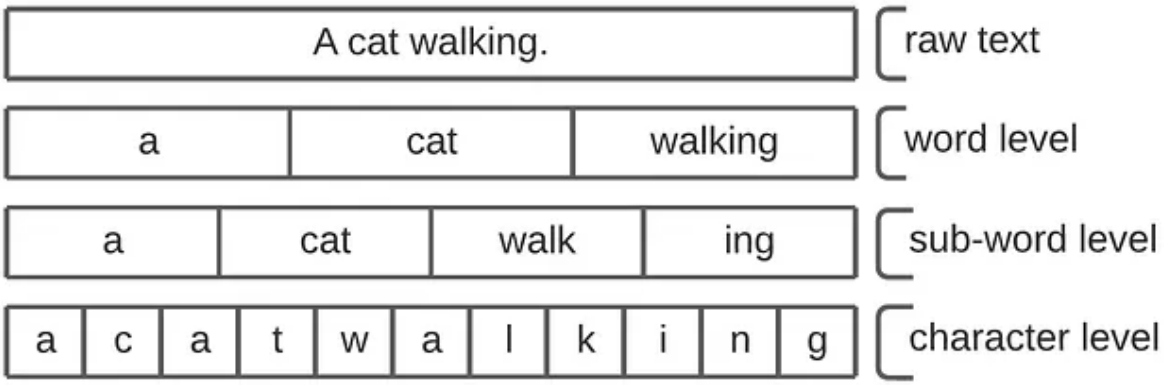
All code for this project can be accessed on GitHub.

Tokenization

Before we get into the model itself we probably need to talk a little about tokenization.

Tokenizers serve two key functions:

1. **Splitting the raw text into smaller parts** — called tokens — at either the word, subword, or character level.
2. **Converting the tokens into a numerical format** a machine learning model can understand.



Tokenization of a sequence at the word, subword, and character level.

A **subword tokenization algorithm** is used in the **BERT tokenizer.** Splitting text into subword units strikes a nice balance between vocabulary size and sequence length. It also better handles rare and out-of-vocabulary words reducing the need to treat them as unknown tokens.

WordPiece is the specific subword tokenization algorithm used in the BERT tokenizer, and it is designed to build a vocabulary of tokens to a defined size through a process of selecting and merging the highest scoring character pairs in the text dataset.

This scoring is the ratio of the frequency of the pair to the product of the frequency of the individual elements.

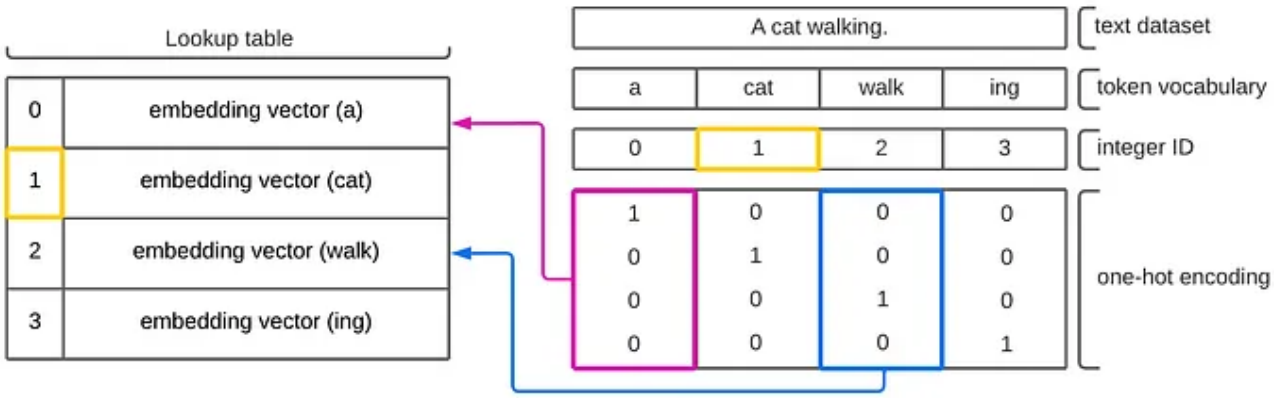
score = (freq_of_pair) / (freq_of_first_element × freq_of_second_element)

For a full understanding of the algorithm, you can read more here.

However, it is not too important for our purposes, as we will be using instances of the **Hugging Face BERT Tokenizer** that are already trained.

Text passed to the BERT tokenizer gets broken down into a sequence of tokens. Each unique token in the vocabulary is assigned an integer ID. This acts as a shorthand or abstraction of a one-hot encoded vector.

One-hot encoded vectors are just vectors where the element at the index corresponding to the token being represented has a value of one and all other elements have a value of zero. The length of the vector is equal to the number of tokens in the vocabulary of the text dataset.

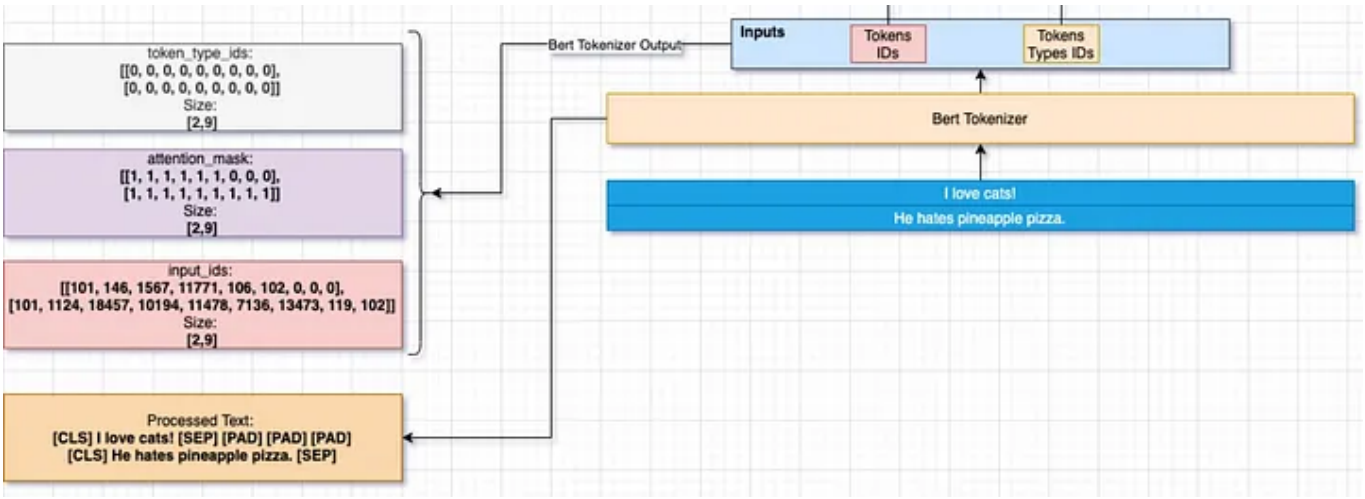


Integer ID and one-hot encoding mapping to the embedding vectors in the lookup table.

Both one-hot encoded vectors and integer IDs serve the same purpose of selecting the corresponding row from a lookup table.

The lookup table is essentially a matrix of trainable embeddings, which we will talk about shortly.

At this point we know enough now about tokenization where the bottom portion of the BERT diagram should start to make sense.



Input and outputs of the BERT Tokenizer. Diagram by George Mihaila.

This diagram shows a batch of two samples of raw text being passed to the tokenizer that outputs the processed text (string tokens) along with three different integer encodings.

Notice that the processed text has three special tokens:

1. **Classification token:** The [CLS] token is placed at the very beginning of a sequence. It can serve as an aggregate representation for the whole sequence.
2. **Separation token:** The [SEP] token is placed at the end of every sentence in a sequence. This makes it possible for BERT to differentiate between sentences and understand sentence boundaries.
3. **Padding token:** The [PAD] token is placed at the end of the sequence — as many as are needed — to extend the length of a sequence so that all sequences in the batch are of the same length. This is a requirement of

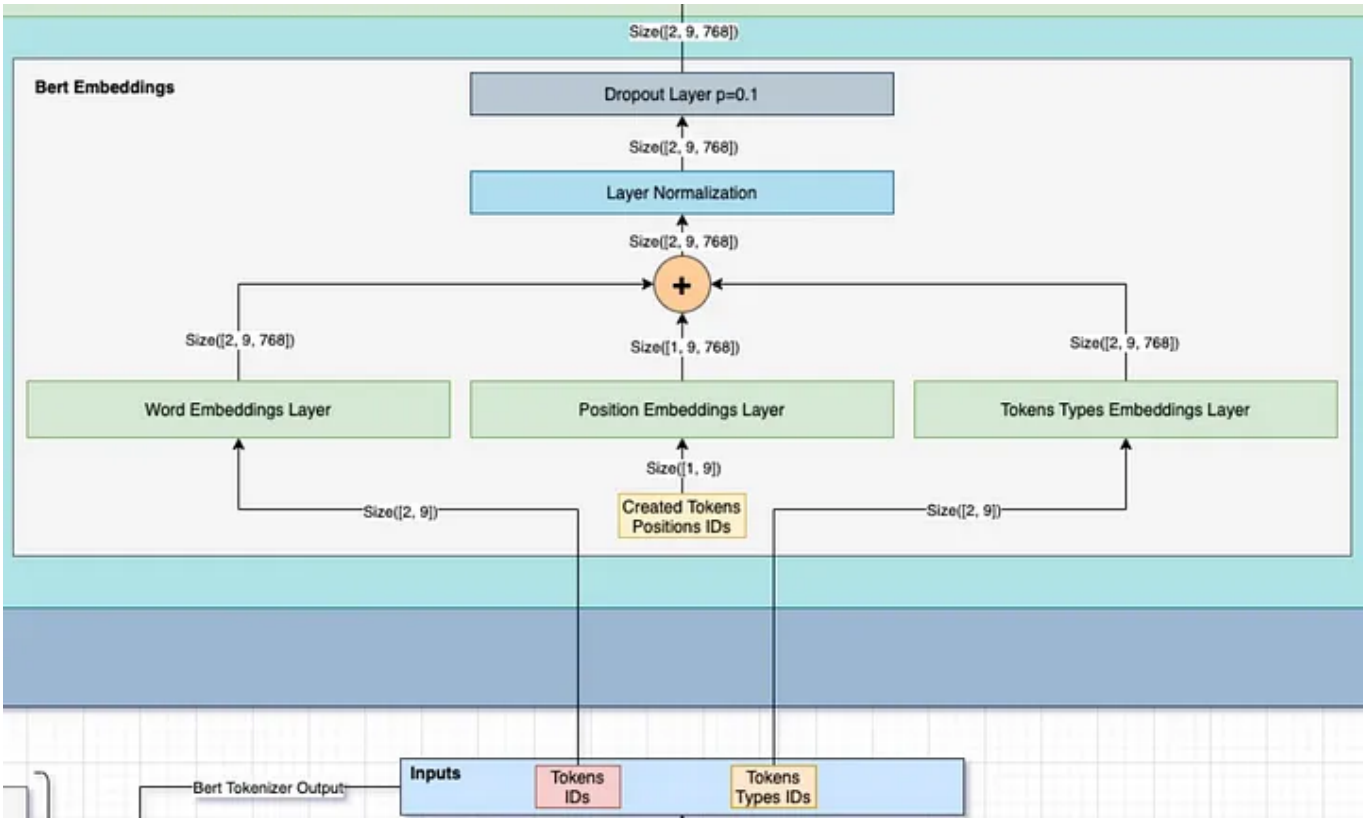
matrix math so that multiple samples (a batch) can be processed in parallel.

The three types of integer encodings that are based on the processed text are as follows:

1. **Input IDs:** The BERT base model has a defined vocabulary of 30522 tokens. Each unique string token maps to a unique integer ID. These are the integer IDs for each token in a sequence.
2. **Attention Mask:** Sequences that are shorter than the defined maximum sequence length are extended with padding tokens. The padding tokens are just placeholders and should not influence the output of the model. The attention mask distinguishes the padding tokens from all the rest. The elements at the indices of non-padding tokens receive a value of one and the elements at the indices of the padding tokens receive a zero.
3. **Token Type IDs:** Next-sentence prediction is a specialized task carried out when pretraining a BERT model. The sequence inputs are constructed to be composed of two sentences. The second sentence is either the sentence immediately following the first sentence from the source text, or it is a random sentence from the source text. The objective is to predict whether the second sentence is indeed the next sentence or a random sentence. Token type IDs designate which tokens in the sequence belong to the first sentence and which tokens belong to the second sentence by assigning zeros and then ones, respectively. Since we are not doing next-sentence prediction, but rather something called sequence classification, all tokens in the sequence are treated the same and encoded with a zero. The token type IDs have no influence on the model output when doing sequence classification.

Embedding Module

If we continue moving up the diagram, we see the encodings from the tokenizer are passed to the Embedding module. It is here that the input IDs and token type IDs are used to lookup their corresponding embedding.



BERT Embedding Module. Diagram by George Mihaila.

So what exactly are embeddings?

Embeddings are vector representations of tokens, which allow us to essentially organize them in space.

They are kind of like coordinates for the tokens. Ideally tokens of similar meaning are located closer together and tokens of dissimilar meaning are further apart.

The length of the embedding vector is a design choice. It determines the dimensionality of the space in which the embeddings reside. The larger the space, the more room to organize the tokens.

In practice embedding lengths are set relatively high to accommodate large vocabularies and the complexities of contextual meaning. BERT base has an embedding length of 768.

The following three embeddings are generated for each token (and then added together):

1. **Word Embeddings**: Each input ID is replaced with the corresponding row vector from the word embedding lookup table. This lookup table will have one row for each unique token in the vocabulary. BERT base has a vocabulary size of 30522.
2. **Token Type Embeddings**: Same principle as the word embeddings except there are only two rows to this lookup table corresponding to which sentence (the first or second) the token is located. Remember this is only relevant for the pretraining task of next-sentence prediction. Otherwise, it has no influence on the model output.
3. **Positional Embeddings**: Positional embeddings serve to provide the model with some sense of the order of the tokens in the sequence. The positional IDs used to retrieve the corresponding positional embeddings from the lookup table will be the same for each sequence — a vector of consecutive integers from 0 to n, where n is the maximum sequence length. BERT base has a max sequence length of 512.

Adding all three of these embeddings together encodes the various aspects of information into a single embedding.

The embedding now contains information about the semantic meaning of the tokens in the sequence, the element-wise position of the tokens in the sequence, and the sentence in which the tokens are located (only relevant for next-sentence prediction).

All of these embeddings are learned during the pretraining phase on large corpora of text and can be further fine-tuned for downstream tasks.

Let's now view the code for the embedding module to bring everything discussed into view.

Embedding module.

There isn't anything too tricky going on here so we will just highlight a few points of note:

- The **config argument** is an object that stores various parameters and settings for the model. It gets passed from module to module throughout the network. It just provides a convenient way of storing and retrieving the configuration details of BERT [line 2].
- **The padding token ID is supplied to the word embedding lookup table to set the corresponding row to all zeros and freeze it there.** The padding embedding is just a place holder and should not influence the model output. Likewise, there is no need to waste computational resources training it [line 5].
- **The position IDs are stored as a buffer**, which in PyTorch is essentially a non-trainable parameter or a value with a persistent state. The position IDs are the same for each sequence (0 to 511 if max sequence length is 512) [line 12].
- **Layer normalization** and **dropout** are applied. These are common operations found in many parts of the model [line 20–21].
- **Layer norm:** Normalizes each embedding — across the length of the embedding- independent of the other embeddings in the sequence and the batch. This helps deal with the vanishing/exploding gradient problem

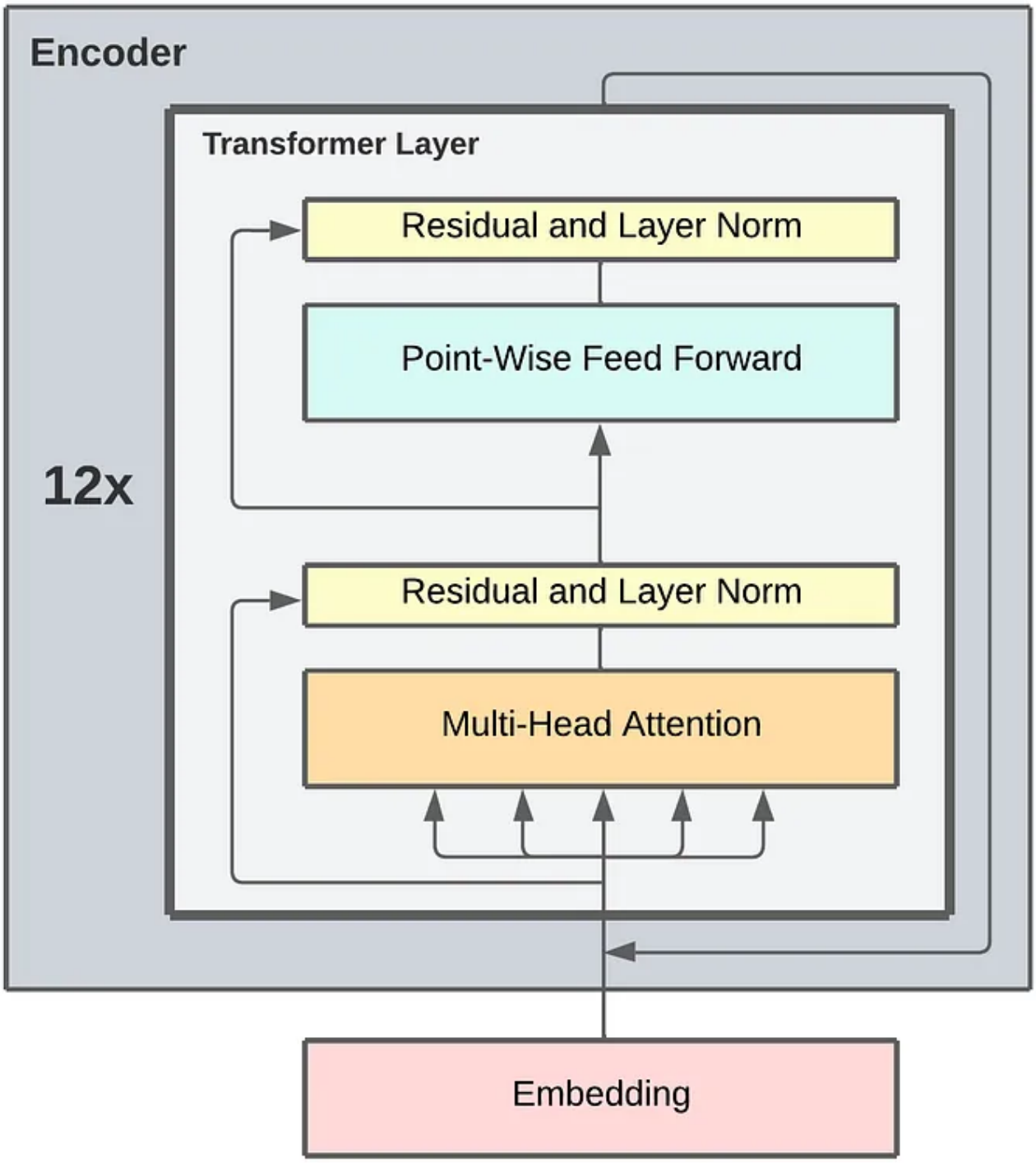
and helps keep the scale of the embeddings from differing too much between layers, which leads to more stable training.

- **Dropout:** A regularization technique that helps prevent overfitting by zeroing out each element in an embedding with some probability (p=0.1 in the case of BERT). This helps the model not become too reliant on any one part of the embedding. In turn the model should be more robust and generalize better to new data.

Encoder

Now that we have our embeddings we will move on to the Encoder, which in the BERT base model has 12 transformer layers (referred to as Bert Layers in this diagram).

The layers are identical to one another, and the output from one layer feeds into the next in a sequential fashion.



BERT Encoder composed of 12 transformer layers — each having multi-headed self attention and point-wise feed forward network.

More layers in the encoder generally allows the model to learn more complex patterns and relationships in the text.

These might be local and syntactic relationships in the lower layers, sentence-level semantics in the middle layers, and document-level relationships in the upper layers.

Each of these layers is composed of two main sub-layers — **multi-headed self-attention** and the **position-wise feed-forward network**.

Multi-Headed Self-Attention

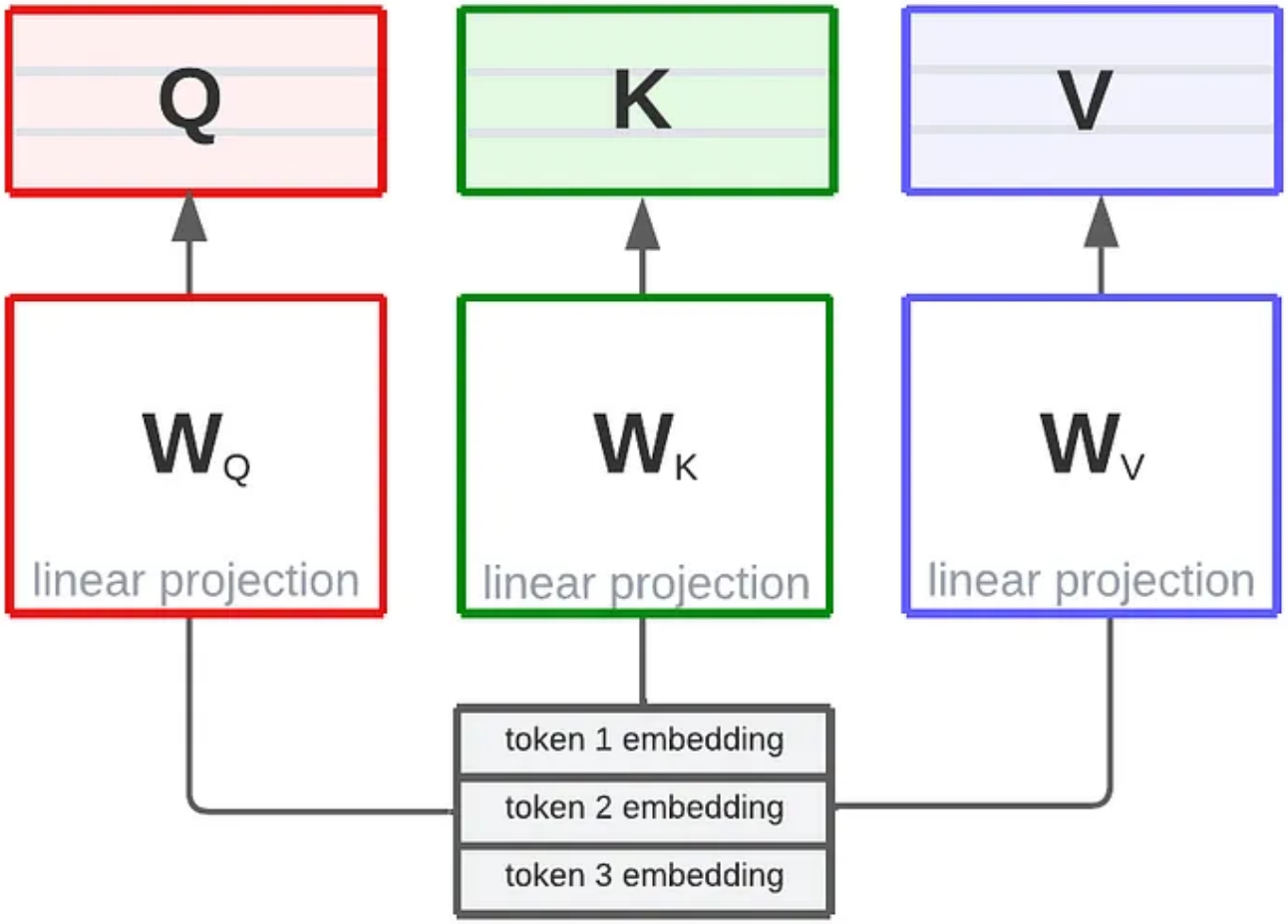
The **self-attention** mechanism is the star of the show as it allows the model to **weigh the importance of each word in the context of all other words** in the sequence. By this process the embedding vector for each token becomes enriched with information on how it relates to the surrounding tokens.

Note that before self-attention is carried out, every token with the same ID will have the same embedding regardless of what the other tokens are in the sequence. They may differ based on their position in the sequence (the position embedding component) but that is it. As English speaking humans we know a word like “bark” has a completely different meaning in the context of a tree compared to the context of a dog. Self-attention can handle such relationships — in addition to more complex and higher-order relationships — by comparing each token in the sequence with all the others, and then enriching each token embedding vector with relevant relational and contextual information.

So how exactly is this contextual enrichment accomplished?

The embedding vector for each token is run through a linear layer that maintains its dimension. So, in goes an embedding vector and out comes

another vector of the same shape. In fact, this is done independently three separate times. So, in goes the embedding vector to each of the three linear layers, and out comes three new vectors.

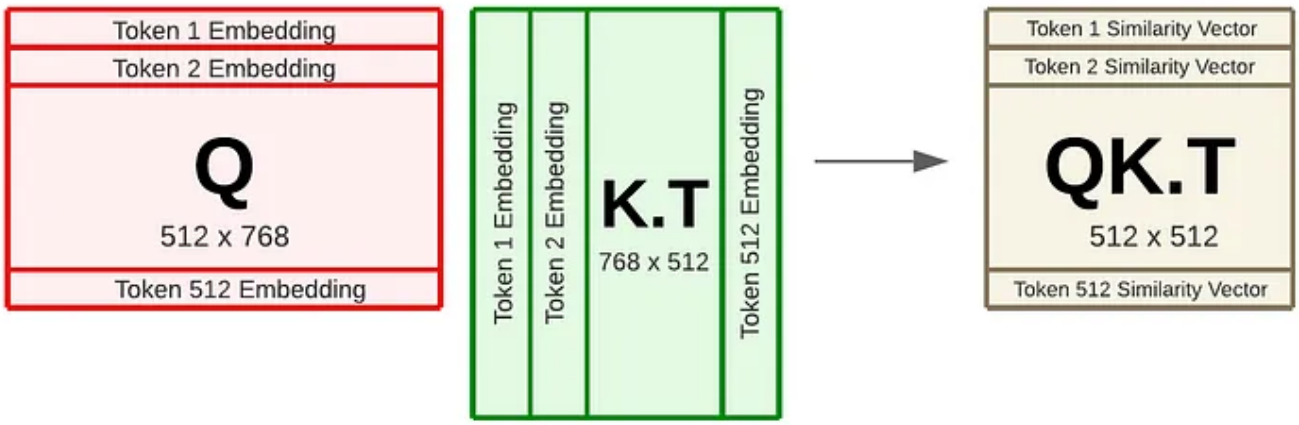


Projecting token embeddings into Query, Key, and Value vectors through matrix multiplication with their respective Weights (W).

These three vectors are referred to as **query**, **key**, and **value** vectors. Since there are multiple tokens in a sequence, we will have query, key, and value matrices (Q, K, V) where each row corresponds to a vector for a token.

The dot product is used as a measure of similarity between two vectors. The dot product yields a scalar, and the larger the scaler value the more similar the two vectors.

Matrix multiplication allows us to take the dot product of each token with all other tokens in one fell swoop. Just note that the key matrix (K) must be transposed (K.T) due to the rules of matrix multiplication. Each row vector of the resulting QK.T matrix corresponds to the same row vector (token) of the query matrix (Q). And each element in a row vector of the QK.T matrix tells us how similar that corresponding token in Q is to each token of the same sequence in K.



The Query (Q) matrix multiplied with the Key (K.T) resulting in the matrix of similarity of scores (QK.T).

Now a few operations are applied to the similarity scores:

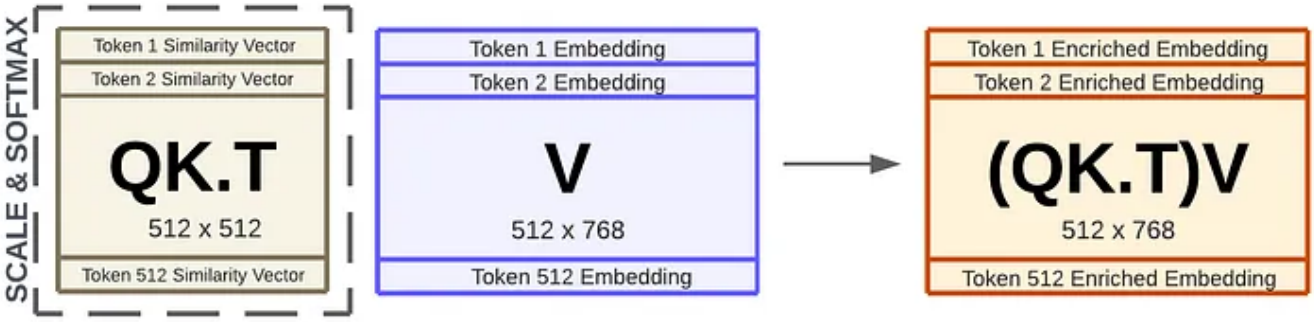
1. The similarity scores are divided by the square root of the dimension size of the vectors in K.

2. The similarity scores involving a padding token are set to negative infinity.
3. The softmax function is applied to the similarity scores along the row dimension to get a probability distribution of weights that add to one.

Scaling down the similarity scores before applying the softmax function smooths out the distribution and helps prevent the gradient from vanishing during backpropagation.

The similarity scores for the padding tokens are set to negative infinity because we do not want any “attention” paid to them.

After applying the softmax function, the padding tokens will receive a weight of zero.



The scaled and soft similarity scores matrix multiplied with the values (V) resulting in enriched embeddings.

The scaled and softmaxed QK.T weight matrix is then matrix multiplied with the value matrix (V). This results in token embeddings that are enriched by way of being a weighted average of all the token vectors in V.

This weighted average is to what degree a token is paying attention to the other tokens in the sequence.

Let's now look at the code for self-attention to see how it is implemented.

The self-attention mechanism is built from three classes:

- **BertSelfAttention**: Responsible for implementing self-attention [line 1].
- **BertSelfOutput**: Applies various operations to the output of BertSelfAttention [line 36].
- **BertAttention**: Houses the two modules above allowing for their execution in sequence [line 52].

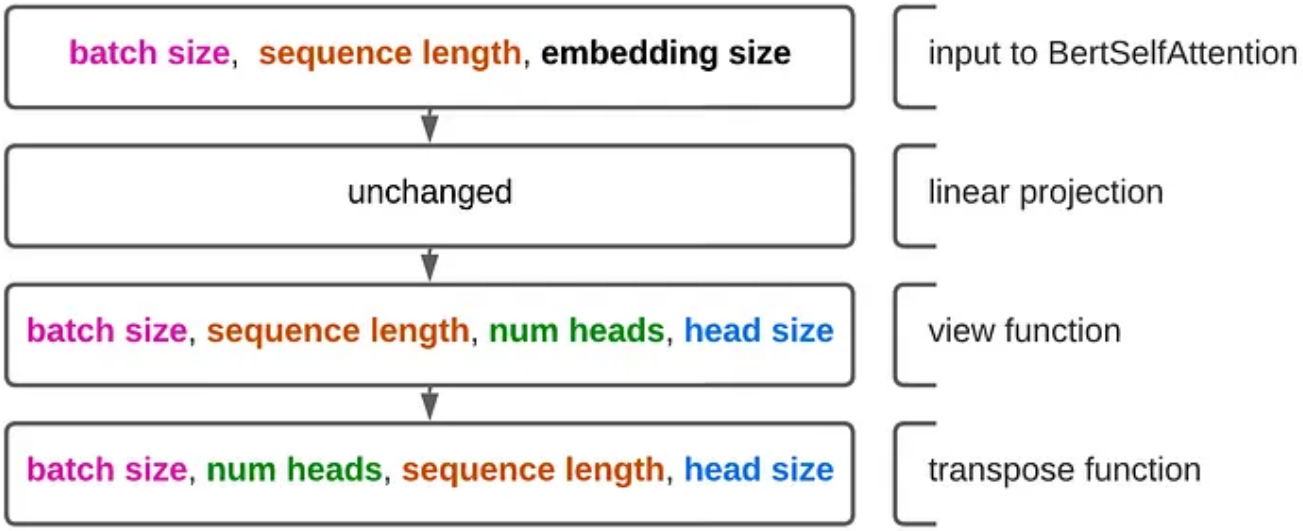
BERT Self Attention

The first thing of note is that there is a parameter for the number of heads and the head size [line 5 and 6]. This is because BERT employs something called multi-headed attention. These are multiple self-attention mechanisms that act independently of each other. There are 12 of them in each transformer layer.

Having multiple attention heads allows the model to focus on different types of relationships and dependencies in a sequence leading to richer token representations.

The size of each head (length of all the vectors within) is just the original embedding vector size divided equally among the number of heads. We see this in the generation of the query, key and value vectors that first come out of their respective linear layers at the length of the full embedding vector and then are separated through the view function [line 16, 17, 18].

The sequence length and the number of heads dimensions are further transposed so that when we apply matrix multiplication, we will be effectively performing independent matrix multiplies on the last two dimensions (sequence length and head size), keeping the batches and attention heads independent [line 16, 17, 18].



Progression of matrix dimensions which happens three times independently on line 16, 17, and 18.

The key matrix is then transposed yet again, this time on the sequence length and head size dimensions, to facilitate the matrix multiplication between the query and key matrices (QK.T) [line 20].

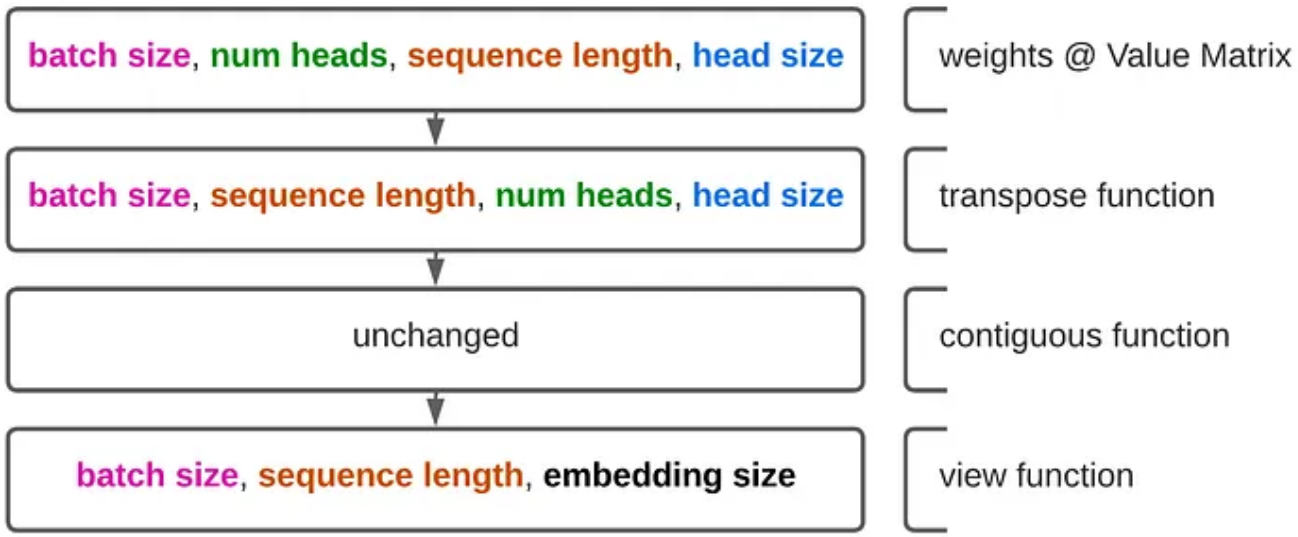
Here the weights are also scaled by 1 over the square root of the length of the vectors in the Key matrix, which is equivalent to the head size to the power of -0.5 [line 20].

There is a cumbersome looking line of code afterwards that is just substituting in negative infinity at the indices where the attention mask is equal to zero [line 23, 24, 25].

And after we multiply the weights (scaled and softmaxed QK.T) with the value vectors to get the enriched token representation vectors [line 30], the dimensions are reordered and then the individual heads are concatenated to form one long vector of length 768 [line 31].

Note the contiguous function does not influence the dimensions of the matrix. It just rewrites the tensor in a contiguous block of memory to

maintain the expected behavior of some PyTorch functions that alter tensor shapes.

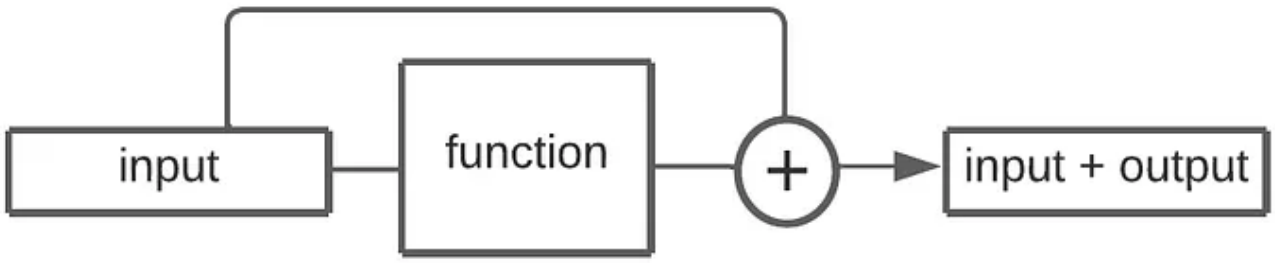


Progression of matrix dimensions through lines 30 and 31 of the code above.

The output from self-attention is further processed in the BertSelfOutput class [line 36]. A linear transformation takes place here, which for some configurations of BERT may be useful in projecting the concatenated output of the multiple attention heads back to the original embedding size.

This is also where we first see a **residual (skip) connection**. Here the input to SelfAttention (original embedding) gets added to the output of SelfAttention.

Basically, the inputs skip over the self-attention process and supplement the outputs.



Skip connection.

The main benefits of skip connections include:

- Help to mitigate the vanishing gradient problem.
- Make it easier for the model to learn identity functions, which are useful for disregarding modules that do not improve model performance.

Last but not least, **BertAttention** just encapsulates BertSelfAttention and BertSelfOutput, and processes them in order [line 52].

Position-Wise Feed-Forward Network

The position-wise feed-forward network is the second main component of a transformer layer and it serves to increase the capacity of the model with additional learnable parameters and facilitates the continued enrichment of the token representations from layer to layer in the encoder. Position-wise just refers to the fact that the transformation is separately and identically applied to each token, or in other words at each position of the sequence.

The series of operations include:

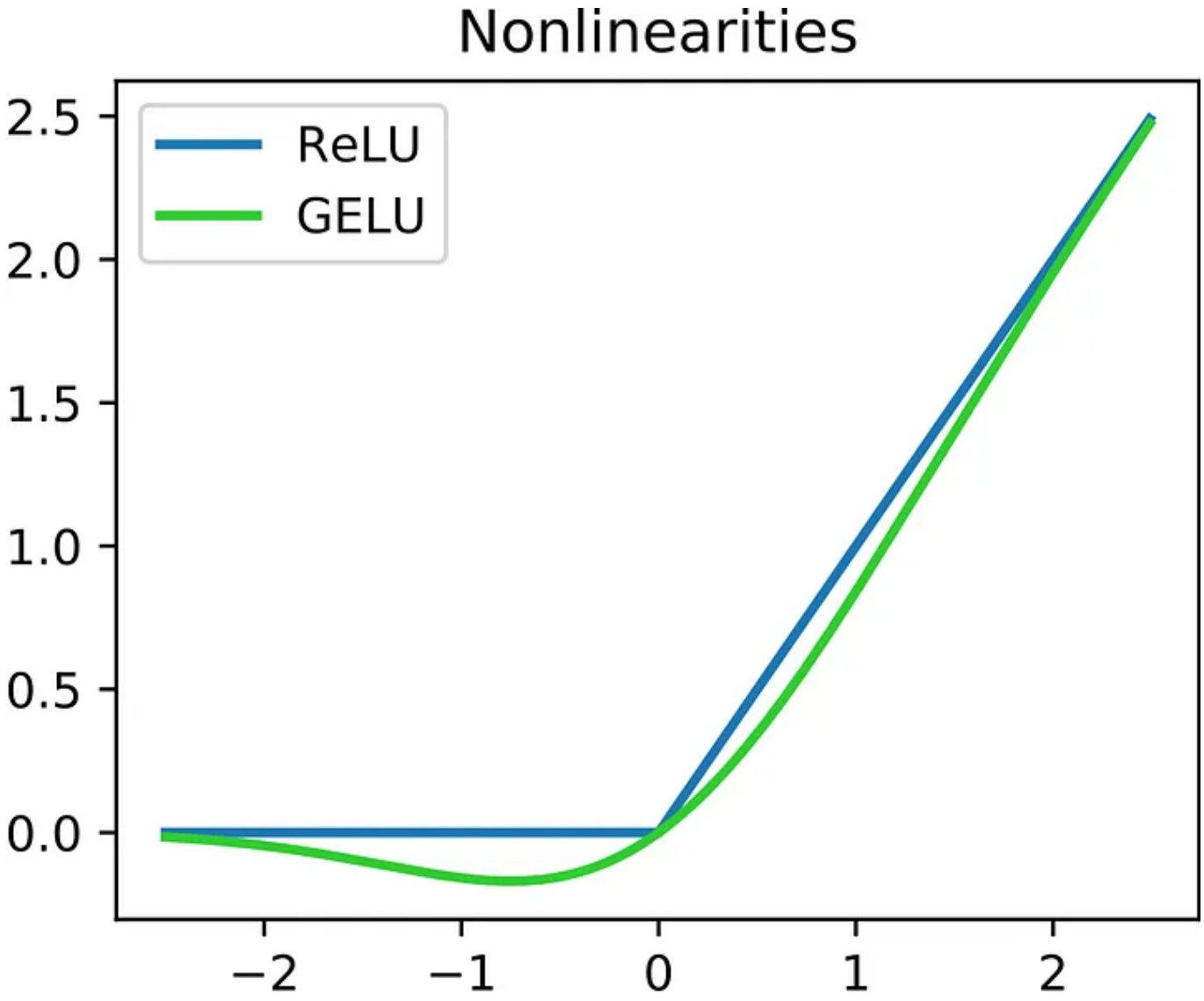
1. Projecting the input to a higher embedding size
2. Running it through a GeLU activation function

3. Projecting it back down to the original embedding size
4. Applying dropout
5. Applying a skip connection
6. Applying layer normalization

There is nothing too fancy in the feed-forward sub-layer, but it is worth pointing out the GeLU operation, which is a non-linear activation function.

Introducing a non-linearity here helps the model capture more complex, non-linear patterns in the data.

GeLU scales each input value by the cumulative probability of all values less than or equal to the input value in a normal distribution with a mean of 0 and variance of 1.



Comparison of GeLU and ReLU functions, highlighting the smoothness and non-zero negative values for GeLU.

Two advantages of GeLU over other activation functions, such as ReLU:

- GeLU is smooth (differentiable everywhere) helping with optimization and convergence.
- For negative inputs, it has non-zero negative gradients allowing for better information flow during backpropagation, which helps with training.

The series of six operations outlined above are broken into two modules, **BertIntermediate** and **BertOutput.**

Notice the **skip connection** [line 24].

BERT Feed Forward Module.

The entire transformer layer comes together in **BertLayer**, where BertAttention, BertIntermediate, and BertOutput get processed in order.

BERT Transformer Layer.

Pooler

The output from the encoder is a matrix where each row is an enriched representation of the corresponding token from the input sequence.

However, for the task of sequence classification, what is needed is a vector representation that captures the meaning of the entire input sequence (not

just one token).

Common methods for producing such a representation include taking either the element-wise mean or maximum of all the token representations.

The method chosen here is just to select the representation of the first token in the input sequence, which is the classification token [line 8].

The [CLS] token was developed for the pretraining phase with next-sentence prediction in mind and is intended to encode the relationship between two sentences.

It is also expected to contain a contextualized, high-level representation of the entire sequence and therefore is a good candidate to be pooled for sequence classification.

BERT Pooler.

The classification token is further processed through a linear projection (dense layer) and non-linear activation function (hyperbolic tangent — tanh) with the usual aim of increasing the capacity of BERT to model complex relationships [line 9 and 10].

As an interesting side note, the hyperbolic tangent activation function was originally chosen for interpretability, but other functions are likely to work just as well.

The **BertModel** class houses all the modules that are considered part of the base architecture of BERT — the Embedding, Encoder, and Pooler — and processes them in order.

BERT Model class.

Notice that both the output from the encoder and the pooler are returned [line 12].

The output from the pooler is all that is needed beyond this point for sequence classification. However, there are some tasks or possible customizations that may need to utilize the un-pooled output. Therefore, it is made accessible.

Classification Head

Finally, the output from the pooler is passed through the classification head, which simply involves projecting the pooled embedding into a space with dimensionality equal to the number of different classes [line 7].

It is called a head because this component of the model can be swapped out to suit a particular task.

This is in contrast to the backbone of BERT — responsible for creating the contextualized representations of the tokens in the sequence — that remains the same regardless of the task.

The **BertForSequenceClassification** class is the outermost class that we call to instantiate our BERT model. It houses both the base architecture (self.bert) and the classification head (self.classifier).

The outputs are the logits for which there is one value for each class. Taking the maximum value of these logits will give us the predicted class. However, if it is desired to interpret the logits as probabilities the softmax function will need to be applied.

Conclusion

That brings us to the conclusion of this post. It is a lot to take in but hopefully you were able to learn a few things that may not have been laid out in as much detail elsewhere.

In the next part of this series, we will dive into the code and theory behind Low-Rank Adaptation (LoRA), a method which reduces the memory demand required for fine-tuning.

NLP | Large Language Models | Machine Learning | Python Programming | Bert

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list

Written by AR

3 Followers

Following

More from AR

AR

LoRA: Low-Rank Adaptation from Scratch —Code and Theory

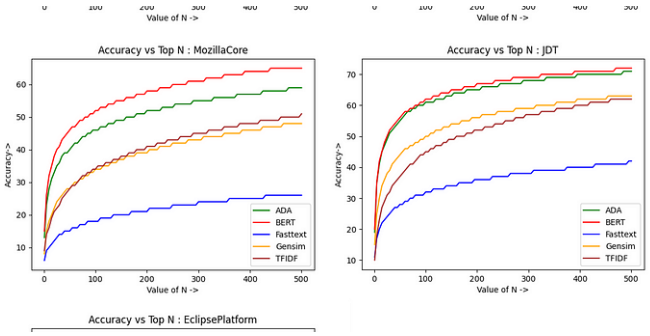
Transformer models can have a lot of parameters which can make fine-tuning them an expensive and time-consuming endeavor...

5 min read · Aug 3

3

See all from AR

Recommended from Medium



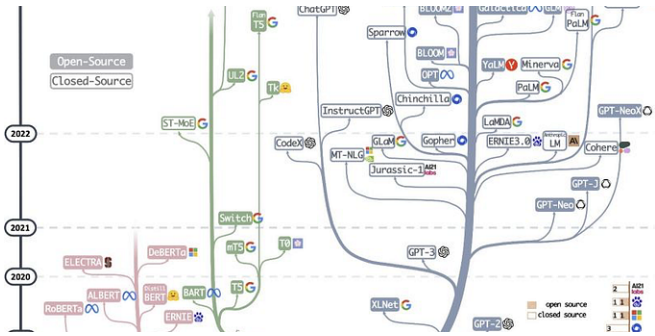
Avinash Patil

Embeddings: BERT better than ChatGPT4?

In this study, we compared the effectiveness of semantic textual similarity methods for...

4 min read · Sep 19

3 1



Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14

372

Lists

Natural Language Processing

669 stories · 283 saves

Coding & Development

11 stories · 200 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves

Stories to Help You Grow as a Software Developer

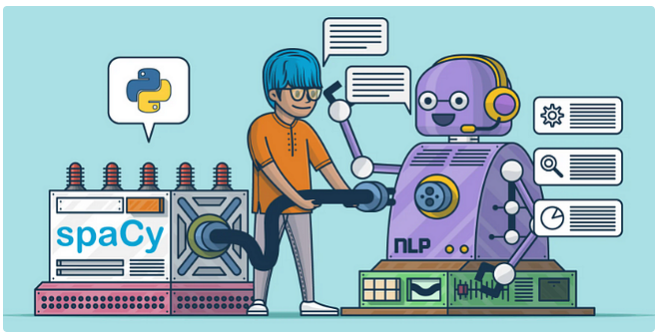
19 stories · 423 saves

Ahmet Taşdemir

Fine-Tuning DistilBERT for Emotion Classification

In this post, we will walk through the process of fine-tuning the DistilBERT model for...

8 min read · Jun 14



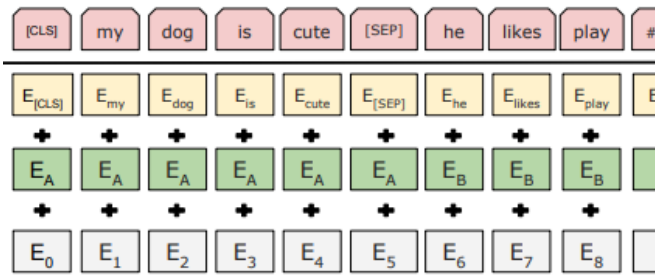
HasancanÇakıcıoğlu

Comprehensive Text Preprocessing NLP (Natural Language...

Text preprocessing plays a crucial role in Natural Language Processing (NLP) by...

12 min read · Jul 9

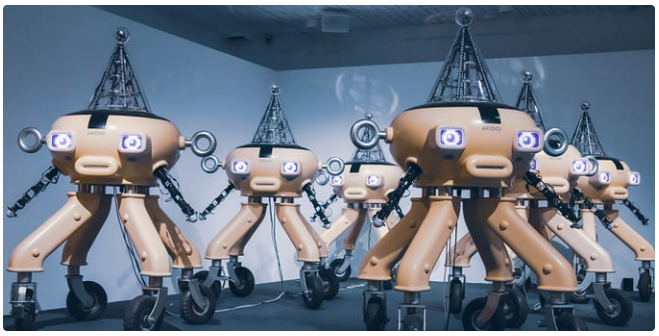
12



Zain ul Abideen

A Comparative Analysis of LLMs like BERT, BART, and T5

Exploring Language Models



AR

LoRA: Low-Rank Adaptation from Scratch—Code and Theory

Transformer models can have a lot of parameters which can make fine-tuning the...

6 min read · Jun 26

20 1

5 min read · Aug 3

3

See more recommendations