Talk 15: Transformer Applications in NLP

Rainer Gogel
gogel@stud.fra-uas.de Matr.No.1272442
Winter semester 2022/23: Learning from Data
Frankfurt University of Applied Sciences
Prof. Dr. Joerg Schaefer

***Abstract*—This document represents the essence of a talk given by the author of this paper as part of the requirements for the Master Program in Computer Science at the University of Applied Sciences in Frankfurt. The module "Learning from Data" was taught by Prof. Dr. Joerg Schaefer in the winter semester 2022/23 and the assigned talk subject was "Transformer Applications in NLP". The talk was divided into three parts and covered the following topics: An introduction to "Word Embeddings", the description of the Transformer [1] architecture on the example of BERT [2] and the application of a pretrained and finetuned BERT model to predict the sentiment of Twitter text messages.**

***Index Terms*—transformers, BERT, word embedding, pretrained language model, NLP**

I. Word Embeddings

Written human language consists of words which first must be encoded into numbers before computers can understand them. One way to encode words is to use so called one-hot encodings where every unique word gets its own binary number vector with one value therein equal to 1, and all others equal to 0. (Fig.1).

	horse	bear	king	queen	banana	apple
horse	1	0	0	0	0	0
bear	0	1	0	0	0	0
king	0	0	1	0	0	0
queen	0	0	0	1	0	0
banana	0	0	0	0	1	0
apple	0	0	0	0	0	1

Fig. 1. Each unique word gets its own column in a tabular data structure. In case the word is present, the number in that column is one, else zero. Each table row then is converted into a vector representing the respective word.

The first disadvantage of this approach is the high consumption of compute memory as each word in our simple example in Fig.1 would require a sparse six-column vector with five 0's and only one 1. The second disadvantage is that this approach does not encode syntactical or semantic word similarities as similar words would have different and unrelated vector representations.

A. Manually crafted features

To encode syntactical or semantic word commonalities, one could think about word characteristics or attributes and manually encode the magnitude of those attributes for each word as shown in Fig.2. The word "horse" for instance, would then be represented as a vector of values for each of these handcrafted features (Fig.3).

	FEATURES						
	animal	can ride it	rich	two legs	four legs	peaceful	food
horse	1	1	0	0	1	1	0.2
bear	1	1	0	0	1	0	0
king	0	0	1	1	0	0.2	0
queen	0	0	1	1	0	0.8	0
banana	0	0	0	0	0	1	1
apple	0	0	0	0	0	1	1

Fig. 2. Each word is represented by manually crafted feature values. A horse, for instance, is an animal, one can ride on it, it has four legs, is usually peaceful and is sometimes processed to food (meat or sausage).

```
vec_horse = [1, 1, 0, 0, 1, 1, 0.2]
```

Fig. 3. The vector for the word "horse" contains the values for each manually crafted feature. See also Fig.2.

This approach ensures that related words are represented similarly as their values in the respective vector position are close to each other. Such similarities can also be calculated numerically by applying the cosine similarity method which is the standardized inner or dot product of the two word vectors to be compared (Fig.4). The standardization ensures that the calculated similarity value lies between zero and one and thus makes comparisons between different word pairs possible.

$$\text{Cos}\,(A, B) = \frac{A \,.\, B}{|A|\,|B|}$$

Fig. 4. Cosine similarity formula: standardized dot or inner product of two vectors or matrices.

The cosine similarities between each pair of those words previously shown is depicted in a matrix in Fig.5. The cosine similarity for the word pair "horse" and "apple", for instance, yields a value of 0.42. For the word pair "apple" and "banana" however, this value is 1.0, owing to the fact that both are fruits

and in our simple example have the same feature values at each index position of their word vector.

cos sim	horse	bear	king	queen	banana	apple
horse	1.00	0.86	0.07	0.24	0.42	0.42
bear	0.86	1.00	0.00	0.00	0.00	0.00
king	0.07	0.00	1.00	0.93	0.10	0.10
queen	0.24	0.00	0.93	1.00	0.35	0.35
banana	0.42	0.00	0.10	0.35	1.00	1.00
apple	0.42	0.00	0.10	0.35	1.00	1.00

Fig. 5. Exemplary cosine similarity matrix of some word pairs calculated on basis of the vectors provided in Fig.2.

B. Learned features

Manually crafting feature values for every word in a vocabulary is cumbersome at best. It is also subject to ambiguity and human misjudgement. Are all "queens" *rich* as the feature value in Fig.2 suggests? And have you ever successfully *ridden* a grizzly "bear"? Is there no *food* in (burger) "king"? And isn't an "apple" *rich* in vitamin C?
Better than crafting features by hand is to have a machine learning model to learn these features and feature values. This is the "word2vec" approach Mikolov et al proposed in their seminal papers in 2013 [3][4]. A shallow, two-layer neural network (Fig.6) is trained with sentences that contain a masked word (CBOW-approach. Alternative: Skip-Gram-approach. See: [4]) to be predicted (target or dependent variable) based on its non-masked adjacent words (input or independent variables) in that sentence.

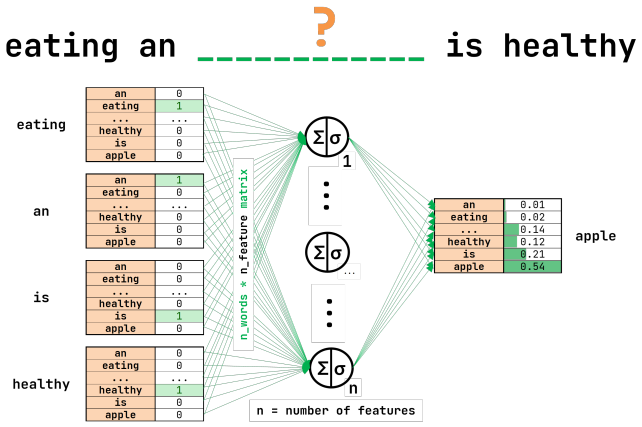

Fig. 6. word2vec: Masked words are learned by a shallow, two-layer neural network.

The number of learned features depends on the number of neurons in the middle layer and the learned weights represent the respective feature values. The learned features do not have names as before (like "animal", "rich", "food", etc.) and thus cannot be interpreted semantically by humans (easily) (Fig.7).

	FEATURES						
	weights 1	weights .	weights .	weights .	weights .	weights .	weights n
horse	0.57	0.99	0.23	0.73	0.73	0.66	0.33
bear	0.55	0.30	0.11	0.92	0.85	0.93	0.79
king	0.19	0.17	0.37	0.99	0.14	0.96	0.64
queen	0.91	0.12	0.05	0.67	0.30	0.36	0.05
banana	0.70	0.42	0.55	0.12	0.57	0.97	0.86
apple	0.60	0.80	0.80	0.19	0.52	0.41	0.48

```
vec_horse =  [0.57, 0.99, 0.23, 0.73,  ... , 0.73, 0.66, 0.33]
 vec_bear =  [0.55, 0.3, 0.11, 0.92,  ... , 0.85, 0.93, 0.79]
 vec_king =  [0.19, 0.17, 0.37, 0.99,  ... , 0.14, 0.96, 0.64]
vec_queen =  [0.91, 0.12, 0.05, 0.67,  ... , 0.3, 0.36, 0.05]
vec_banana = [0.7, 0.42, 0.55, 0.12,  ... , 0.57, 0.97, 0.86]
vec_apple =  [0.6, 0.8, 0.8, 0.19,  ... , 0.52, 0.41, 0.48]
```

Fig. 7. The learned parameters/weights represent the feature values.

The learned word vectors can also be used to perform mathematical operations. An element-wise addition of the vectors representing the word "king" and "man" and the subsequent subtraction of "woman" yields a vector that is very similar to the vector of the word "queen" (Fig.8).

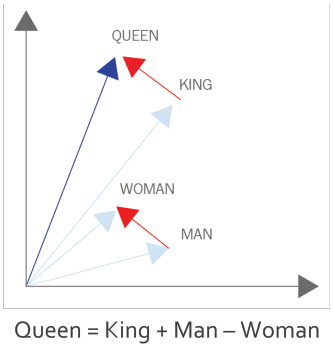

Fig. 8. Add and subtract word vectors: the infamous "king + man - woman = queen" example. Image taken from: [5]

C. The problem with static word embeddings

The learned word vectors are also known as word embeddings and work well for tasks such as measuring similarities between individual words. But they often fail when the scope goes beyond just words towards the semantic meaning of whole sentences. A sentence is simply not just a chain of individual and independent words but a construct containing interdependencies.

One of these word-wise inter-dependencies are homophones, i.e. words that are spelt the same but have different meanings. The word *"flies"*, for instance, in the sentence *"time flies like an arrow"* refers to a verb, i.e. an activity, whereas the same word in the sentence *"fruit flies like a banana"* refers to a noun and an insect species. The meaning of the word *"bank"* in the sentence *"he withdraws money from his bank"* is strongly

affected by the surrounding words *"withdraws"* and *"money"* as they clearly suggest that *"bank"* in that sentence refers to a financial institution. In contrast, the meaning of the same word *"bank"* in the sentence *"he was fishing in the river from a sand bank"* is strongly affected by the words *"river"* and *"sand"* (Fig.9).

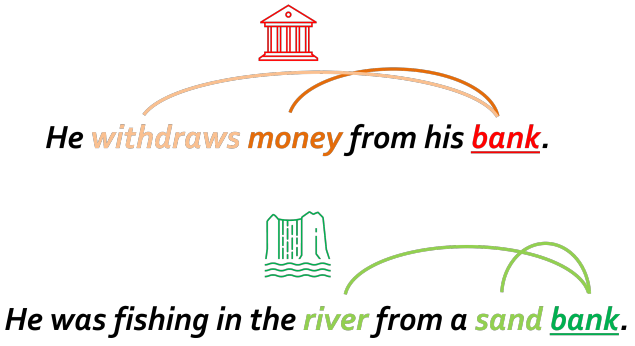

Fig. 9. Homophone "bank" is dependent on its context

When humans read these sentences, they derive the meaning of the word "bank" by paying "attention" to these adjacent words [6].
There are more inter-dependencies in sentences than homophones, but in general it can be said that the meaning of a word depends on its respective context.

English linguist John Rupert Firth [7] in that respect famously stated that:

"You shall know a word by the company it keeps!"

Word embeddings such as word2vec [3][4] are static in the sense that they treat equal words alike without taking their context into account. For the example above, static approaches encode the word *"flies"* with the same vector independent of whether the word appears in the sentence *"time flies like an arrow"* or *"fruit flies like a banana"*. This is one of the reasons why static word embeddings such as word2vec often fail to understand the meaning of sentences. What is needed are dynamic word embeddings that are able to take this context into account. Such a dynamic or contextualized word embedding would represent the word *"flies"* in our example with different vectors dependent on the sentence and context the word appears in.

II. BERT

Here, transformers [1] came to the rescue. Viswani et al in 2017 in their seminal paper "Attention is all you need" ("AIAYN") proposed a new neural network architecture for the domain of Natural Language Processing ("NLP") and dubbed it *"transformers"*. Transformers embrace the human "attention" principle discussed above, i.e. the ability to infer the meaning of a word by paying "attention" to its adjacent words in that sentence.
The importance of the "attention" principle had already been discovered earlier [8]. In 2018, ULMFiT [9] and ELMO [10] implemented it in a bi-directional LSTM [11] recurrent neural network ("RNN"). These LSTM-based models performed well on specific tasks but suffered from long training times and fading context memory if sentences were longer than just a few words. BERT ("Bidirectional Encoder Architecture from Transformers") was released in 2018 by Google researchers [2] and is the first neural network model (partly) based on the architecture proposed in the AIAYN paper. Unlike ELMO and ULMFiT, transformer-based architectures such as BERT, do not rely on recurrence, but on "self-attention", a concept explained in later chapters. This results in much shorter training times and, more importantly, in the preservation of context memory even if sentences are very long.

A. The general architecture of BERT

The transformer architecture consists of *Encoder* and *Decoder* stacks. As its name implies, BERT ("Bidirectional *Encoder* ...") only uses the Encoder part of the transformer (Fig.10).

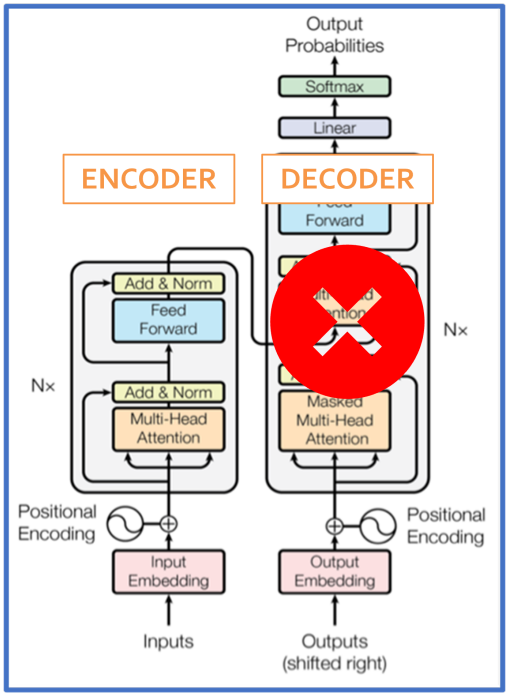

Fig. 10. Transformer architecture: Encoder and Decoder stacks with N layers. BERT only uses the Encoder stack. The "bert-base-uncased" model has 12 layers (N=12).

There are multiple versions of BERT, among them "bert-base-uncased". "Uncased" in this context means that upper case letters in the text are ignored and converted to lower case, loosing some information in the process. "Base" refers to the model size in terms of vector dimension and the number of layers and heads. The "base" model has 12 layers with 12 heads each.
A *layer* refers to an Encoder or Decoder block as shown in Fig.10 to be stacked vertically. A *head* is also known as a

Hidden State and refers to the dynamic word embeddings in each layer.
BERT uses a sub-word tokenization method called *"Word-Piece"* [12] that splits some but not all words into multiple tokens. Among the three most common tokenization methods (letter tokenization, sub-word tokenization, word tokenization), the sub-word tokenization method has been found superior [12][13]. Each token vector in the "bert-base-uncased" model has a length of 768 and each sentence can contain up to 512 tokens. The "base" version contains 110 Mio. learnable parameters/weights.
If BERT is mentioned within the next few chapters, it refers to the pretrained PyTorch [14] version of the "bert-base-uncased" model downloaded from the Hugging Face website [15]. All learnable parameters were already trained and thus frozen for the purpose of the subsequent analysis. The pretrained model contains 30,522 word tokens.
Raw text for both, upstream (training) and downstream (prediction) tasks, must first be preprocessed and tokenized. As this was not subject of the talk, it is assumed that this transformation has already been done.

B. The BERT architecture in greater detail

In the next sections, the following sentences (*"sample sentences"*) are used to go through each step of BERT's Encoder block (Idea from: [13]):

"time flies like an arrow"
"fruit flies like a banana"

Each word in these two sentences only consists of one token. This makes comparisons in the next sections easier than if words were split into multiple tokens. The expressions "word" and "token" in the next chapters are used interchangeably as they mean the same in this example. Both sentences also have the same length (n=5) which avoids padding issues, i.e. the need to add "empty" tokens in the shorter sentence. If only the first sentence is shown, it will subsequently just be referred to as the *Sample Sentence*.

1) Embedding layers: BERT has three different kinds of initial embeddings, i.e. embeddings in the first layer:

- *Token Embeddings*
- *Position Embeddings*
- *Segment Embeddings*

All three embedding layers in BERT are matrices (PyTorch tensors) of learnable parameters.

a) Token Embeddings: The *Token Embedding* matrix in BERT has 30,522 rows and 768 columns representing 30,522 tokens, each with a feature vector of length 768. Each feature vector is randomly initialized before training and thereafter contains learned feature values for each of the 30,522 word tokens. Although the *Token Embedding* layer is only the first part of BERT's architecture, it per se already can be interpreted as a 2-layer shallow neural network comparable to the architecture of word2vec [4]. Like the values in the word vectors of word2vec, the learned values of a given token vector already represent feature values of a token but these are still static at this stage, i.e. same words are represented by the same token vectors in all sentences, independent of their context.
Once learned, the *Token Embedding* matrix serves as a lookup table for each token. The *Token Embeddings* for the tokens in our *sample sentences* can be retrieved by their token identification numbers ("token ids").
PyTorch tensor objects in BERT in general have a dimension of: *[number of batches, number of tokens per batch, number of feature values per token]*. As we pack our two *sample sentences* into two different batches, the two sentences are concretely represented as a PyTorch tensor with a dimension of: *[2, 5, 768]*. For simplification and demonstration purposes, special tokens such as [CLS] or [SEP] are left out.

b) Position Embeddings: In LSTM-based models such as ULMFiT [9] and ELMO [10], there is no need to encode token positions as these kinds of RNNs by design process tokens sequentially. In contrast and as shown in later chapters, BERT does not make use of recurrence but relies on the "self-attention" mechanism which processes information parallelly. As the order of words in sentences syntactically and semantically play an important role, the positions of these words in each sentence must be encoded. Each batch (or sentence in the given example) in BERT can have up to 512 tokens. To be able to later combine *Token* and *Position Embeddings*, each token position is also encoded with a vector of length 768. Once learned, the *Position Embedding* matrix also serves as a lookup table for each position, but there is no need to enter token position numbers or ids because the position of a word within a sentence is known from their tensor position and handled internally.

c) Segment Embeddings: As stated previously, the two *sample sentences* are packed into two different batches. It would also be possible to pack them into just one batch so that our PyTorch tensor would have a dimension of *[1, 10, 768]* instead of *[2, 5, 768]*. But then an encoding of which token belongs to which sentence would be required[1]. This is the purpose of *Segment Embeddings*. The *Segement Embeddings* matrix has a dimension of *[2, 768]*. The first of the two row vectors in that matrix is applied for tokens that are in the first sentence, the second vector for tokens that are in the second sentence. *Segment Embeddings* are required in the training phase for the task of "next sentence prediction".

[1]*Segment Embeddings* are already dealt with in the tokenization phase. The Hugging Face AutoTokenizer object, for instance, receives as input either a list of sentence strings or a list of tuples of sentence strings. In the first case, sentences are packed into different batches whereas in the second case, two sentences in a tuple are packed into only one batch. BERT only allows for up to two sentences per batch.

d) Input Embeddings: Sum of all Embeddings: The final *Input Embeddings* tensor is an element-wise sum of the values in the *Token Embeddings*, *Position Embeddings* and *Segment Embeddings* (Fig.11). In addition, the tensor values get normalized and "dropout"-regularized.

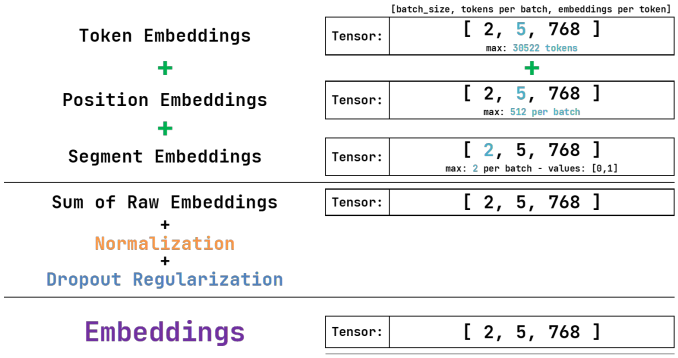

Fig. 11. *Input Embeddings*: Element-wise sum of *Token Embeddings*, *Position Embeddings* and *Segment Embeddings*. In addition, "Normalization" and "Droput Regularization" is applied.

This *Input Embeddings* tensor is then passed to the next layer of the BERT architecture (Fig.12).

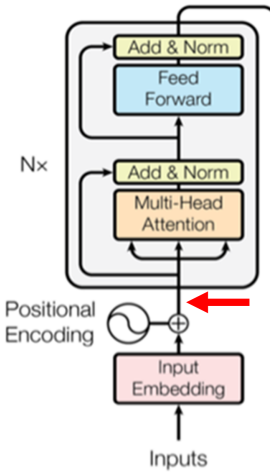

Fig. 12. Red Arrow: *Input Embeddings* get passed to the first Encoder layer ("grey box") of the BERT Encoder stack.

2) Multi-Head Attention: The first element of BERT's first Encoder layer ("grey box" in Fig.13) is the *"Multi-Head Attention"* module. It consists of several sub-layers depicted in the orange box on the right of Fig.13.

For demonstration purposes, the focus in the following analysis will only be laid on the *Sample Sentence* "time flies like an arrow". The batch dimension thus will be ignored and the dimension of the *Sample Sentence* will be assumed to be *[number of tokens per sentence, number of feature values per token]* or *[5, 768]*.

a) Three Linear Layers: The *Input Embeddings* tensor gets passed to three different, learnable *Linear Layers*, i.e. ANN

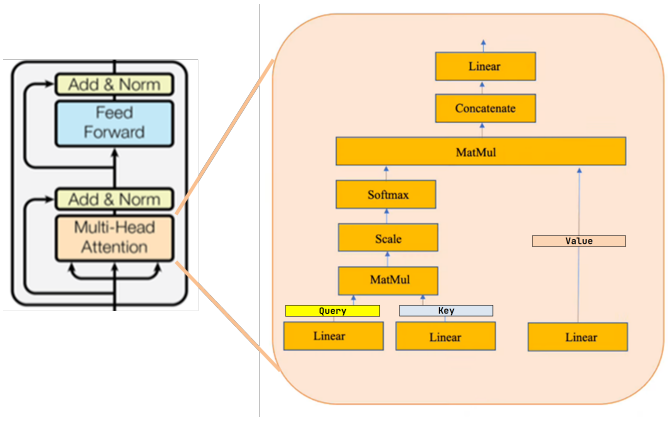

Fig. 13. Multi-Head Attention. Image from: [2] and [16]

layers whose parameters/weights have a dimension of *[768, 64]* each. The *Input Embeddings* matrix gets multiplied by these learned parameters to produce three more matrices that are given very peculiar names [2]: *Query*, *Key* and *Value* (Fig.14).

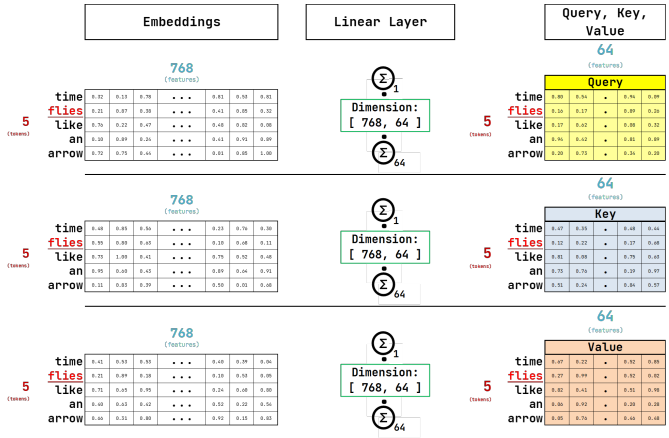

Fig. 14. Each weight/parameter matrix of the *Three Linear Layers* has a dimension of *[768, 64]*. The matrix multiplication of the *Input Embeddings* matrix with these three weight/parameter matrices "produce" the *Query*, *Key* and *Value* tensors. Each of them has a dimension of *[5, 64]*.

The three *Linear Layers* convert the *Input Embeddings* dimension of *[5, 768]* to the *Query*-, *Key*- and *Value*-dimension of *[5, 64]*. It can be assumed that the *Query*, *Key* and *Value* matrices still contain the encoded feature, position and segment information for the tokens in the *Sample Sentence*, but in a condensed form. One task of the three *Linear Layers* thus is the dimension reduction of the *Input Embeddings*.

An important aspect of this transformation is that the three *Query*, *Key* and *Value* matrices are different from each other as each *Linear Layer* (hopefully) has learned different parameters. This will be further elaborated on in a later chapter.

b) Attention Filter: The next step in the *Multi-Head Attention* module is the matrix multiplication depicted in Fig.15. The *[5, 64]*-dimensional *Query* matrix is multiplied by the transposed *[5, 64]*-dimensional *Key* matrix so that the resulting matrix is of dimension *[5, 5]*, equivalent to the number of words in the *Sample Sentence*. Thereafter, all values get divided by the square root of the column dimension of the *Query* and *Key* matrices, which is sqrt(64) or 8. Lastly, the values row-wise undergo a softmax operation, i.e. get scaled to values in between zero and one that row-wise all add up to one. The resulting matrix here (Fig.15) named *"Attention Filter"* became also known as "Scaled Dot-Product Attention".

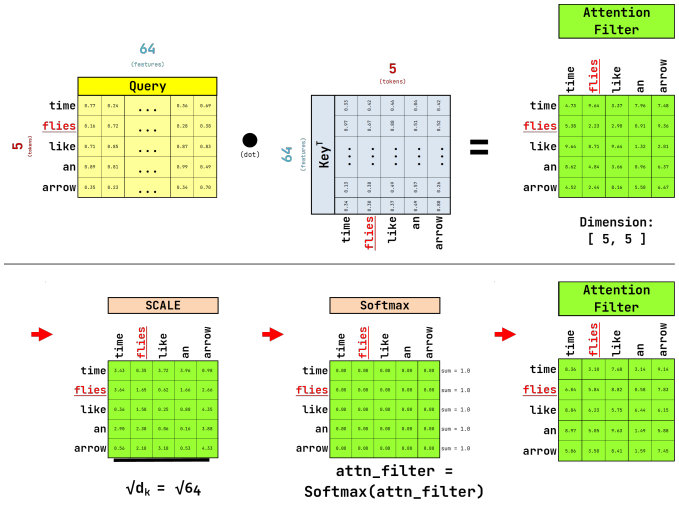

Fig. 15. *Attention Filter*: Matrix multiplication of *Query* and *Key* matrices plus applying Scaling and Softmax operations.

c) Analysis of the Attention Filter: A matrix multiplication of two matrices in general is the inner or dot product of every row vector of the first matrix with every column vector of the second matrix. The scalar value in the *"Attention Filter"* matrix at row index i and column index j is the result of the dot product of the i-th row vector in the *Query* matrix with the j-th column vector of the *Key* matrix. For instance, in Fig.16, the 2nd row vector of the *Query matrix* (representing the word "flies") and the 5th column vector of the *Key matrix* (representing the word "arrow") after a dot product operation yield a scalar value in the 2nd row and 5th column of the *Attention Filter*.

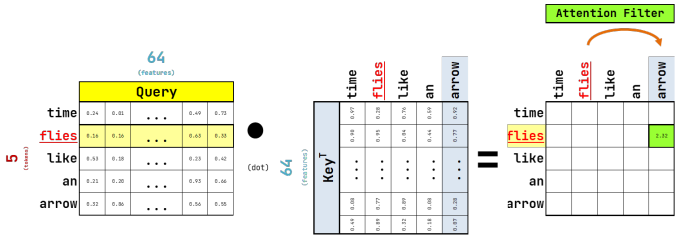

Fig. 16. Matrix multiplication: Calculation of values in the *"flies"* row of the *"Attention Filter"*.

In the section *Word Embeddings*, it was shown, that the normalized dot product of two vectors express the cosine similarity between the words representing these vectors. The dot product of the vectors in Fig.16 representing the "flies" and "arrow" tokens thus can be interpreted as the not-yet-normalized cosine similarity between the "flies" and "arrow" tokens in the *Sample Sentence*. After the matrix multiplication, the values are scaled and row-wise undergo a softmax operation. This can be interpreted as a form of normalization. The *Attention Filter* matrix is asymmetric as the softmax function is only applied row-wise but not column-wise. The values in the *Attention Filter* matrix thus represent some form of similarity that should only be read row-wise or horizontally. The similarity value between the word "flies" and the word "arrow" in the *Attention Filter* matrix thus can only be found at the coordinates of the 2nd row and the 5th column but not at the coordinates of the 5th row and the 2nd column (Fig.16).

d) Analysis of the Matrix Multiplication: If the three *Linear Layers* in the *Multi-Head Attention* module were to produce equal *Query*, *Key* and *Value* matrices, the (upper left to lower right) diagonal values in the *Attention Filter* matrix would be one or close to one (Fig.17).

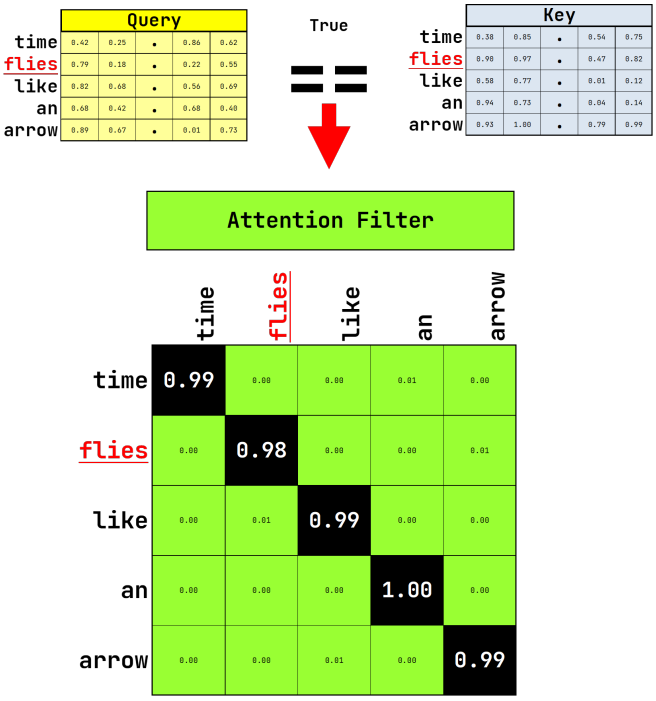

Fig. 17. Matrix multiplication: The diagonal values in the *Attention Filter* matrix would be close to one if the *Query* and *Key* matrices were alike.

This is because the vector representation of any token in both, the *Key* and *Value* matrices, then would be equal and the (somehow) normalized dot product of the two equal vectors would yield values close to one. The similarity of a word in a sentence would be highest with itself and the similarity to other words would be negligibly small.
But this is not what the *Attention Filter* is supposed to do.

The task of the *Attention Filter* is not to identify semantically or syntactically similar words, but to identify those adjacent words in a sentence that help determine the context for and the meaning of a given word. In the *Sample Sentence*, a human reader would probably pay attention to the words ”arrow” and ”time” to determine the context and the meaning of the word ”flies”. Given the *Sample Sentence*, an ideal *Attention Filter* would probably have higher values at the coordinates of the 2nd row and both, the 1st and 5th column. The higher values at these coordinates would reflect the ”attention” paid to the words ”time” and ”arrow” that is needed to determine the semantic or syntactic meaning of the word ”flies”. And an ideal *Attention Filter* would probably also have lower values in the diagonal (i-index=j-index) to lower the attention that a word pays to itself.

So what is needed to have the algorithm produce these identified higher and lower values in the *Attention Filter* at these coordinates respectively? The 1st and 5th row vector (representing the words ”time” and ”arrow”) of the *Key* matrix would need to become more similar to the 2nd row vector (representing the word ”flies”) of the *Query* matrix. Because only then the relevant similarity values in the *Attention Filter* would increase. The values in the 2nd row vector of the *Key* and *Query* matrix would need to diverge from each other as both represent the word ”flies”. Because only then the attention values in the *Attention Filter* diagonal, which represents the attention magnitude that words pay to themselves, would decrease.

e) The importance of the Three Linear Layers: The task of the three *Linear Layers* thus is not just the dimension reduction of the *Input Embeddings* as stated above, but more importantly, to ”construct” the *Query*, *Key* and *Value* matrices so that they incorporate the above outlined properties. This construction process hinges on the *[768, 64]*-dimensional weight/parameter matrices of the *Three Linear Layers*.

In the *Sample Sentence*, these weight/parameter matrices must shape the 1st and 5th row vector (representing the words ”time” and ”arrow”) of the *Key* matrix in a way so that they are more similar to the 2nd row vector (representing the word ”flies”) of the *Query* matrix. And they must also shape the values in the 2nd row vector of the *Key* and *Query* matrix so that they diverge from each other in order to decrease the attention (magnitude) that words pay to themselves.

There is probably a third requirement that becomes obvious only in the next chapter: the *Value* matrix, that is ”produced” by the weight/parameter matrix of the *Linear Layer*, must somehow conserve the encoded feature, position and segment information from the *Input Embeddings*. This is because the *Attention Filter* matrix later gets multiplied by the *Value* matrix to theoretically highlight and emphasize the most important features, positions and segments there. If the *Value* matrix would not represent the (condensed) feature, position and segment information, then the simple narrative of human-like ”attention” fell apart. Because if the *Attention Filter* in the later multiplication process does not highlight the most important features, positions and segments of tokens, what does it highlight then?

So the main task of the weight/parameter matrix of the *Linear Layer* that ”produces” the *Value* matrix is probably to ”only” reduce the dimension of the *Input Embeddings* from *[5, 768]* to *[5, 64]*.

The conclusion here is that there must be some important pattern and logic in the transformation from the *Input Embeddings* to the *Query*, *Key* and *Value* matrices. But such an analysis goes beyond the scope of the talk and this paper and will thus not be discussed further. It can be stated however, that the general requirement for the attention mechanism to work is that the *Query* and *Key* matrices at least must be different from each other (Fig.18) and that the weight/parameter matrices of the *Three Linear Layers* produce the *Query*, *Key* and *Value* matrices with the properties outlined above.

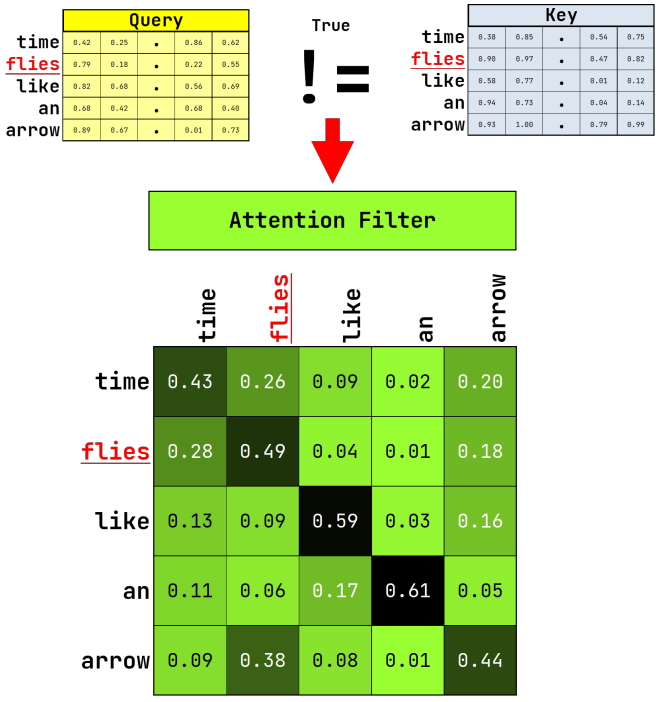

Fig. 18. Matrix multiplication: The diagonal values in the *Attention Filter* matrix would be very different from one if the *Query* and *Key* matrices are different.

f) Bidirectional Self-Attention: The multiplication of each row vector of the *Query* matrix with each column vector of the *Key* matrix is not done sequentially but in parallel as a matrix operation. This parallel processing is much faster than the usual sequential processing in LSTMs. In addition, it is also *bi-directional* as each *Query* matrix row vector is multiplied by each *Key* matrix column vector independent of the position of the word within a sentence that this column vector represents. The value in the 2nd row and 1st column of the *Attention Filter* matrix in Fig.19, for instance, is the ”attention value” paid to the word ”time”, a word that comes **before** the word ”flies” in the sentence.

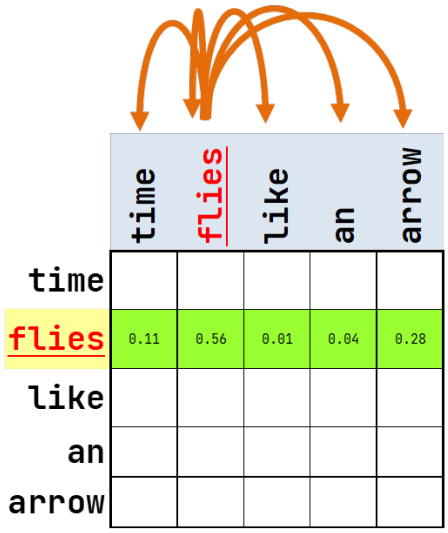

Fig. 19. Bidirectional Self-Attention: The dot products of the row and column vectors are *bi-directional*, i.e. towards previous as well as subsequent words in a sentence. To understand the context of a given word in a sentence, *attention* is paid to adjacent and relevant words within that sentence it*self*.

The value in the 2nd row and 5th column is the ”attention value” paid to the word ”arrow”, a word that comes **after** the word ”flies” in that sentence.
The attribute ”self” in *Self-Attention* is owed to the fact that the attention is paid within the sentence itself.
For longer sentences with many words in it, the context information is also not fading like in LSTMs. Even if context words are many words away from the word they are supposed to pay the most attention to, these context words nevertheless can have high values in the *Attention Filter* matrix. The mechanism and the result of a vector multiplication in general is unaffected by how far these vectors are away in a sentence.

g) Filtered Values: As indicated above, the *Attention Filter* matrix in the next step gets multiplied by the *Value* matrix (Fig.20), which was ”produced” earlier together with the *Query* and *Key* matrices.

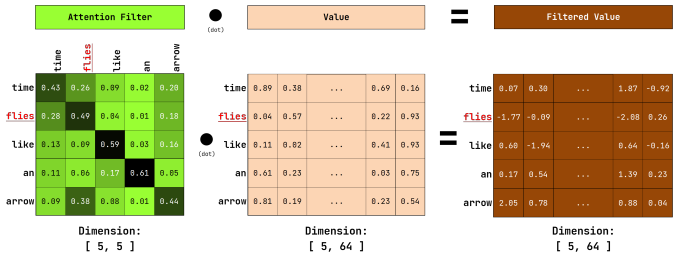

Fig. 20. *Filtered Values*: Matrix-Multiplication of the *Attention Filter* with the *Value* matrix supposedly highlights important features, positions and segments of the *Value* matrix.

Assuming that the *Value* matrix somehow conserves the feature, position and segment information from the *Input Embeddings*, the resulting *Filtered Value* matrix (dark brown matrix on the right of Fig.20) accentuates the relevant feature values in the *Sample Sentence*.
After the *Attention Filter* has identified the most important context words in a sentence, the multiplication of the *Attention Filter* matrix with the *Value* matrix carries over the most important feature values of the latter to the *Filtered Value* matrix. For a particular word in a sentence, important context words carry over higher values to the *Filtered Value* matrix, less important context words carry over lower or no values to it. Most importantly: for a particular word, this contribution from the *Value* matrix not only comes from the vector of the word itself, but also from vectors of other words within that sentence.
To stress this point, an example will be given that focuses on just one coordinate of the *Filtered Value* matrix: the 2nd row and the 1st column as shown in Fig.21.

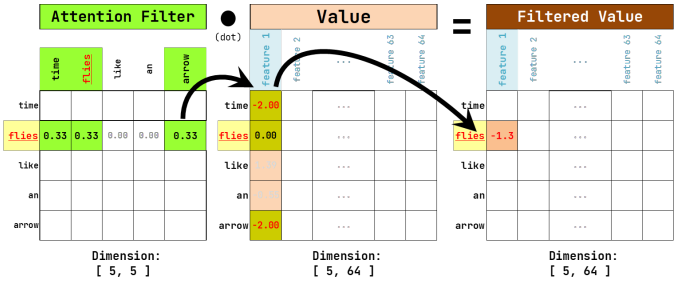

Fig. 21. *Filtered Values: ”time flies like an arrow”.*

It is assumed that the *Attention Filter* has identified the words ”time” and ”arrow” to be important context words for the word ”flies” in the *Sample Sentence*. The *Attention Filter* thus shows exemplary values of 1/3 for each of these context words and the word ”flies” itself, and a value of 0 for the remaining words ”like” and ”an”. It is further assumed that the first column of the *Value* matrix represents a semantically interpretable feature that humans would describe as ”food-like” (like the column ”food” in the handcrafted feature table in the first chapter). If a given word in a wider sense has something to do with ”food”, it is assumed that the values in the first column of the *Value* matrix are positive, else zero or negative. In Fig.21, the exemplary ”food” feature values for the word ”time” and ”arrow” are -2.0 as both words are assumed to have nothing to do with ”food”. The exemplary ”food” feature value for the word ”flies” itself is assumed to be 0 (as some ”flies” might be edible insects [17] and to make the point clearer). The dot product of the 2nd row of the *Attention Filter* with the 1st column of the *Value* matrix so yields a value of -1.3 at the 2nd row and 1st column of the *Filtered Value* matrix (see Fig.21).
The 2nd row of the *Filtered Value* matrix already represents the contextualized or dynamic word embedding vector of the word ”flies” in the *Sample Sentence*. Whereas the ”food” feature value in the *Value* matrix for the word ”flies” was 0.0, this value at the same coordinate in the *Filtered Value* matrix has changed to -1.3. This change in the vector representation of the word ”flies” can be entirely attributed to the negative ”food”

feature value contributions coming from the words ”time” and ”arrow” in the *Value* matrix.
The same analysis is now done for the second of the *sample sentences*: ”fruit flies like a banana” (shown in Fig.22).

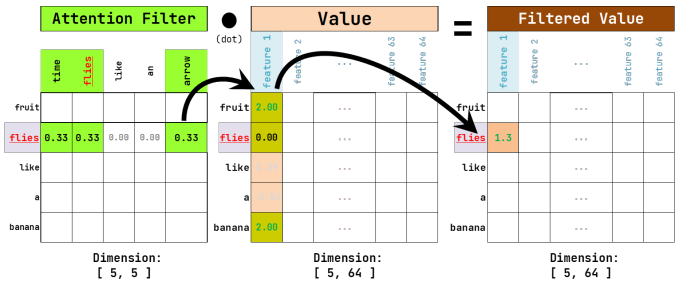

Fig. 22. *Filtered Values: ”fruit flies like a banana”.*

Here, the assumption is that the *Attention Filter* has identified the words ”fruit” and ”banana” to be important context words for the word ”flies”, assigning the same exemplary values of 1/3 to them, like before. The exemplary ”food” feature values for the word ”fruit” and ”banana” are now assumed to be +2.0 as both words are strongly related to the idea of ”food”. The exemplary ”food” feature value for the word ”flies” itself has not changed as it is indirectly coming from the static *Input Embeddings*. Every unique word (representation) in the *Value* matrix is the same for all sentences. The dot product of the 2nd row of the *Attention Filter* with the 1st column of the *Value* matrix **now** yields a positive value of +1.3 (see Fig.22), again coming from a value of 0.0 in the *Value* matrix. This change in the vector representation of the word ”flies” in the *Filtered Value* matrix can be entirely attributed to the positive ”food” feature value contributions coming from the words ”fruit” and ”banana” in the *Value* matrix.
The application of this principle to not only one column (here the ”food” feature column), but to all 64 columns of the *Value* and *Filtered Value* matrices ensures that different semantic and syntactic aspects of word inter-dependencies are accounted for. The *Filtered Value* matrix is where all the attention magic plays out. Whereas the *Value* matrix can be considered a **static** word embedding, the *Filtered Value* matrix truly is a **dynamic** representation of words as it depends on the context words that are identified by the *Attention Filter* and that are different for every sentence. The embedding vector of the word ”flies” in the *Filtered Value* matrix is different for the two *sample sentences* depending on their context.

The *Attention Filter* so extracts or accentuates the most relevant feature values in the *sample sentences*. It can be compared to a *Convolution Kernel* in a Convolutional Neural Network (”CNN”) that does a similar job (see Fig.23).

h) Multiple Heads: The *Filtered Value* matrix is also known as ”head”. BERT, as previously stated, has 12 stacked layers with 12 *heads* each, which sums up to a total of 144 *heads*. The procedure described above thus is carried out 12 times in parallel to get 12 *heads* per layer. Each *Filtered Value*

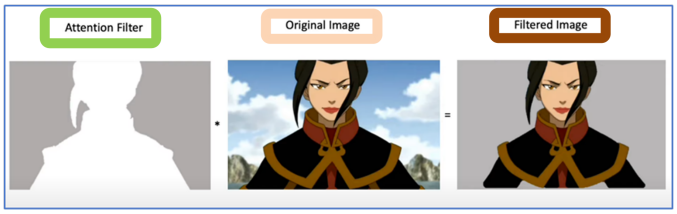

Fig. 23. *Filtered Values*: An *Attention Filter* is comparable to a *Convolution Kernel* in a Convolutional Neural Network (”CNN”). Where the latter highlights some important features in a given image, the former is supposed to highlight some important features of words in a sentence. Image from: [16]

matrix or *head* in our example has a dimension of *[5, 64]*. The 12 *heads* in each layer are horizontally concatenated to arrive at the layer-wise *Output Embedding* matrix with a dimension of *[5, 768]* (see Fig.24). The *Output Embedding* matrix is often referred to as the ”Hidden State”. It has the same dimension as the *Input Embeddings* matrix, and for a good reason: The output of each Encoder layer, i.e. the *Output Embedding* matrix, is the input to the next Encoder layer in the 12-layer Encoder stack.

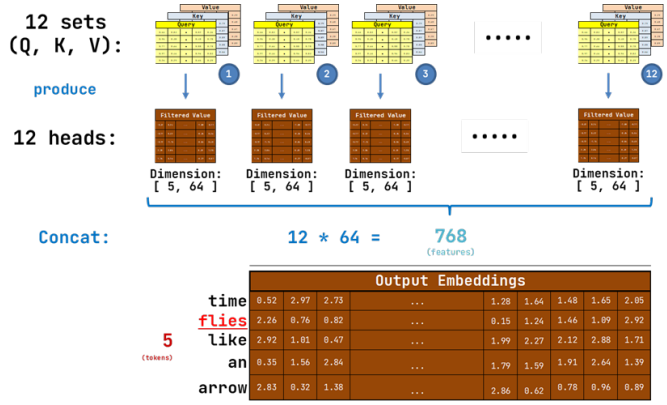

Fig. 24. *Filtered Values*: A *Filtered Value* matrix is also know as ”head”. Within one layer, the 12 *heads* are concatenated horizontally to arrive at the input and output dimension of *[5, 768]*

This gives BERT (in the training phase) the chance to learn not only 64 different features per token (or per word in our example) as discussed above, but 144 times that amount (144 *heads* x 64 features per *head* = 9,216 features). It is assumed though, that the horizontal concatenation of *heads* leaves these features somehow grouped and separated *head*-wise within each *Output Embedding* matrix.

i) Linear Layer in the Multi-Head Attention Module: To activate these respective feature groups for each sentence, the next step in the BERT architecture is a *Linear Layer*, see Fig.25.
The task of the *Linear Layer* probably is to learn common features across the feature groups and to transfer this aggregated knowledge to the final output of each Encoder layer, the *Output Embedding* matrix (on the right in Fig.25). The *Linear Layer* is the last part of the *Multi-Head Attention* module as

shown in Fig.13.

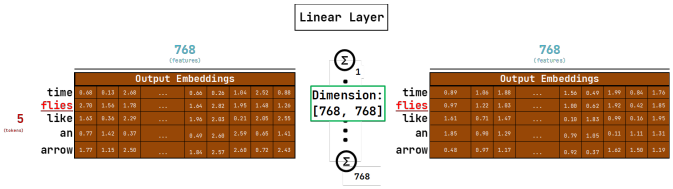

Fig. 25. *Linear Layer*: The *Linear Layer* probably activates those "feature groups" that are still separated in the left *Output Embedding* matrix. The 12 *head* matrices therein were concatenated horizontally, but not logically combined. The task of the *Linear Layer* probably is to learn common features across these feature groups resulting in the *Output Embedding* matrix on the right of this figure. The dimension of *[5, 768]* remains unchanged.

3) Add & Norm 1: The next module in BERT's Encoder block is called "Add & Norm". This module has the goal to preserve the knowledge learned so far and to avoid the vanishing gradient problem. The sequential transformation of the *Input Embeddings* in each step of the layers and modules discussed so far bears the risk that the initial feature, position and segment information goes lost on the way to some extent. To preserve the initial knowledge of the input information, the initial *Input Embeddings* matrix is added element-wise to the the *Output Embeddings* matrix spit out by the *Multi-Head Attention* module (see Fig.26). This "add & norm"-procedure not only applies to the first of the 12 layers in BERT, but to all input embeddings in the 12-layer stack coming in from previous layers.

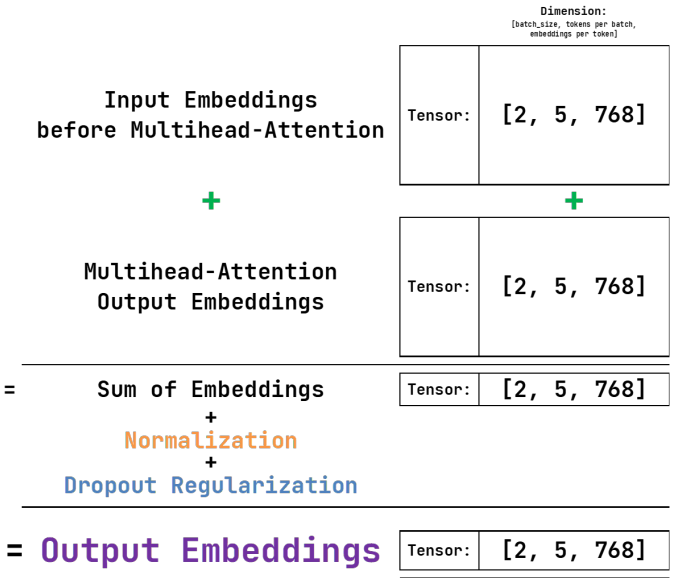

Fig. 26. *Add & Norm 1*: Preserve knowledge and avoid vanishing or exploding gradient problem.

In addition, the resulting matrix is also normalized and "dropout"-regularized to avoid vanishing or exploding gradients in the backward path during the training phase.

4) Feed-Forward Layer: The next big puzzle piece in BERT's Encoder block is the *Feed-Forward Layer*, which is a 2-layer ANN (Fig.27). Except for the "dropout"-regularization in previous steps, it is the first component in BERT's Encoder block that does a non-linear transformation as for the first time an activation function (GELU [2]) is applied after the first ANN layer.

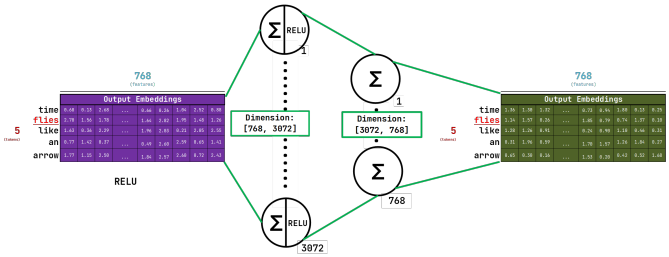

Fig. 27. *Feed-Forward Layer*

This first of the two ANN layers has a dimension of *[768, 3072]* of learnable parameters. It thus transforms the *[5, 768]*-dimensional *Output Embeddings* matrix from before to a *[5, 3072]*-dimensional *Intermediate Embeddings* matrix shown in Fig.28 at the bottom. It so "expands" the number of feature columns of the *Output Embeddings* matrix by a factor of 4. This can be interpreted as an auto-decoder-like operation to extract even more features out of the *Output Embeddings* matrix.

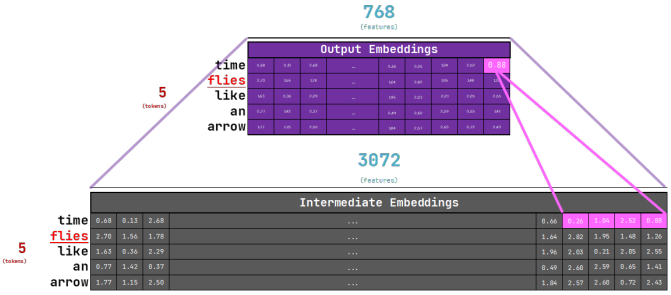

Fig. 28. *Feed-Forward Layer*: Human-interpretable key-value store?

The second of the two ANN layers has a dimension of *[3072, 768]* of learnable parameters. One of its tasks is to reduce the dimension of the *Intermediate Embeddings* matrix back to the desired dimension of *[5, 768]* as depicted in Fig.27. Geva et al [18] in this two-layered ANN see a "Key-Value-Memory" function, where the first ANN layer in a dictionary-like data structure represents the "Keys", and the second ANN layer the corresponding weighted sum of "Values". They show that the patterns in this ANN are even human-interpretable, where lower layers tend to capture shallow patterns, while upper layers learn more semantic ones. The *Feed-Forward Layers* constitute around two-thirds of a transformer model's learnable parameters/weights [18] and thus could indeed serve as some kind of memory cells or even as the numerical representation of human language if applied to the domain of NLP. As this complex topic was not subject in the talk and this paper, it will not be discussed further.

5) Add & Norm 2: The last piece in BERT's Encoder block is another "Add & Norm" module. This module again has the goal to preserve the knowledge learned so far and to avoid the vanishing gradient problem. But instead of preserving the knowledge of the initial *Input Embeddings* matrix, at this level it tries to preserve the information from the *Output Embeddings* spit out by the first "Add & Norm" module ("Add & Norm 1", see above). This "add & norm"-procedure again not only applies to the first of the 12 layers in BERT, but to all of them.

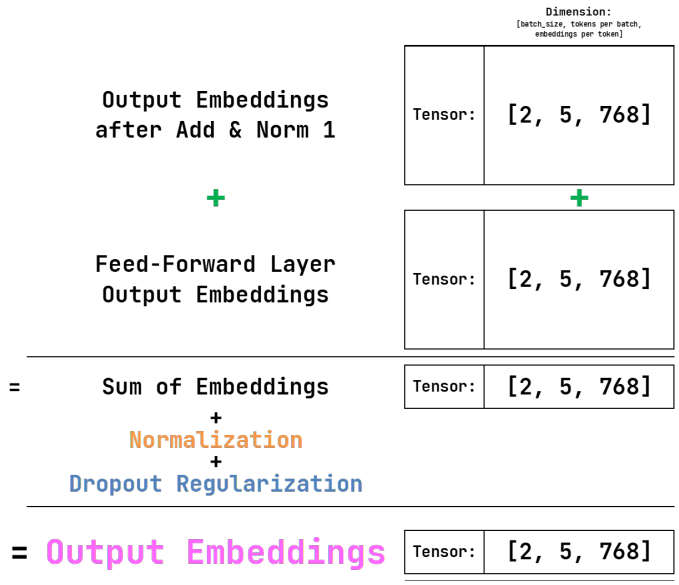

Fig. 29. *Add & Norm 2*: Again: Preserve knowledge and avoid vanishing or exploding gradient problem like in *Add & Norm 1*.

6) 12 Encoder Blocks: So far, this paper discussed how just one Encoder block of BERT works. As previously stated, BERT has 12 such Encoder blocks that are stacked vertically as shown in Fig.30. The output of one Encoder block, i.e. an *Output Embeddings* matrix, serves as the input to the next higher Encoder block in the stack. Each *Output Embedding* matrix in each of the 12 layers has (hopefully) learned different features of human language in general in an upstream task (training). BERT thus can be applied to any general language-related Machine Learning ("ML") downstream task (prediction).

If the language of the task or the task itself is domain-specific, additional steps might be helpful. Such a task is to classify the sentiment of Twitter text messages, as there the sentences are shorter and the language contextually different from the text corpus BERT was trained on [2].

C. Sentiment Classification of Twitter messages

In such a case, it is common to use a pre-trained BERT model, freeze the learned weights/parameters there and only train an added ANN-layer with task-specific settings and domain-specific training data.

The training data used was the "emotion" data set downloaded from the Hugging Face [15] "datasets" package . The data set

Fig. 30. BERT: 12 Encoder blocks: The output of one Encoder block serves as the input to the next higher Encoder block in the stack.

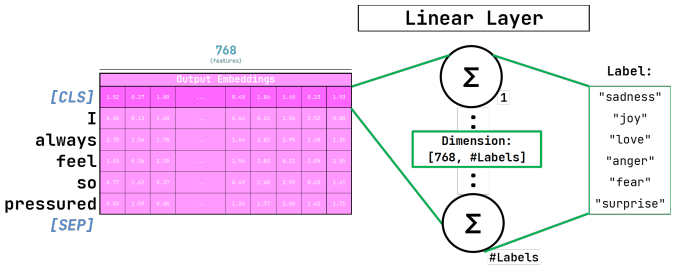

Fig. 31. Fine-tuning a pre-trained BERT model: Freeze all weights/parameters of BERT and only train an added *Linear Layer* with a domain-specific data set.

contains "train", "test" and "validation" batches. The "train" batch contains 2,000 Twitter short messages that are labeled with one of the six sentiment classes: "sadness", "joy", "love", "anger", "fear" and "surprise". The added ANN-layer thus has a dimension of *[768, 6]* with the number of columns

there representing the number of classes, see Fig.31. To extract the embedding of an entire sentence and not just a single word/token, the classification is done on the special [CLS] token. In our exemplary *sample sentences*, there were no such special tokens, but in actual applications, they are usually added. The abbreviation "CLS" stands for "Class" and this [CLS] token, if added, stands at the first position just before the first word/token in a sentence. As the [CLS] token also goes through the attention mechanism explained earlier, it contains the contextual embedding for the entire sentence. The other token embeddings cannot be used for the sentence classification task as they represent the contextualized embeddings for the token itself, but not for the whole sentence. The PyTorch [14] version of the model was downloaded from Hugging Face [15] and trained on a GPU with the help of the PyTorch "Trainer" and "TrainingArguments" classes. The prediction task was done on the "test" batch of the "emotion" data set and achieved an accuracy of around 95%. The training code and the prediction process was shown and executed in the talk and the code was handed in as part of the project submission.

III. Conclusion

This paper documents the essence of a talk with the subject: "Transformer applications in NLP". It consists of three parts, but mainly focuses on the inner workings of the transformer-based BERT model. It particularly highlights the importance of the attention mechanism and shows how this mechanism solves the problem of static word embeddings found in earlier NLP models such as word2vec. The paper uses concrete exemplary words, sentences and numbers to clarify every transformation step gone on the way through a BERT Encoder module.

Transformer-based models, with the attention mechanism at the core of it, have revolutionized the NLP world and contributed to the current hype of large language models (LLMs) such as ChatGPT. Although some of the inner workings of such models are known, many questions remain. This not only applies to specific aspects such as whether the Feed-Forward layers in BERT are really key-value stores of semantic features, but also to more general and philosophic questions such as how exactly the knowledge of human language is represented by such models. I conclude with the notion that this is an exiting and fascinating field that probably still has a lot of surprises and wonders to reveal.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/pdf/1706.03762.pdf

[2] J. Devlin, M.-W. Chang, K. Lee, and K. N. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: https://arxiv.org/abs/1810.04805

[3] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[5] S. Ozdemir and D. Susarla, *Feature Engineering Made Easy: Identify unique features from your dataset in order to build powerful machine learning systems*. Packt Publishing Ltd, 2018.

[6] M. Meeter, Y. Marzouki, A. E. Avramiea, J. Snell, and J. Grainger, "The role of attention in word recognition: Results from ob1-reader," *Cognitive science*, vol. 44, no. 7, p. e12846, 2020.

[7] H. Widdowson, "Jr firth, 1957, papers in linguistics 1934–51," *International Journal of Applied Linguistics*, vol. 17, no. 3, pp. 402–413, 2007.

[8] M. E. Peters, W. Ammar, C. Bhagavatula, and R. Power, "Semi-supervised sequence tagging with bidirectional language models," *arXiv preprint arXiv:1705.00108*, 2017.

[9] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.

[10] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations. arxiv 2018," *arXiv preprint arXiv:1802.05365*, vol. 12, 2018.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[12] X. Song, A. Salcianu, Y. Song, D. Dopson, and D. Zhou, "Fast wordpiece tokenization," *arXiv preprint arXiv:2012.15524*, 2020.

[13] L. Tunstall, L. Von Werra, and T. Wolf, *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

[14] "Pytorch," https://pytorch.org/, accessed: 2023-02-15.

[15] "Hugging face," https://huggingface.co/, accessed: 2023-02-15.

[16] H. A. M. of Intelligence. Visual guide to transformer neural networks series - episode 2. Youtube. Accessed: 2023-02-15. [Online]. Available: https://www.youtube.com/watch?v=mMa2PmYJlCo

[17] Insects as food. Wikipedia. Accessed: 2023-02-15. [Online]. Available: https://en.wikipedia.org/wiki/Insects_as_food

[18] M. Geva, R. Schuster, J. Berant, and O. Levy, "Transformer feed-forward layers are key-value memories," *arXiv preprint arXiv:2012.14913*, 2020.