KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Hyderabad

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)

III YEAR II SEM

NATURAL LANGUAGE PROCESSING (CM621PE)

UNIT-4 NOTES

Course Objectives: The course will help to

1. Understand NLP concepts.
2. Understand RNN concepts for language modeling.
3. Understand sequence to sequence models.
4. Understand attentions and transformers.
5. Understand NLP applications using deep learning techniques.

Course Outcomes: After learning the concepts of this course, the student is able to

1. Describe NLP concepts and techniques.
2. Explain RNN for NLP
3. Develop sequence to sequence models along with CNN.
4. Demonstrate attentions and transformers for NLP.
5. Implement applications of NLP using deep learning techniques.

UNIT-IV

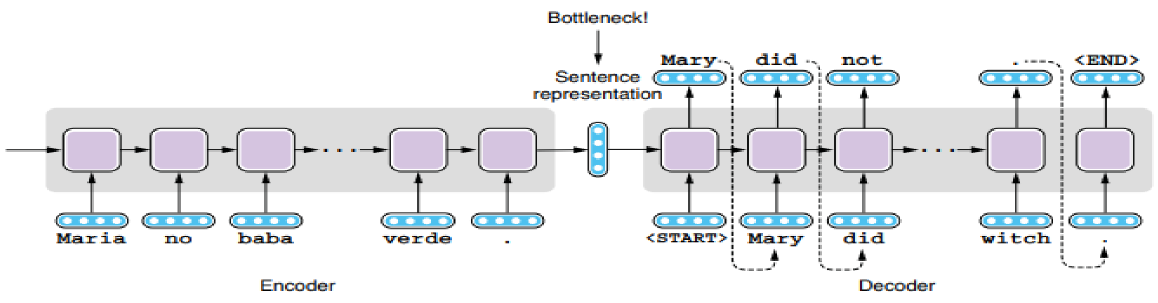
Attention and Transformer: What is attention? - Sequence-to-sequence with attention - Transformer and self-attention - Transformer-based language models - Case study: Spell-checker

Transfer learning with pretrained language models: Transfer learning - BERT - Case study 1: Sentiment analysis with BERT - Other pretrained language models - Case study 2: Natural language inference with BERT

- **The Transformers** —a new type of encoder-decoder neural network architecture based on the concept of self-attention. It is without a doubt the most important NLP model since it appeared in 2017.
- Not only is it a powerful model itself (for machine translation and various Seq2Seq tasks, for example), but it is also used as the underlying architecture that powers numerous modern NLP pretrained models, including GPT-2 and BERT.
- The developments in modern NLP since 2017 can be best summarized as "the era of the Transformer."
- In this chapter, we start with attention, a mechanism that made a breakthrough in machine translation, then move on to introducing self-attention, the concept that forms the foundation of the Transformer model.

What is attention?

- Seq2Seq models—NLP models that transform one sequence to another using an encoder and a decoder. Seq2Seq is a versatile and powerful paradigm with many applications, although the "vanilla" Seq2Seq models are not without limitation.



Limitation of vanilla Seq2Seq models

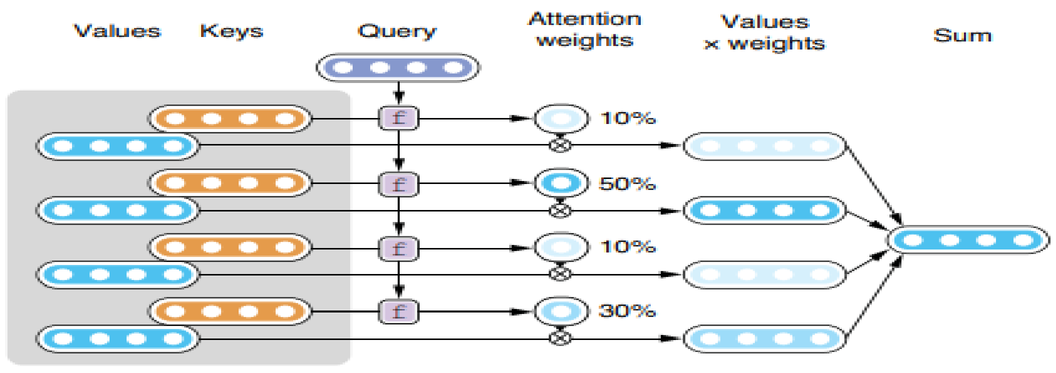
- Seq2Seq models consist of an encoder and a decoder. The decoder takes a sequence of tokens in the source language and runs it through an RNN, which produces a fixed-length vector at the end. This fixed-length vector is a representation of the input sentence. The decoder, which is another RNN, takes this vector and produces a sequence in the target language, token by token.
- This Seq2Seq architecture is quite simple and powerful, but it is known that its vanilla version does not translate sentences as well.

- Its encoder is trying to “compress” all the information in the source sentence into the sentence representation, which is a vector of some fixed length (e.g., 256 floating-point numbers), and the decoder is trying to restore the entire target sentence just from that vector.
- The size of the vector is fixed no matter how long (or how short) the source sentence is. The intermediate vector is a huge bottleneck. If you think of how humans actually translate between languages, this sounds quite difficult and somewhat unusual.
- Compressing all the information into one vector may (and does) work for short sentences, but it becomes increasingly difficult as the sentences get longer and longer.

Attention mechanism Instead of relying on a single, fixed-length vector to represent all the information in a sentence, the decoder would have a much easier time if there was a mechanism where it can refer to some specific part of the encoder as it generates the target tokens.

- This is similar to how human translators (the decoder) reference the source sentence (the encoder) as needed.
- This can be achieved by using attention, which is a mechanism in neural networks that focuses on a specific part of the input and computes its context-dependent summary.
- It is like having some sort of key-value store that contains all of the input’s information and then looking it up with a query (the current context).
- The stored values are not just a single vector but usually a list of vectors, one for each token, associated with corresponding keys.
- This effectively increases the size of the “memory” the decoder can refer to when it’s making a prediction

Figure below illustrates a generic attention mechanism with the following features:



1 The inputs to an attention mechanism are the values and their associated keys. The input values can take many different forms, but in NLP, they are almost always lists of vectors.

For Seq2Seq models, the keys and values here are the hidden states of the encoder, which represent token-by-token encoding of the input sentence.

2. Each key associated with a value is compared against the query using an attention function f. By applying f to the query and each one of the keys, you get a set of scores, one per key-value pair, which are then normalized to obtain a set of attention weights. The specific function f depends on the architecture (more on this later). For Seq2Seq models, this gives you a distribution over the input tokens. The more relevant an input token is, the larger the weight it gets.

3. The input values are weighted by their corresponding weights obtained in step 2 and summed up to compute the final summary vector. For Seq2Seq models, this summary vector is appended to the decoder hidden states to aid the translation process.

- Because of step 3, the output of an attention mechanism is always a weighted sum of the input vectors, but how they are weighted is determined by the attention weights, which are in turn are calculated from the keys and the query.
- In other words, what an attention mechanism computes is a context (query)-dependent summary of the input.
- Downstream components of a neural network (e.g., the decoder of an RNN-based Seq2Seq model, or the upper layers of a Transformer model) use this summary to further process the input.

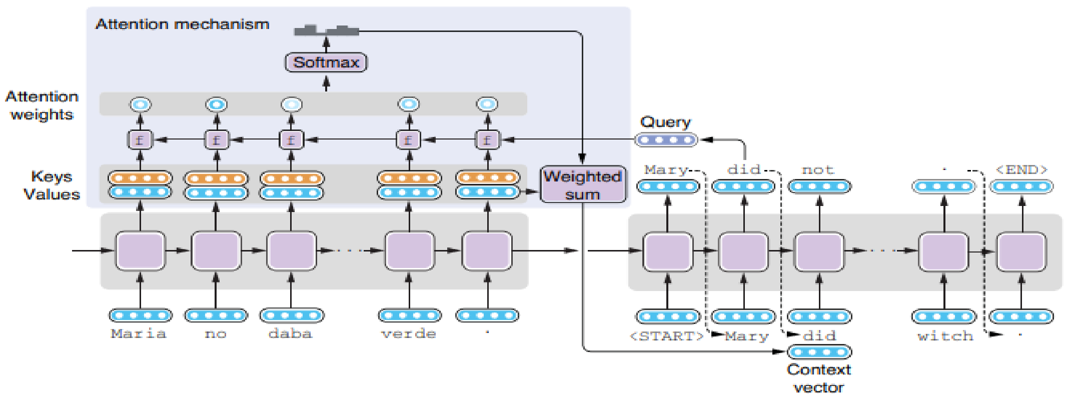
Conclusion: An attention mechanism is a technique used in machine learning and artificial intelligence to improve the performance of models by focusing on relevant information. It allows models to selectively attend to different parts of the input data, assigning varying degrees of importance or weight to different elements.

Encoder-decoder attention

Attention is a mechanism for creating a summary of the input under a specific context.

We used a key-value store and a query as an analogy for how it works.

Figure below illustrates a Seq2Seq model with attention



- It is just an RNN-based Seq2Seq model with some extra "things" added on top of the encoder (the lightly shaded box in top left corner of the figure).
- If we ignore what's inside and see it as a black box, all it does is simply take a query and return some sort of summary created from the input.

1)The input to the attention mechanism is the list of hidden states computed by the encoder.

- These hidden states are used as both keys and values (i.e., the keys and the values are identical).
- The encoder hidden state at a certain token (e.g., at token "no") reflects the information about that token and all the tokens leading up to it (if the RNN is unidirectional) or the entire sentence (if the RNN is bidirectional).

2) Let's say you finished decoding up to "Mary did." The hidden states of the decoder at that point are used as the query, which is compared against every key using function f.

- This produces a list of attention scores, one per each key-value pair.

- These scores determine which part of the input the decoder should attend to when it's trying to generate a word that follows "Mary did."

3) These scores are converted to a probability distribution (a set of positive values that sum to one), which is used to determine which vectors should get the most attention.

- The return value from this attention mechanism is the sum of all values, weighted by the attention scores after normalizing with softmax.

what the attention function f looks like?

A couple of variants of f are possible, depending on how it computes the attention scores between the key and the query, but these details do not matter much here.

- One thing to note is that in the original paper proposing the attention mechanism, the authors used a "mini" neural network to calculate attention scores from the key and the query.

This "mini" network-based attention function is not something you just plug in to an RNN model post hoc and expect it to work.

It is optimized as part of the entire network—that is, as the entire network gets optimized by minimizing the loss function, the attention mechanism also gets better at generating summaries because doing so well also helps the decoder generate better translation and lower the loss function.

In other words, attention is calculating some sort of "soft" word alignment between the source and the target tokens.

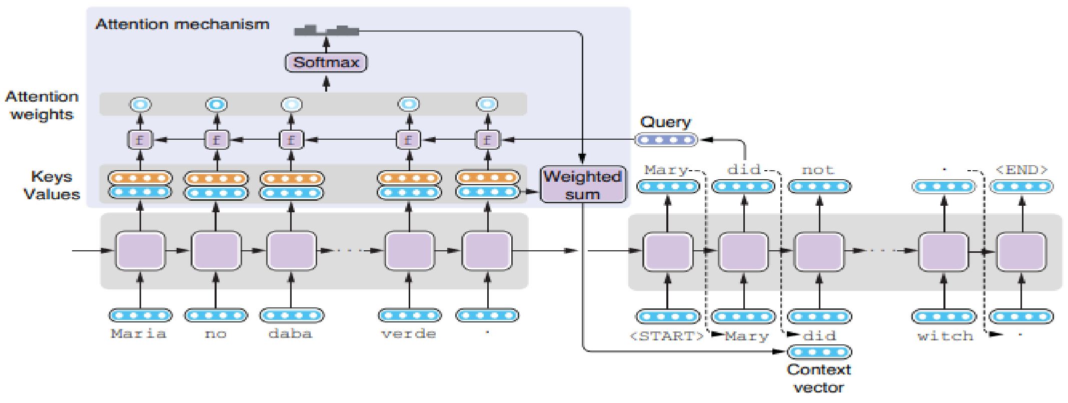
Transformer and self-Attention

Encoder-decoder attention

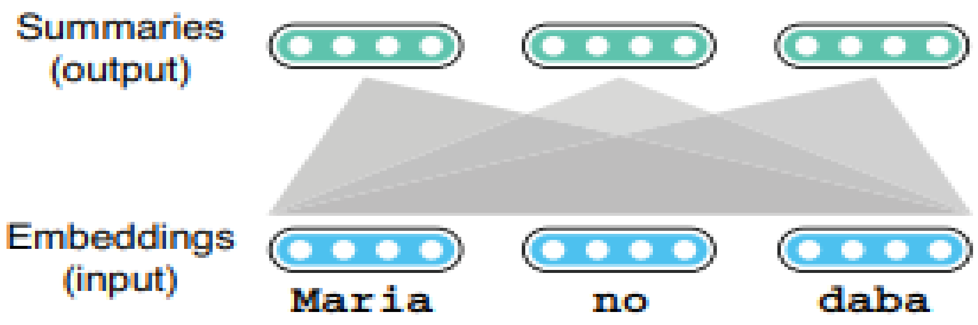
Attention is a mechanism for creating a summary of the input under a specific context.

We used a key-value store and a query as an analogy for how it works.

Figure below illustrates a Seq2Seq model with attention



- How the Transformer model works and, specifically, how it generates high-quality translations by using a new mechanism called selfattention?
- Self-attention creates a summary of the entire input, but it does this for each token using the token as the context.
- Attention is a mechanism that creates a context-dependent summary of the input.
- For RNN-based Seq2Seq models, the input is the encoder hidden states, whereas the context is the decoder hidden states.
- The core idea of the Transformer, self-attention, also creates a summary of the input, except for one key difference—the context in which the summary is created is also the input itself.
- See figure below for a simplified illustration of a self-attention mechanism



Why is this a good thing?

Why does it even work?

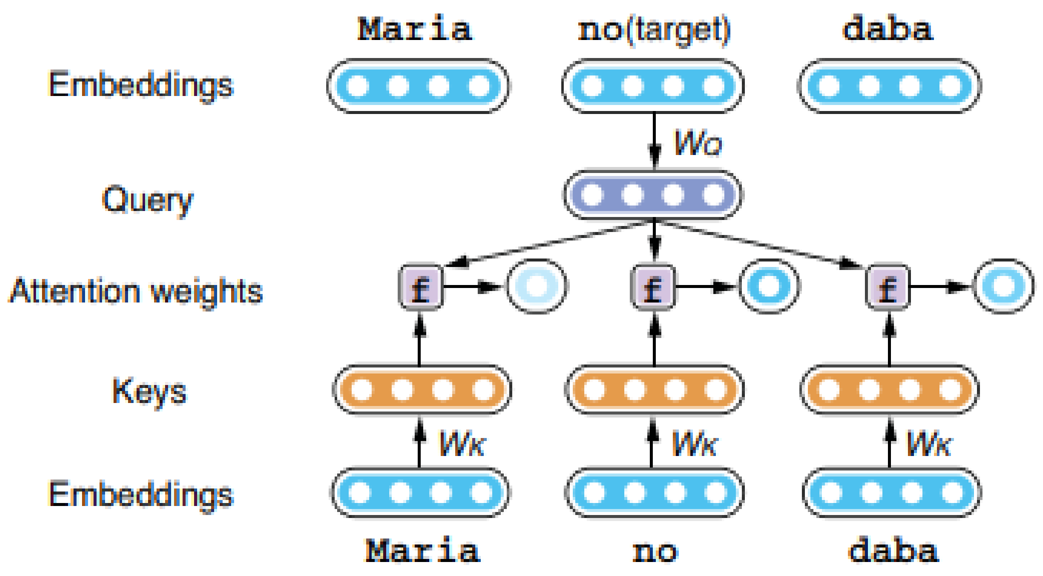
RNNs can also create a summary of the input by looping over the input tokens while updating an internal variable (hidden states). This works—we previously saw that RNNs can generate good translations when combined with attention, but they have one critical issue: because RNNs process the input sequentially, it becomes progressively more difficult to deal with long-range dependencies between tokens as the sentence gets longer.

Example:

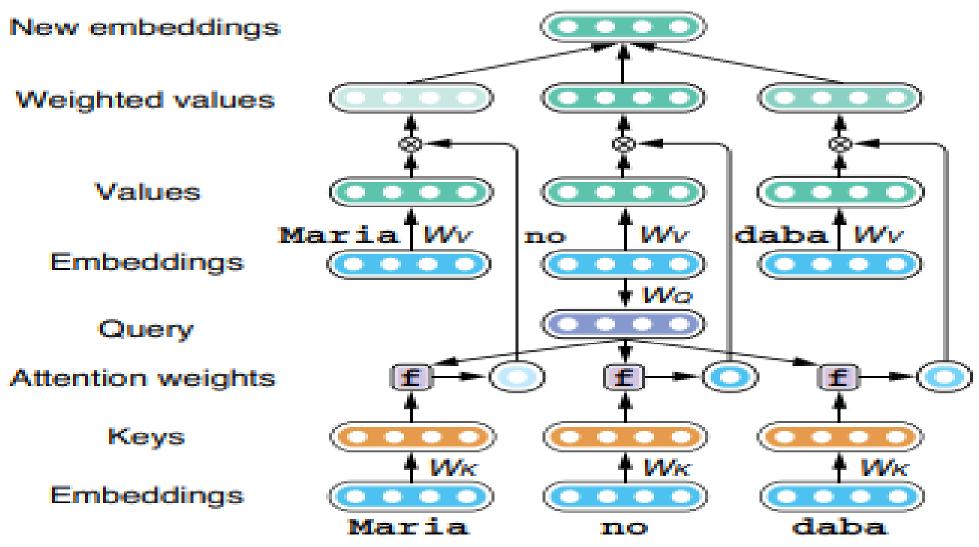
If the input sentence is "The Law will never be perfect, but its application should be just," understanding what the pronoun "its" refers to ("The Law") is important for understanding what the sentence means and for any subsequent tasks (such as translating the sentence accurately).

However, if you use an RNN to encode this sentence, to learn this coreference relationship, the RNN needs to learn to remember the noun "The Law" in the hidden states first, then wait until the loop encounters the target pronoun ("its") while learning to ignore everything unrelated in between. This sounds like a complicated trick for a neural network to learn.

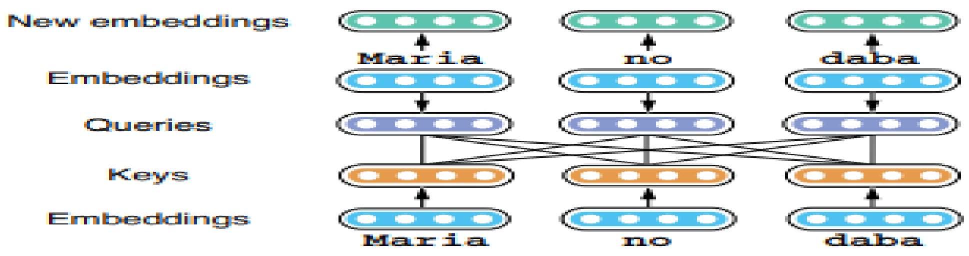
- Let's assume we are translating Spanish into English and would like to encode the first few words, "Maria no daba," in the input sentence. Let's also focus on one specific token, "no," and how its embeddings are computed from the entire input.
- The first step is to compare the target token against all tokens in the input. Self-attention does this by converting the target into a query by using projection WQ as well as converting all the tokens into keys using projection WK and computing attention weights using function f.
- The attention weights computed by f are normalized and converted to a probability distribution by the softmax function.
- Figure below illustrates these steps where attention weights are computed. For words like "its," we expect that the weight will be higher for related words such as "Law" in the example shown earlier.



- In the next step, the vector corresponding to each input token is converted to a value vector by projection WV.
- Each projected value is weighted by the corresponding attention weight and is summed up to produce a summary vector. See figure below for an illustration.

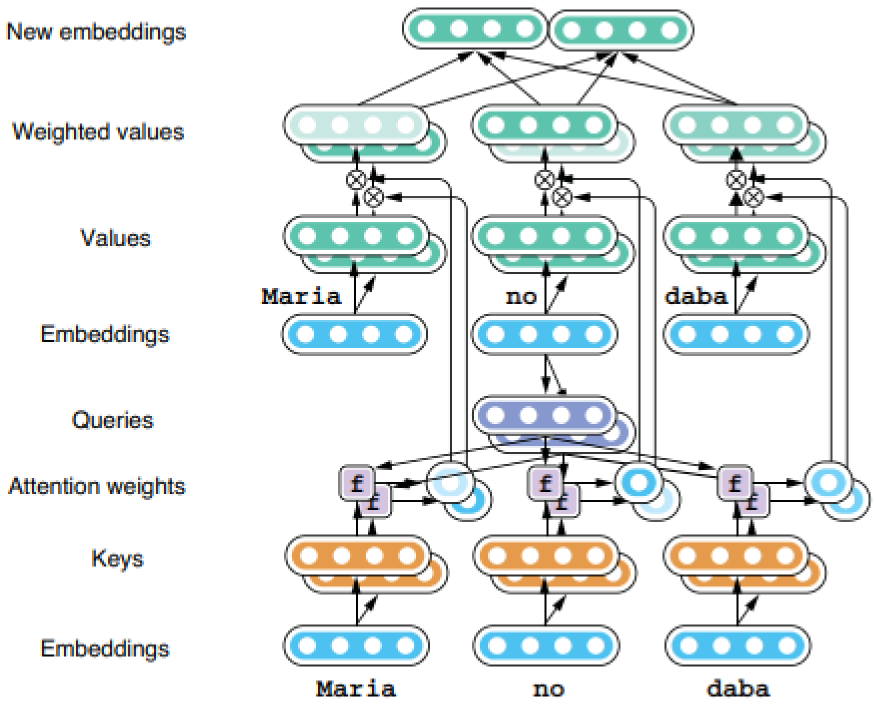


This would be it if this were the “regular” encoder-decoder attention mechanism. You need only one summary vector per each token during decoding. However, one key difference between encoder-decoder attention and self-attention is the latter repeats this process for every single token in the input. As shown in figure below, this produces a new set of embeddings for the input, one for each token



- Each summary produced by self-attention takes all the tokens in the input sequence into consideration, but with different weights.
- It is, therefore, straightforward for words like “its” to incorporate some information from related words, such as “The Law,” no matter how far apart these two words are.
- Using an analogy, self-attention produces summaries through random access over the input.
- This is in contrast to RNNs, which allow only sequential access over the input, and is one of the key reasons why the Transformer is such a powerful model for encoding and decoding natural language text.
- with self-attention mechanism we can use only one aspect of the input sequence to generate summaries.
- If you want self-attention to learn which word each pronoun refers to, it can do that—but you may also want to “mix in” information from other words based on some other linguistic aspects.
- For example, you may want to refer to some other words that the pronoun modifies (“applications,” in this case).
- The solution is to have multiple sets of keys, values, and queries per token and compute multiple sets of attention weights to “mix” values that focus on different aspects of the input.

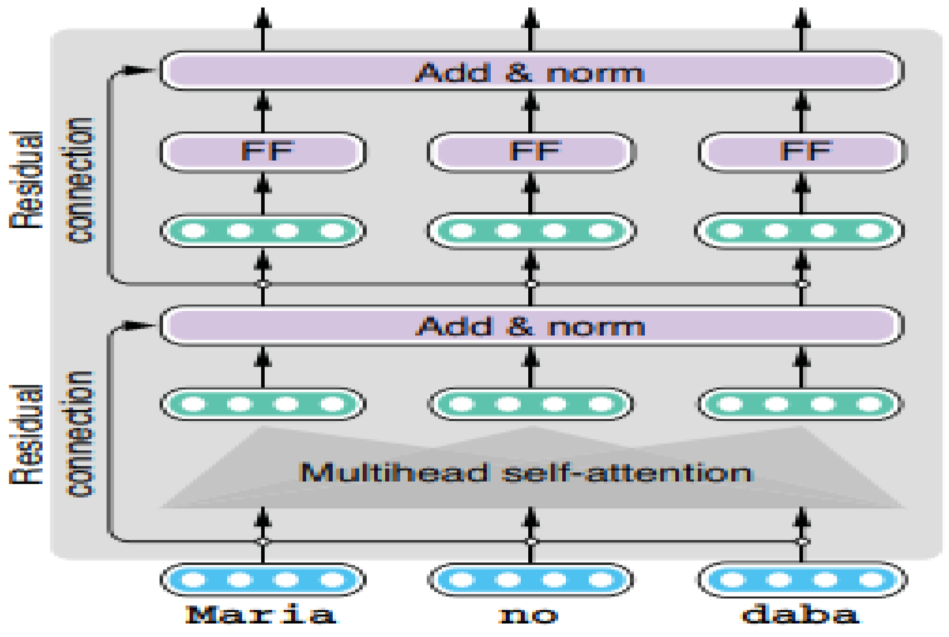
- The final embeddings are a combination of summaries generated this way. This mechanism is called multihead self-attention



Transformer

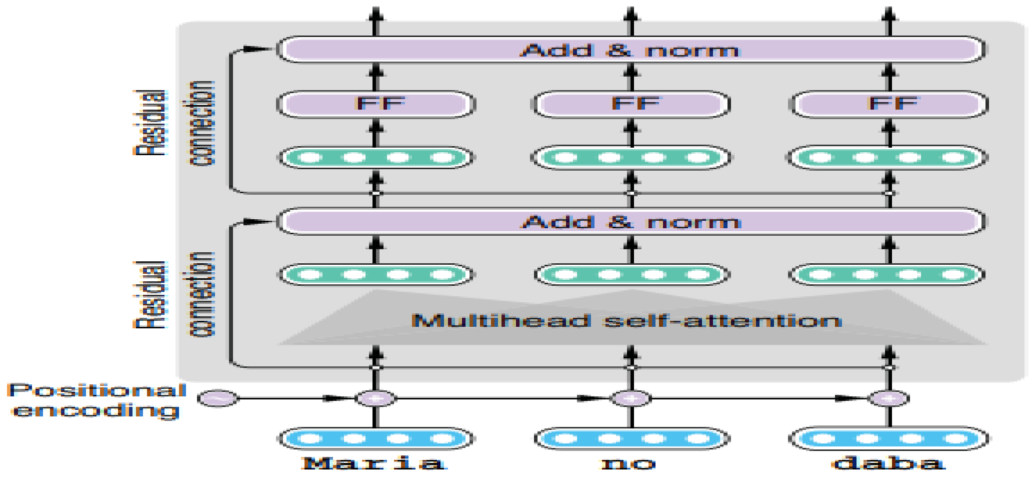
The Transformer model doesn't just use a single step of self-attention to encode or decode natural language text. It applies self-attention repeatedly to the inputs to gradually transform them. As with multilayer RNNs, the Transformer also groups a series of transformation operations into a layer and applies it repeatedly.

Figure below shows one layer of the Transformer encoder.



The self-attention operation is completely independent of positions.

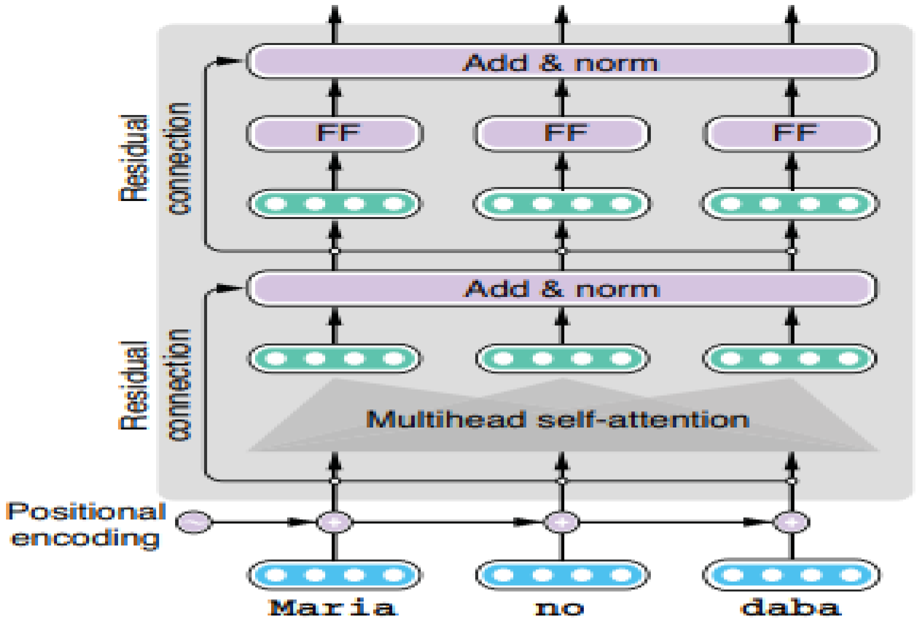
In other words, the embedded results of self-attention would be completely identical even if, for example, we flipped the word order between "Maria" and "daba," because the operation looks only at the word itself and the aggregated embeddings from other words, regardless of where they are. This is obviously very limiting—what a natural language sentence means depends a lot on how its words are ordered.



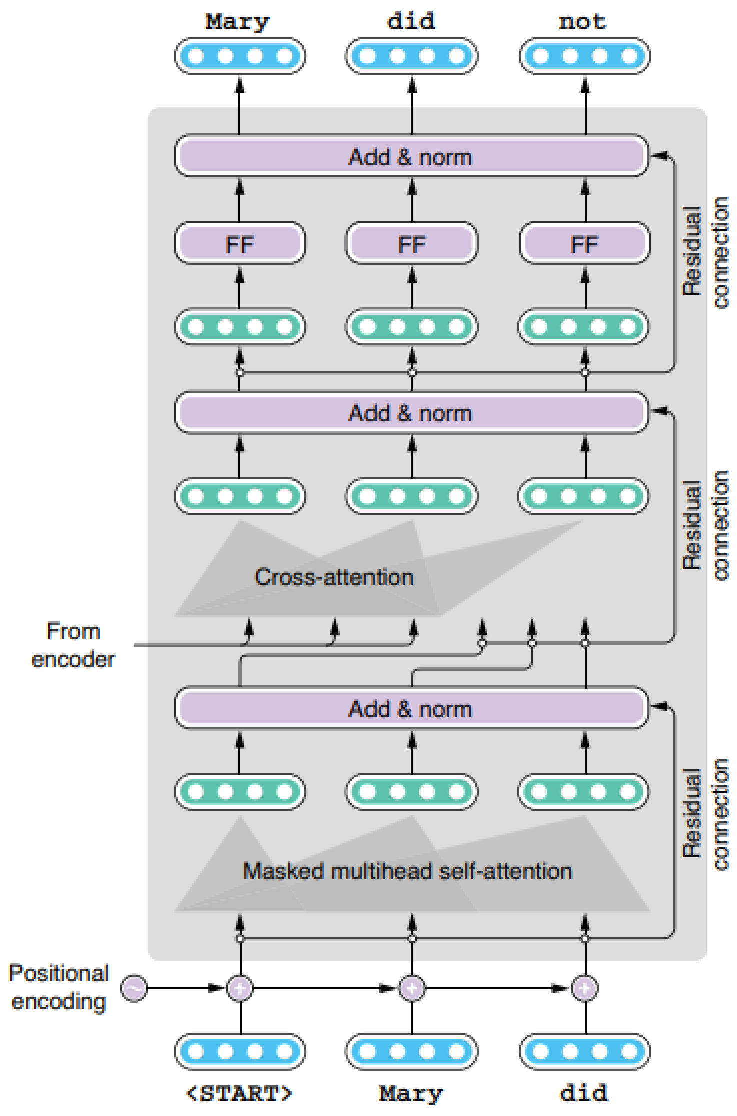
How does the Transformer encode word order, then?

The Transformer model solves this problem by generating some artificial embeddings that differ from position to position and adding them to word embeddings before they are fed to the layers.

These embeddings, called positional encoding and shown in figure below, are either generated by some mathematical function (such as sine curves) or learned during training per position. This way, the Transformer can distinguish between "Maria" at the first position and "Maria" at the third position, because they have different positional encoding



- Figure below shows the Transformer decoder. Although a lot is going on, make sure to notice two important things.
- First, you'll notice one extra mechanism called cross-attention inserted between the self-attention and feed-forward networks.
- This cross-attention mechanism is similar to the encoder-decoder attention mechanism.
- This works exactly the same as self-attention, except that the values for the attention come from the encoder, not the decoder, summarizing the information extracted from the encoder.



- Finally, the Transformer model generates the target sentence in exactly the same way as RNN-based Seq2Seq models we've previously learned.
- The decoder is initialized by a special token and produces a probability distribution over possible next tokens.
- From here, we can proceed by choosing the token with the maximum probability or keeping a few tokens with the highest probability while searching for the path that maximizes the total score.
- In fact, if you look at the Transformer decoder as a black box, the way it produces the target sequence is exactly the same as RNNs, and you can use the same set of decoding algorithms.

Transformer based Language models

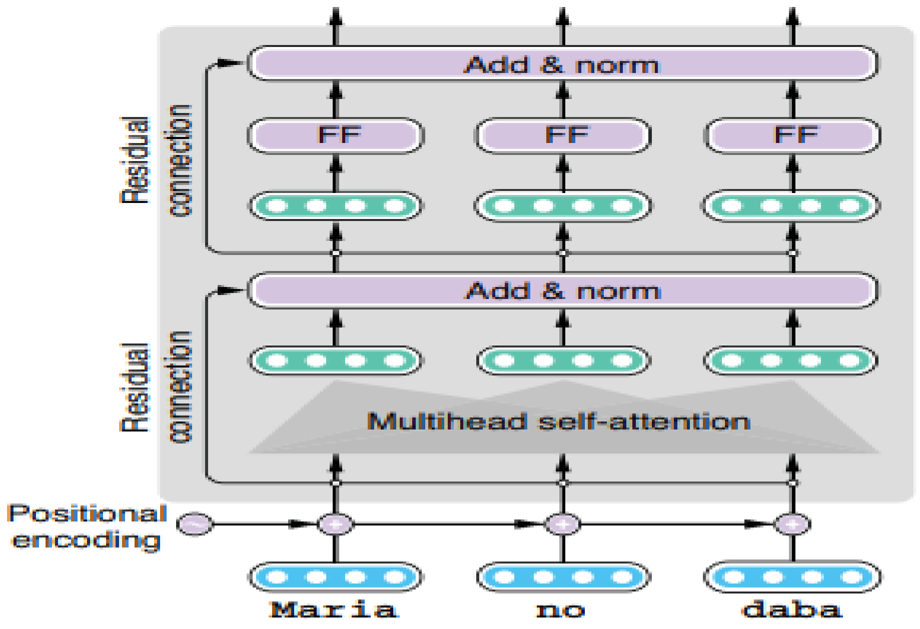
Transformers

The transformer was originally developed to solve the problem of sequence-to-sequence tasks, such as machine translation, but later it has become a foundational model for various natural language processing (NLP) tasks.

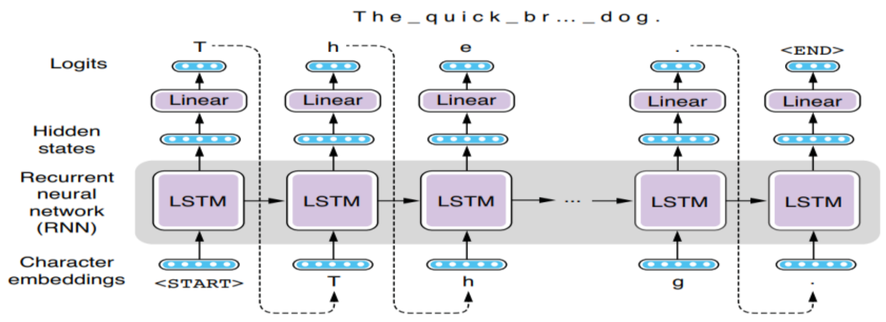
The key features of the Transformer are discussed below:

- **Self-attention:** A distinctive aspect of the transformer is its utilization of a self-attention mechanism, enabling the model to assign varying weights to words in a sequence based on their relevance to one another. This facilitates the parallel capture of long-range dependencies. The implementation involves multiple self-attention heads, providing the model with diverse perspectives on word relationships.

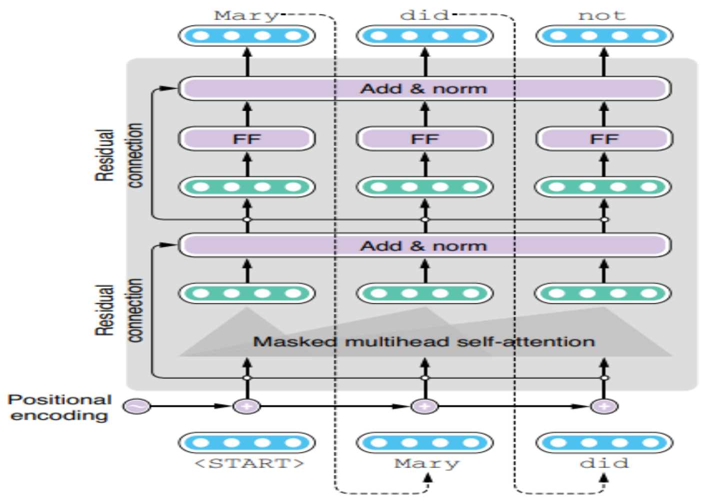
- **Positional Encodings:** Since all the inputs are processed parallelly the transformer layers do not have information about the token sequence. To address the lack of inherent understanding of token sequence order, positional encodings are incorporated into input embeddings.



we built a language-generation model based on a character LSTM-RNN. To recap, given a prefix (a partial sentence generated so far), the model uses an LSTM-based RNN (a neural network with a loop) to produce a probability distribution over possible next tokens, as shown in figure below.



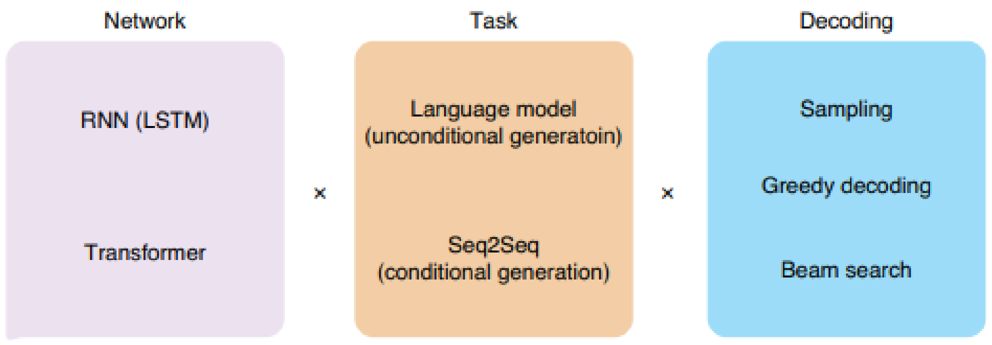
- Figure below shows how an architecture similar to the Transformer can be used for language generation. Except for a few minor differences (such as lack of cross-attention), the structure is almost identical to the Transformer decoder.



Decoding for Seq2Seq models and language generation with language models are very similar tasks, where the output sequence is produced token by token, feeding itself back to the network.

The only difference is that the former has some form of input (the source sentence) whereas the latter does not (the model feeds itself). These two tasks are also called unconditional and conditional generation, respectively.

Figure below illustrates these three components (network, task, and decoding) and how they can be combined to solve a specific problem.



Transformer-XL

Transformer-XL, a variant of the Transformer developed by the researchers at Google Brain.

Because there is no inherent “loop” in the original Transformer model, unlike RNNs, the original Transformer is not good at dealing with super-long context.

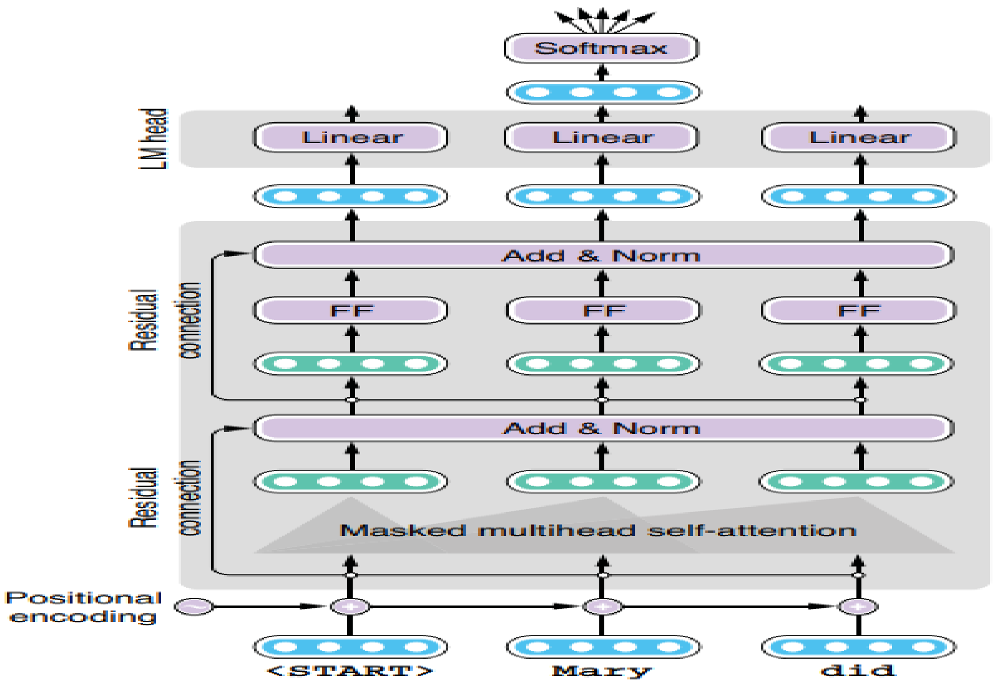
In training language models with the Transformer, you first split long texts into shorter chunks of, say, 512 words, and feed them to the model separately. This means the model is unable to capture dependencies longer than 512 words.

- Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context by Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, Ruslan Salakhutdinov. It’s a causal (uni-directional) transformer with relative positioning (sinusoïdal) embeddings which can reuse previously computed hidden-states to attend to longer context (memory). This model also uses adaptive softmax inputs and outputs (tied).

Performance Transformer XL

As per the paper:

- Transformer-XL can process up to 6,400 tokens in one batch, compared to 512 tokens for the original Transformer. This means that it can capture more long-term dependencies and generate more coherent and diverse texts.
- Transformer-XL learns dependency that is 80% longer than RNNs and 450% longer than vanilla Transformers, achieves better performance on both short and long sequences, and is up to 1,800+ times faster than vanilla Transformers during evaluation.
- Transformer-XL4 addresses this issue by making a few improvements over the vanilla Transformer model (“XL” means extra-long). The model reuses its hidden states from the previous segment, effectively creating a loop that passes information between different segments of texts.
- It also improves the positional encoding scheme we touched on earlier to make it easier for the model to deal with longer texts.



An LM (language model) head is a specific layer added to a neural network that converts its hidden states to a set of scores that determine which tokens to generate next. These scores (also called logits) are then fed to a softmax layer to obtain a probability distribution over possible next tokens

<u>GPT-2</u>

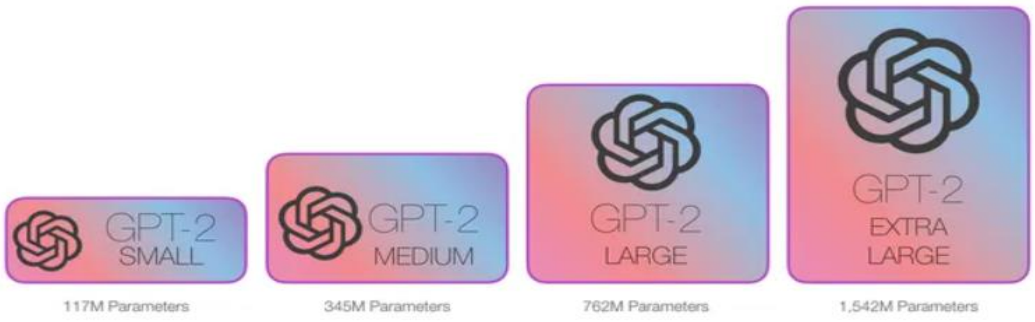
- GPT-2, short for Generative Pre-trained Transformer 2, developed by OpenAI has introduced a revolutionary approach to natural language understanding and text generation through innovative pre-training techniques on a vast corpus of internet text and transfer learning.
- Technically, GPT-2 is just a huge Transformer model, just like the one we introduced earlier. The main difference is its size (the largest model has 48 layers!) and the fact that the model is trained on a huge amount of natural language text collected from the web.
- The OpenAI team publicly released the implementation and the pretrained weights, so we can easily try out the model.

Pre-training and Transfer Learning

- One of GPT-2's key innovations is pre-training on a massive corpus of internet text. This pre-training equips the model with general linguistic knowledge, allowing it to understand grammar, syntax, and semantics across various topics. This model can then be fine-tuned for specific tasks.

Pre-training on Massive Text Corpora

- **The Corpus of Internet Text**
  GPT-2's journey begins with pre-training on a massive and diverse corpus of Internet text. This corpus comprises vast text data from the World Wide Web, encompassing various subjects, languages, and writing styles. This data's sheer scale and diversity provide GPT-2 with a treasure trove of linguistic patterns, structures, and nuances.



- **Equipping GPT-2 with Linguistic Knowledge**
  During the pre-training phase, GPT-2 learns to discern and internalize the underlying principles of language. It becomes proficient in recognizing grammatical rules, syntactic structures, and semantic relationships. By processing an extensive range of textual content, the model gains a deep understanding of the intricacies of human language.
- **Contextual Learning**
  GPT-2's pre-training involves contextual learning, examining words and phrases in the context of the surrounding text. This contextual understanding is a hallmark of its ability to generate contextually relevant and coherent text. It can infer meaning from the interplay of words within a sentence or document

From Transformer Architecture to GPT-2

- GPT-2 is built upon the Transformer architecture, revolutionizing various natural language processing tasks. This architecture relies on self-attention mechanisms, enabling the model to weigh the importance of different words in a sentence concerning each other. The Transformer's success laid the foundation for GPT-2.

<u>XLM</u>

- XLM (cross-lingual language model), proposed by researchers at Facebook AI Research, is a Transformer-based cross-lingual language model that can generate and encode texts in multiple languages.
- By learning how to encode multilingual texts, the model can be used for transfer learning between different languages.
- Cross-lingual Language Models (XLMs) enable natural language processing tasks to be performed across multiple languages, improving performance and generalization in multilingual contexts.
- Cross-lingual Language Models (XLMs) have emerged as a powerful tool for natural language processing (NLP) tasks, enabling models to work effectively across multiple languages.

Practical applications of XLMs include:

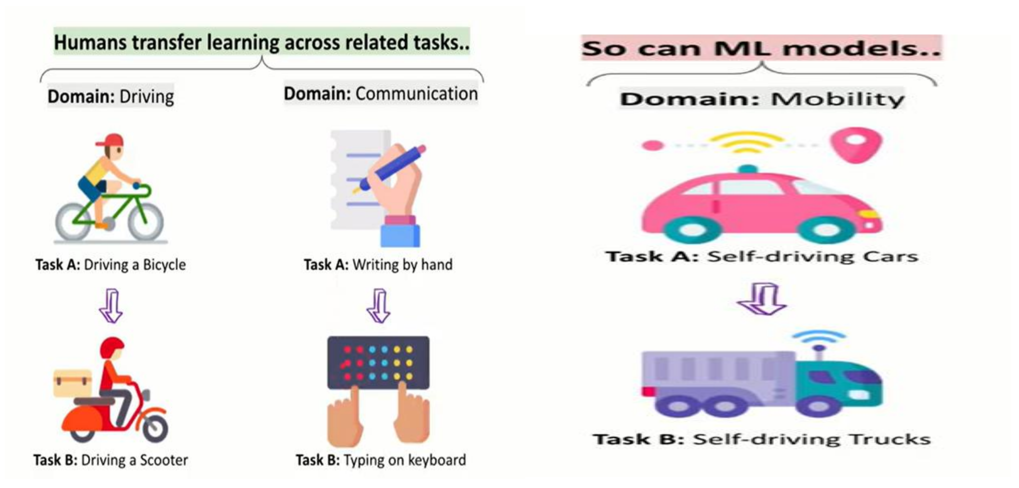
- 1. Multilingual chatbots: XLMs can be used to develop chatbots that understand and respond to user queries in multiple languages, improving user experience and accessibility.
- 2. Cross-lingual sentiment analysis: Companies can use XLMs to analyze customer feedback in different languages, helping them make data-driven decisions and improve their products and services.

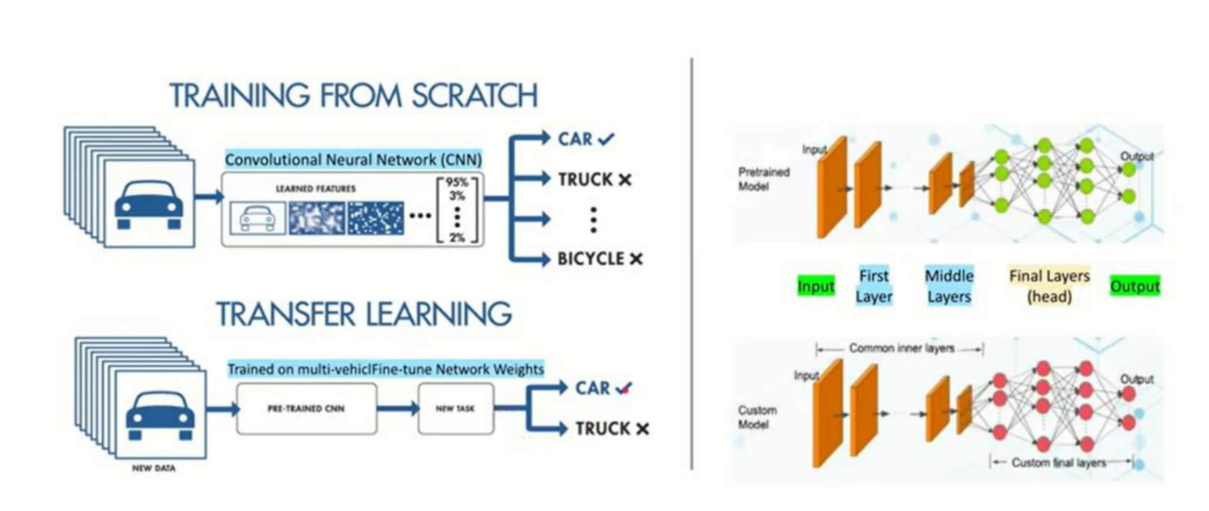
- 3. Machine translation: XLMs can be employed to improve the quality of machine translation systems, enabling more accurate translations between languages.

Transfer learning

Transfer Learning is a machine learning/Deep Learning technique where a model developed for one task is reused as the starting point for a model on a second task.

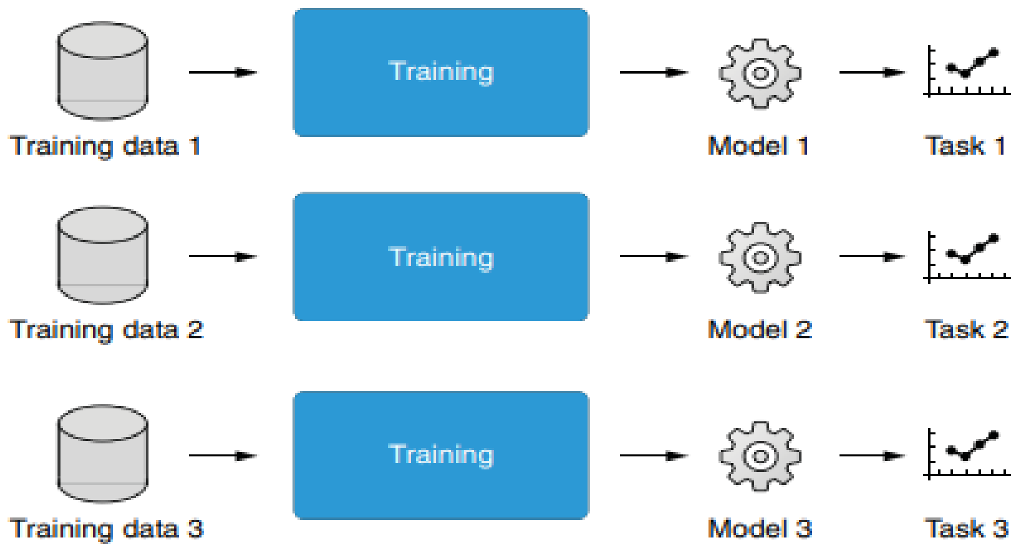
- This approach leverages the knowledge gained while solving one problem and applies it to a different but related problem.
- Transfer learning is particularly useful when the dataset for the target task is small or insufficient to train a model from scratch.


Humans transfer learning across related tasks..
Domain: Driving
Domain: Communication
Task A: Driving a Bicycle
Task A: Writing by hand
Task B: Driving a Scooter
Task B: Typing on keyboard
So can ML models..
Domain: Mobility
Task A: Self-driving Cars
Task B: Self-driving Trucks



Traditional machine learning

- In traditional machine learning, before the advent of pretrained language models, NLP models were trained on a per-task basis, and they were useful only for the type of the task they were trained for as seen in figure below.



Key Concepts of Transfer Learning

1. Pre-training and Fine-tuning:

Pre-training: A model is trained on a large, general dataset to learn broad patterns and representations. This phase aims to develop a comprehensive understanding of the data.

Fine-tuning: The pre-trained model is further trained on a smaller, specific dataset tailored to the target task. This phase refines the model's understanding to perform well on the new task.

2. Feature Transfer:

The initial layers of neural networks often learn general features such as edges, textures, or simple shapes in images. These features are transferable across tasks.

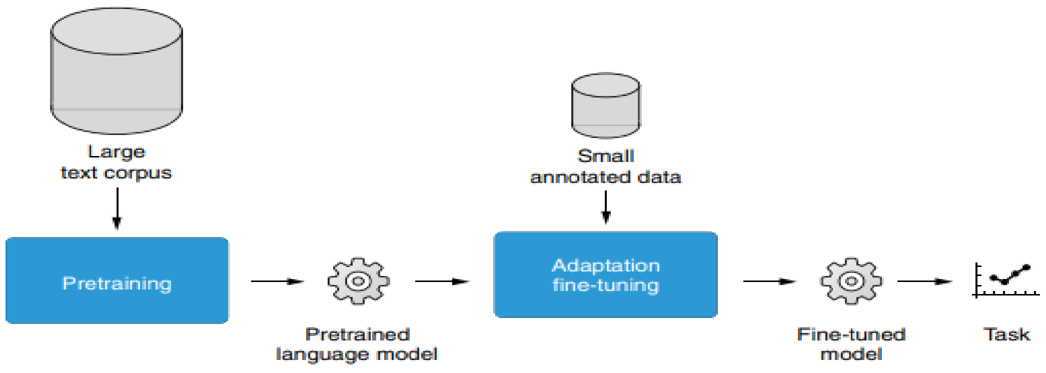
By reusing these layers, the model can quickly adapt to new tasks with fewer training examples.

3. Domain Adaptation:

This involves adapting a pre-trained model to a new domain where the data distribution differs from the original training domain.

Techniques like domain adversarial training are used to minimize the discrepancy between the source and target domains.

- Transfer learning always consists of two or more steps—a machine learning model is first trained for one task **(called pretraining),** which is then adjusted and used in another **(called adaptation).**
- If the same model is used for both tasks, the second step is called **fine-tuning**, because you are tuning the same model slightly but for a different task. See figure below for an illustration of transfer learning in NLP.



- **Note:** The concept called **domain adaptation** is closely related to transfer learning. Domain adaptation is a technique where you train a machine learning model in one domain (e.g., news) and adapt it to another domain (e.g., social media), but these are for the same task (e.g., text classification).

Types of Transfer Learning

1. Inductive Transfer Learning:

The source and target tasks are different, but the domains may be the same.

Example: Using a model pre-trained on ImageNet to classify medical images.

2. Transductive Transfer Learning:

The source and target tasks are the same, but the domains are different.

Example: Adapting a spam detection model trained on English emails to detect spam in French emails.

3. Unsupervised Transfer Learning:

Both tasks and domains are different, but there's no labeled data in the target domain.

Example: Using a pre-trained language model to generate text in a low-resource language

Transfer Learning in NLP

Transfer learning has significantly impacted Natural Language Processing (NLP) by enabling models to perform well on a variety of tasks without needing large annotated datasets for each task.

1. Pre-trained Word Embeddings:

Word2Vec: Learns distributed representations of words by predicting neighboring words in a sentence.

GloVe: Captures global word-word co-occurrence statistics from a corpus to produce dense vector representations.

2. Contextualized Word Embeddings:

ELMo: Generates context-sensitive embeddings for each word in a sentence by considering the entire context.

BERT: Uses a transformer-based architecture to pre-train a bidirectional model, capturing deep contextual relationships between words.

3. Transformers and Language Models:

GPT: A transformer-based model pre-trained on a large corpus and fine-tuned for specific tasks like text generation.

T5: Frames all NLP tasks as a text-to-text problem, allowing the model to be pre-trained on a diverse range of tasks and fine-tuned for specific applications.

Advantages of Transfer Learning

1. Reduced Training Time:

Pre-trained models have already learned fundamental patterns, reducing the need for extensive training from scratch.

2. Improved Performance:

Leveraging knowledge from a large dataset improves the performance on tasks with limited data.

3. Resource Efficiency:

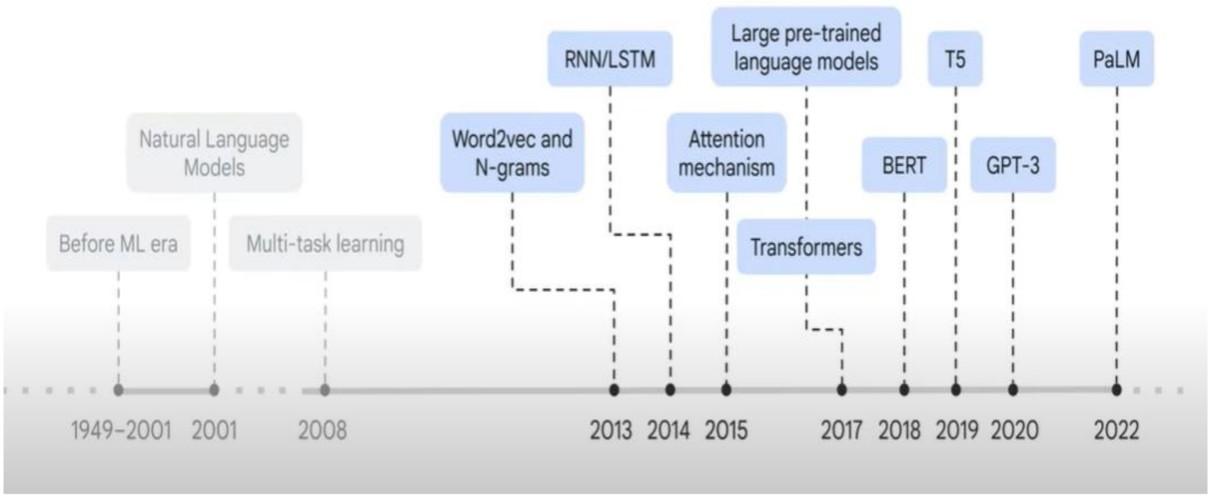
Requires fewer computational resources compared to training a model from scratch.

4. Enhanced Generalization:

Models can generalize better to new tasks and domains by reusing learned features.

BERT

Language modelling history



"**[Attention Is All You Need.](#)**"

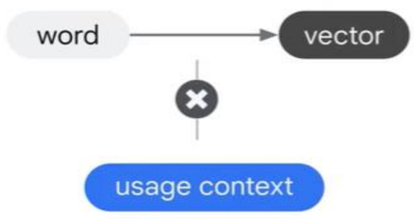
6 Dec 2017

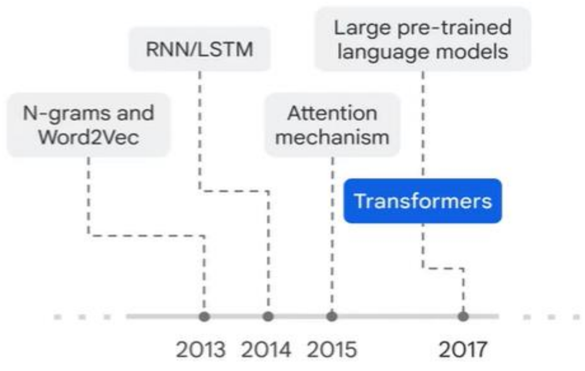
Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

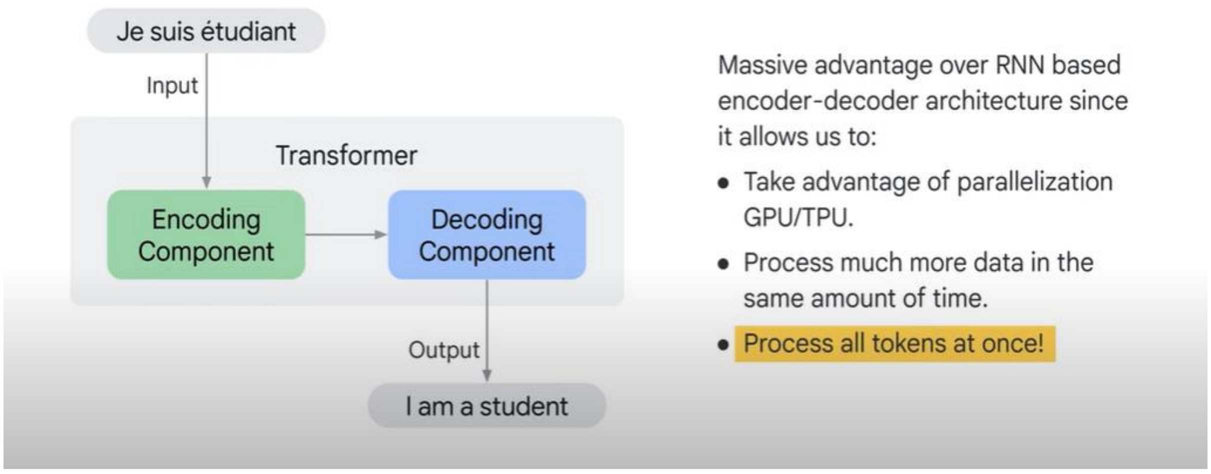
Recurrent neural networks, long short-term memory and gated recurrent neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language modeling and machine translation. Numerous efforts have since continued to push the boundaries of recurrent language models and encoder-decoder architectures.

Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences .





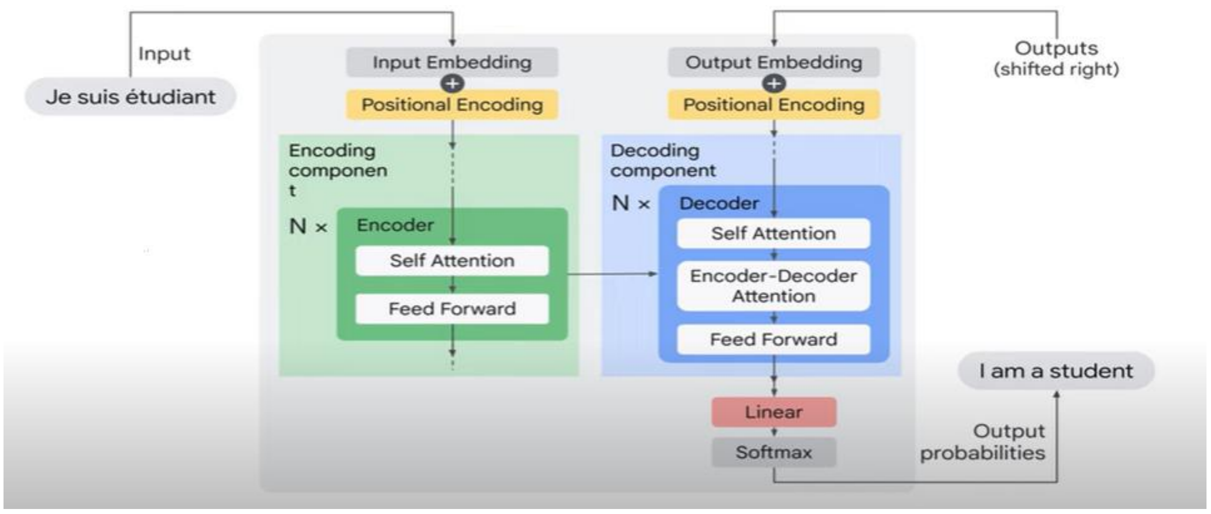
Transformers



Massive advantage over RNN based encoder-decoder architecture since it allows us to:

- Take advantage of parallelization GPU/TPU.
- Process much more data in the same amount of time.
- Process all tokens at once!

Transfomer model



BERT

- BERT (Bidirectional Encoder Representations from Transformers) is by far the most popular and most influential pretrained language model to date that revolutionized how people train and build NLP models.

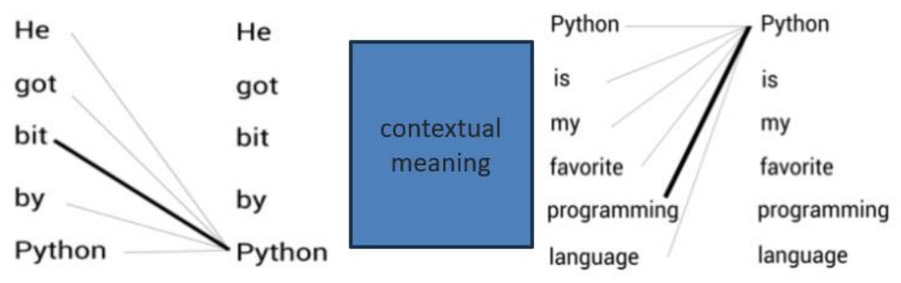
- BERT is a language representation model by Google. It uses two steps, pre-training and fine-tuning, to create state-of-the-art models for a wide range of tasks.
- **BERT is based on the Transformer architecture.**
- BERT is pre-trained on a large corpus of unlabelled text including the entire Wikipedia(that's 2,500 million words!) and Book Corpus (800 million words).
- BERT is a **"deeply bidirectional"** model. Bidirectional means that BERT learns information from both the left and the right side of a token's context during the training phase.

Limitations of word embeddings

- Word embeddings are a powerful concept that can give your application a boost in the performance, although they are not without limitation.
- One obvious issue is that they cannot take context into account. Words you see in natural language are often polysemous, meaning they may have more than one meaning, depending on their context.
- However, because word embeddings are trained per token type, all the different meanings are compressed into a single vector.
- For example, training a single vector for "dog" or "apple" cannot deal with the fact that "hot dog" or "Big Apple" are not a type of animal or fruit, respectively.
- As another example, consider what "play" means in these sentences:
- "They played games," "I play Chopin," "We play baseball," and "Hamlet is a play by Shakespeare" .
- These occurrences of "play" have different meanings, and assigning a single vector wouldn't help much in downstream NLP tasks (e.g., in classifying the topic into sports, music, and art).

Sentence A: He got bit by a Python.

Sentence B: Python is my favorite programming language .

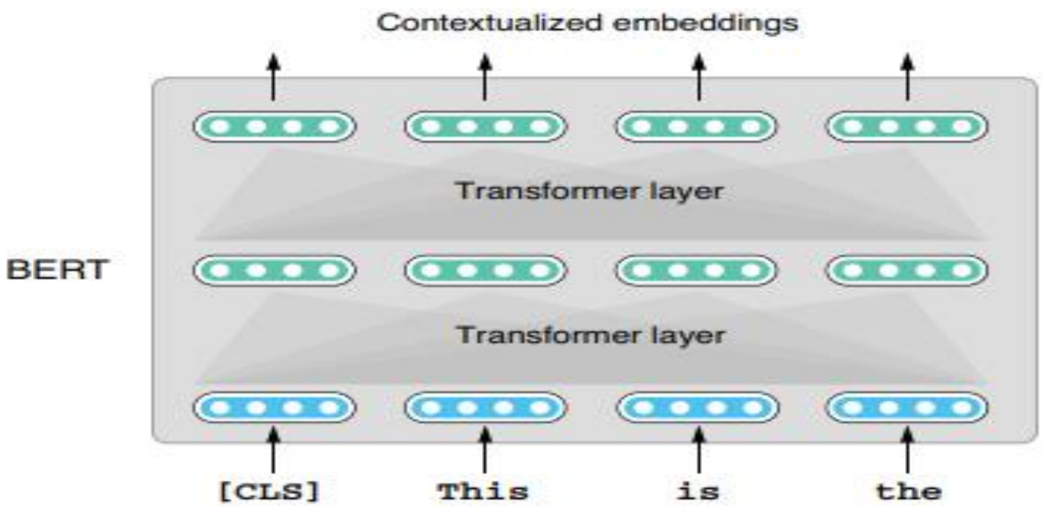


- ❑ Now if get embedding of the word ***"Python"*** using embedding model like word2vec we will get same embedding for both the sentences and therefore will render the meaning of the word in both sentences .
- ❑ This is because word2vec is a context-free model , it will ignore the context and give the same embedding for the word ***"Python"*** irrespective of the context.

- Now if get embedding of the word ***"Python"*** using embedding model like word2vec we will get same embedding for both the sentences and therefore will render the meaning of the word in both sentences .
- This is because word2vec is a context-free model , it will ignore the context and give the same embedding for the word ***"Python"*** irrespective of the context.

The Transformer uses a mechanism called self-attention to gradually transform the input sequence by summarizing it.

- The core idea of BERT is simple: it uses the Transformer (the Transformer encoder, to be precise) to transform the input into contextualized embeddings. The Transformer transforms the input through a series of layers by gradually summarizing the input. Similarly, BERT contextualizes the input through a series of Transformer encoder layers. This is illustrated in figure below



<u>Self-supervised learning</u>

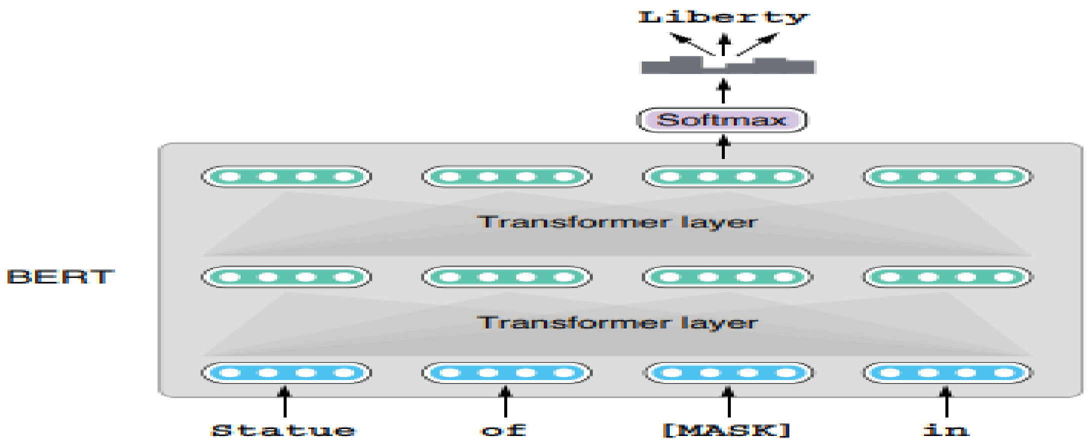
- The Transformer, which was originally proposed for machine translation, is trained using parallel text. Its encoder and decoder are optimized to minimize the loss function, which is the cross entropy defined by the difference between the decoder output and the expected, correct translation.
- However, the purpose of pretraining BERT is to derive high-quality contextualized embeddings, and BERT has only an encoder. How can we "train" BERT so that it is useful for downstream NLP tasks? If you think of BERT just as another way of deriving embeddings, you can draw inspiration from how word embeddings are trained.
- To train word embeddings, we make up a "fake" task where surrounding words are predicted with word embeddings. We are not interested in the prediction but rather the "by-product" of the training, which is the word embeddings derived as the parameters of the model.
- This type of training paradigm where the data itself provides training signals is called **self-supervised learning, or simply self-supervision**, in modern machine learning.
- Self-supervised learning is still one type of supervised learning from the model's point of view—the model is trained in such a way that it minimizes the loss function defined by the

training signal. It is where the training signal comes from that is different. In supervised learning, training signals usually come from human annotations. In self-supervised learning, training signals come from the data itself with no human intervention.

- With increasingly larger datasets and more powerful models, self-supervised learning has become a popular way to pretrain NLP models in the past several years.
- But why does it work so well?
- Two factors contribute to this—one is that the type of selfsupervision here is trivially simple to create (just extracting surrounding words for Word2vec), but it requires deep understanding of the language to solve it.
- For example "My trip to the beach was ruined by bad ___," not only does the system need to understand the sentence but it also needs to be equipped with some sort of "common sense" for what type of things could ruin a trip to a beach (e.g., bad weather, heavy traffic).
- The knowledge required to predict the surrounding words ranges from simple collocation/association (e.g., The Statue of ____ in New ____), syntactic and grammatical (e.g., "My birthday is ___ May"), and semantic (the previous example). Second, there is virtually no limit on the amount of data used for self-supervision, because all you need is clean, plain text. You can download large datasets (e.g., Wikipedia dump) or crawl and filter web pages, which is a popular way to train many pretrained language models

Pretraining BERT

- Now that we all understand how useful self-supervised learning can be for pretraining language models, let's see how we can use it for pretraining BERT. As mentioned earlier, BERT is just a Transformer encoder that transforms the input into a series of embeddings that take context into account.
- For pretraining word embeddings, you could simply predict surrounding words based on the embeddings of the target word. For pretraining unidirectional language models, you could simply predict the next token based on the tokens that come before the target. But for bidirectional language models such as BERT, you cannot use these strategies, because the input for the prediction (contextualized embeddings) also depends on what comes before and after the input.
- The inventors of BERT solved this with a brilliant idea called **masked language model (MLM),** where they drop (mask) words randomly in a given sentence and let the model predict what the dropped word is. Specifically, after replacing a small percentage of words in a sentence with a special placeholder, BERT uses the Transformer to encode the input and then uses a feedforward layer and a softmax layers to derive a probability distribution over possible words that can fill in that blank.



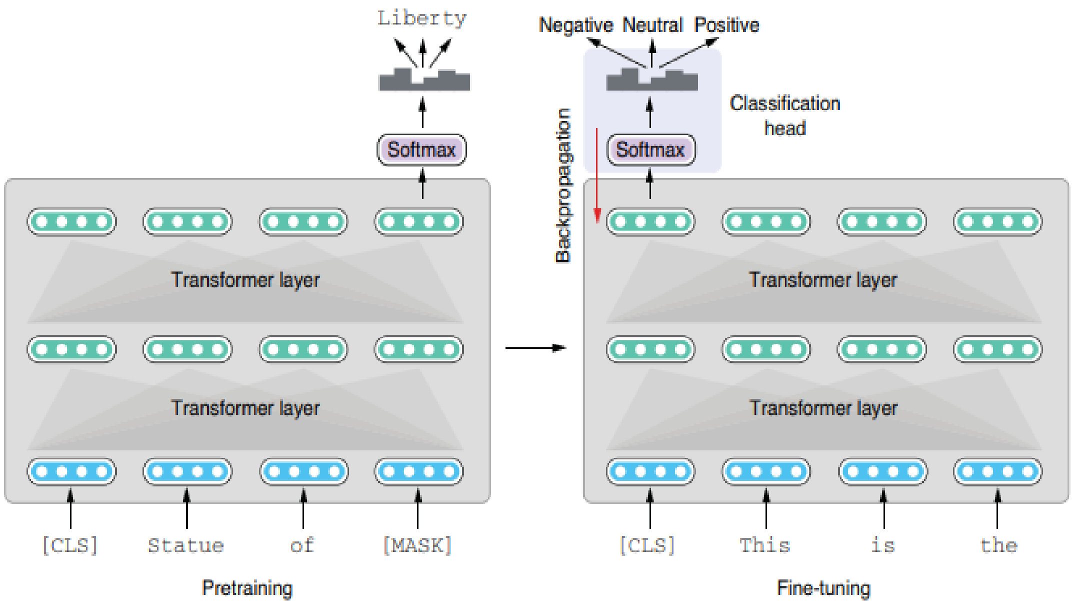
- Masking and predicting words is not a completely new idea—it's closely related to **cloze tests**, where the test-taker is asked to replace the removed words in a sentence.
- Finally, BERT is pretrained not just with the **masked language model** but also with another type of task called **next-sentence prediction (NSP),** where two sentences are given to BERT and the model is asked to predict whether the second sentence is the "real" next sentence of the first.

<u>Adapting BERT</u>

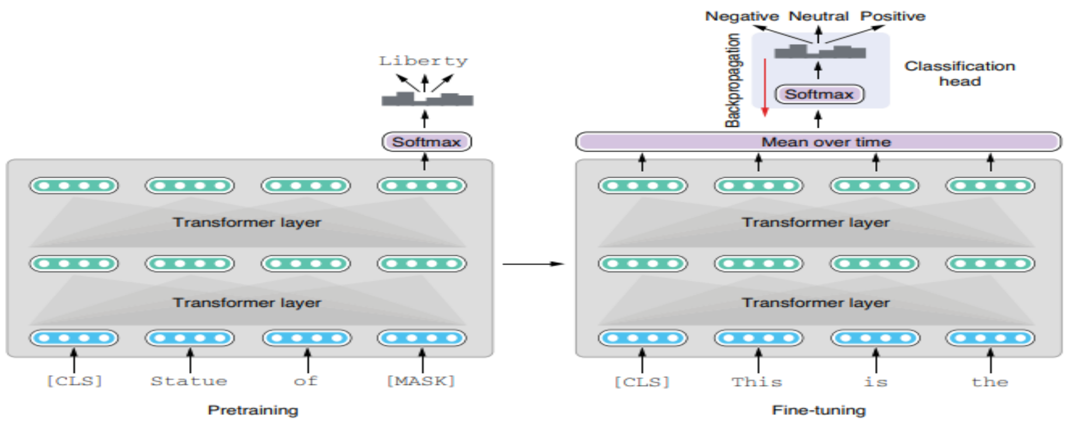
- At the second (and final) stage of transfer learning, a pretrained model is adapted to the target task so that the latter can leverage signals learned by the former
- There are two main ways to adapt BERT to individual downstream tasks: **fine-tuning and feature extraction**
- In fine-tuning, the neural network architecture is slightly modified so that it can produce the type of predictions for the task in question, and the entire network is continuously trained on the training data for the task so that the loss function is minimized. This is exactly the way you train a neural network for NLP tasks, such as sentiment analysis, with one important difference—BERT "inherits" the model weights learned through pretraining, instead of being initialized randomly and trained from scratch. In this way, the downstream task can leverage the powerful representations learned by BERT through pretraining on a large amount of data
- The exact way the BERT architecture is modified varies, depending on the final task.

<u>ex: sentence-prediction task</u>

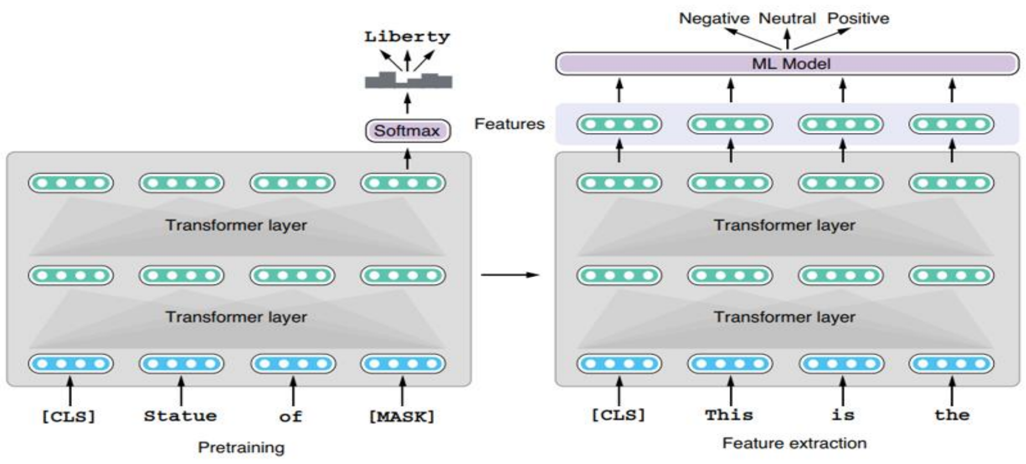
- BERT 227 includes sentiment analysis.
- BERT prepends a special token [CLS] (for classification) to every sentence at the pretraining phase. You can extract the hidden states of BERT with this token and use them as the representation of the sentence. As with other classification tasks, a linear layer can compress this representation into a set of "scores" that correspond to how likely each label is the correct answer. You can then use softmax to derive a probability distribution
- In other words, we are attaching a classification head to BERT to solve a sentence-prediction task. See figure below for an illustration



Another variation in fine-tuning BERT uses all the embeddings, averaged over the input tokens. In this method, called **mean over time or bag of embeddings**, all the embeddings produced by BERT are summed up and divided by the length of input, just like the bag-of-words model, to produce a single vector. This method is less popular than using the CLS special token but may work better depending on the task. Figure below illustrates this.



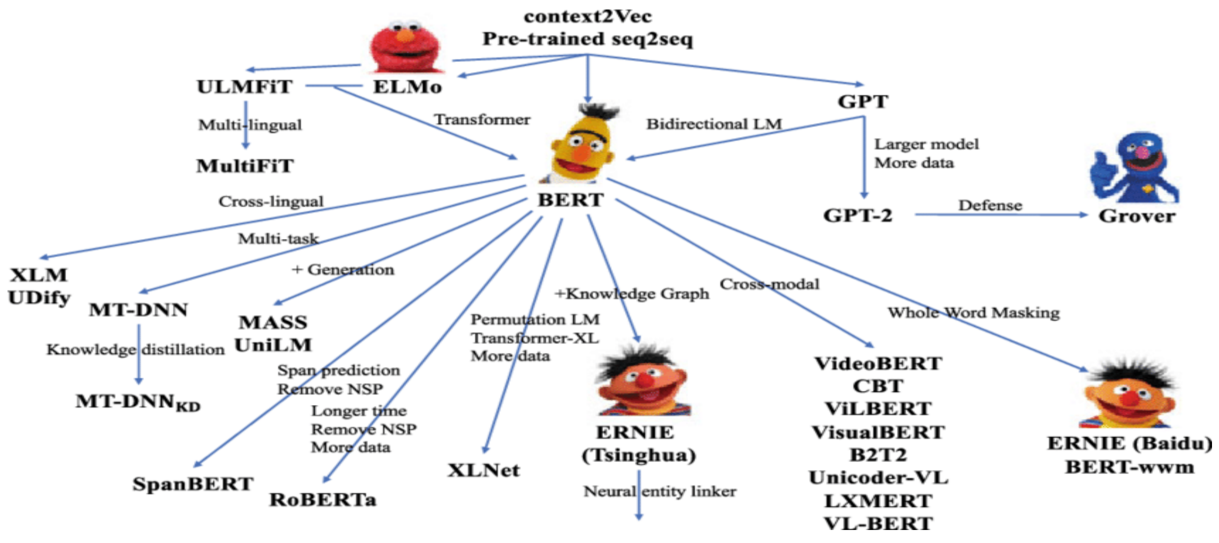
Another way to adapt BERT for downstream NLP tasks is feature extraction. Here BERT is used to extract features, which are simply a sequence of contextualized embeddings produced by the final layer of BERT. You can simply feed these vectors to another machine learning model as features and make predictions, as shown in figure below.



The BERT Family

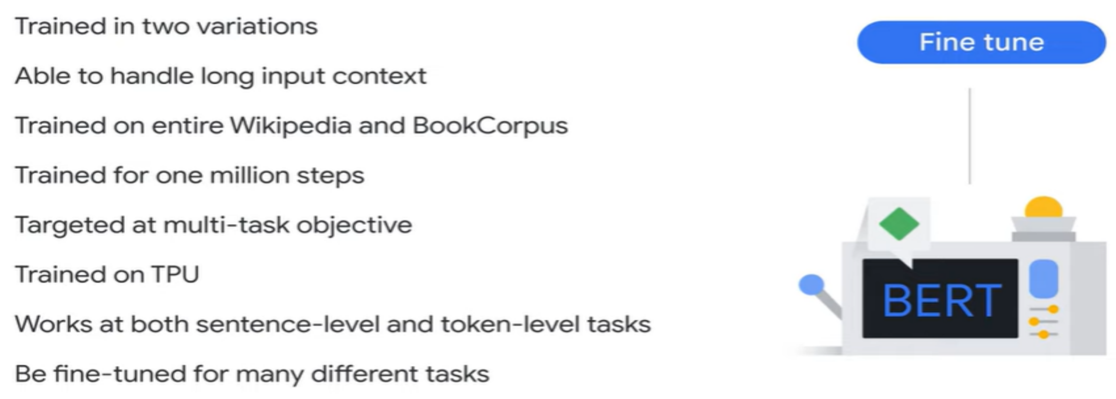
These are some of the most popular variants of BERT:

- **ALBERT** by Google and more — This paper describes parameter reduction techniques to lower memory reduction and increase the training speed of BERT models.
- **RoBERTa** by Facebook — This paper for FAIR believes the original BERT models were under-trained and shows with more training/tuning it can outperform the initial results.
- **ERNIE**: Enhanced Representation through Knowledge Integration by Baidu — It is inspired by the masking strategy of BERT and learns language representation enhanced by knowledge masking strategies, which includes entity-level masking and phrase-level masking.
- **DistilBERT** — Smaller BERT using model distillation from Huggingface.



Sentiment Analysis using BERT

BERT Overview


Trained in two variations

Able to handle long input context

Trained on entire Wikipedia and BookCorpus

Trained for one million steps

Targeted at multi-task objective

Trained on TPU

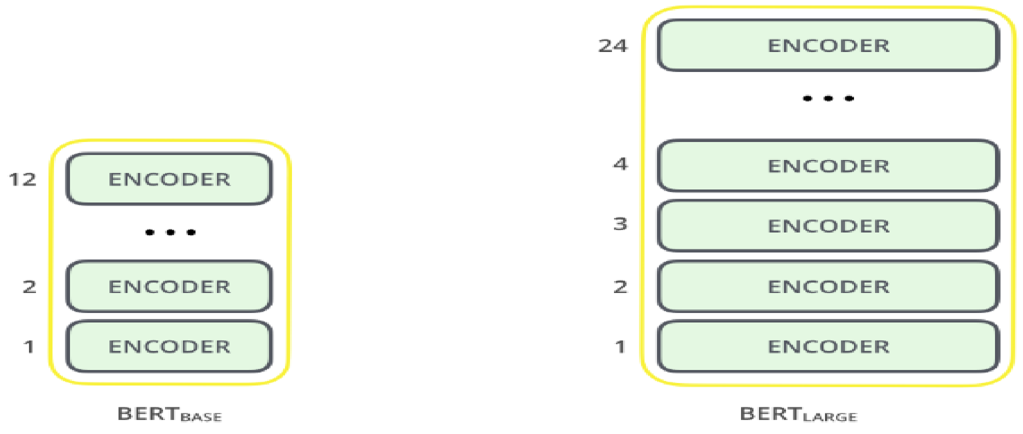
Works at both sentence-level and token-level tasks

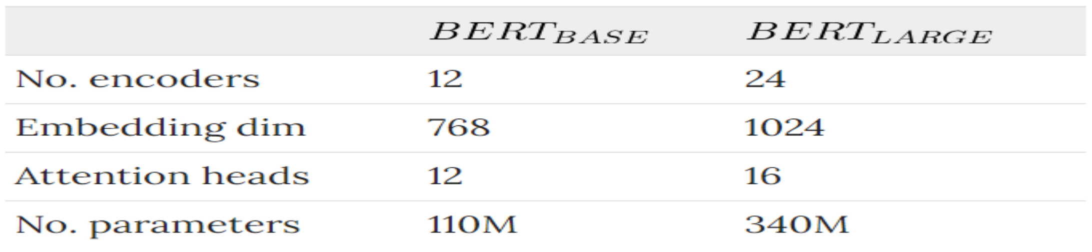
Be fine-tuned for many different tasks

Fine tune
BERT

BERT’s Architecture

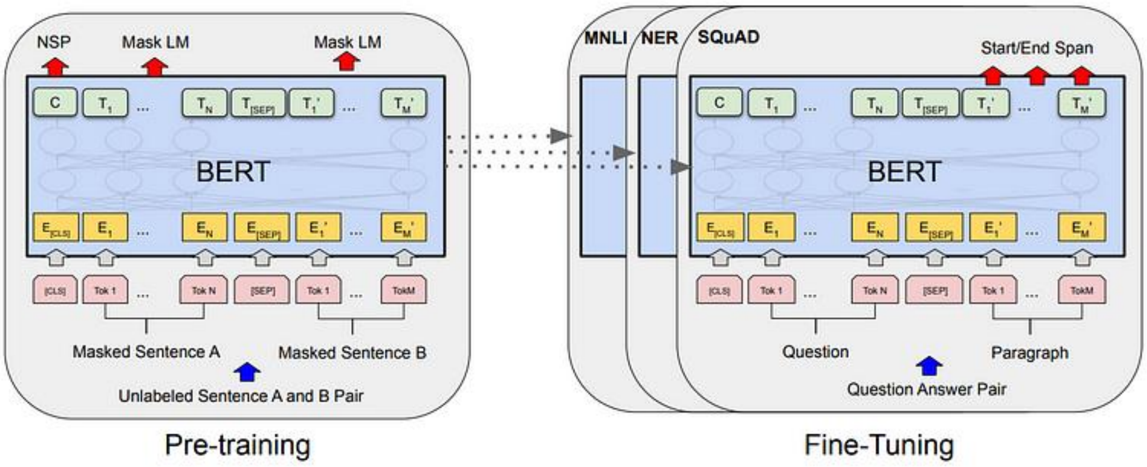
The BERT architecture builds on top of Transformer. We currently have two variants available:





	$BERT_{BASE}$	$BERT_{LARGE}$
No. encoders	12	24
Embedding dim	768	1024
Attention heads	12	16
No. parameters	110M	340M

BERT is a stacked Transformer’s Encoder model. It has two phases — pre-training and fine-tuning.



There are two steps to the BERT framework:

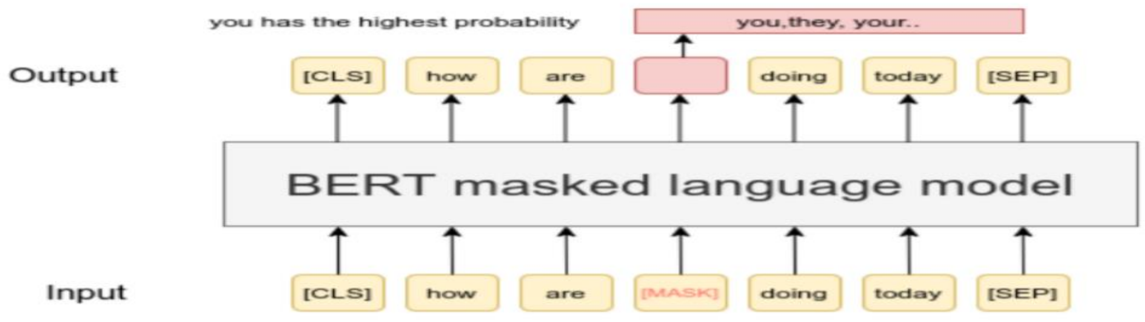
1. Pre-training: The model is trained on unlabeled data over two unsupervised pre-training tasks (Masked Language Modeling and Next Sequence Prediction)
2. Fine-tuning: The BERT model is initialized with pre-trained parameters and these parameters are fine-tuned using labeled data from downstream tasks like Text Similarity, Question Answer pairs, Classification, etc.

BERT is pre-trained on two NLP tasks:

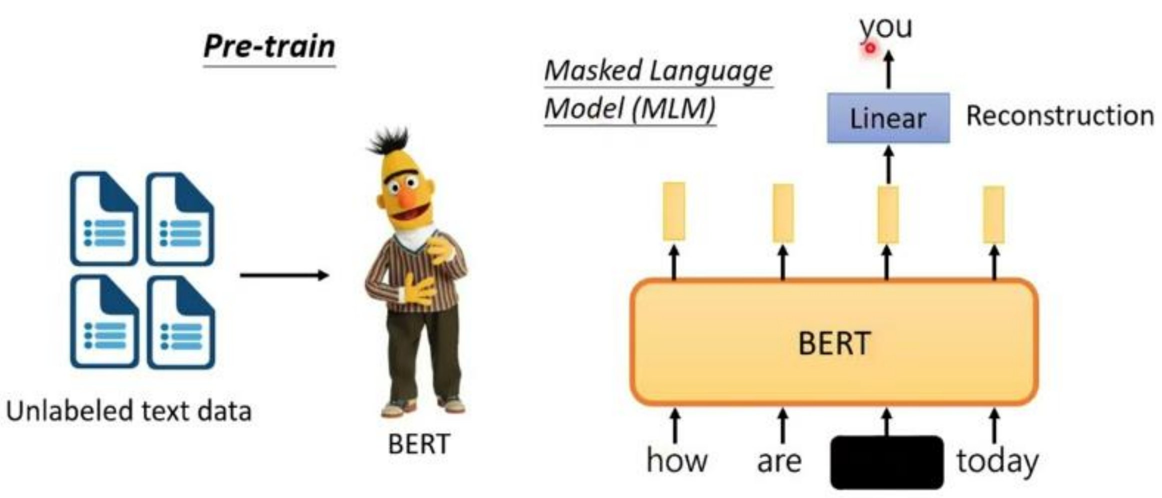
- ❑ Masked Language Modeling
- ❑ Next Sentence Prediction

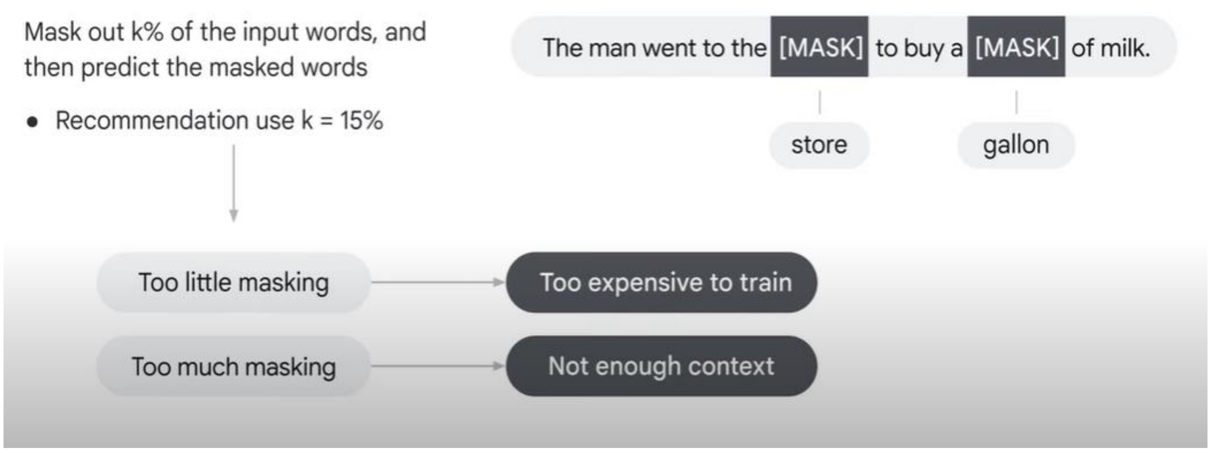
(i) Masked Language Modeling (Bi-directionality)

- **Need for Bi-directionality**
- BERT is designed as a ***deeply bidirectional*** model. The network effectively captures information from both the right and left context of a token from the first layer itself and all the way through to the last layer.



The figure above describes Masked Language Modelling that BERT uses. It masks a few input words and computes the word probability for a pool of candidate words and assigns it the word with the highest probability.




Mask out k% of the input words, and then predict the masked words

- Recommendation use k = 15%

The man went to the [MASK] to buy a [MASK] of milk.

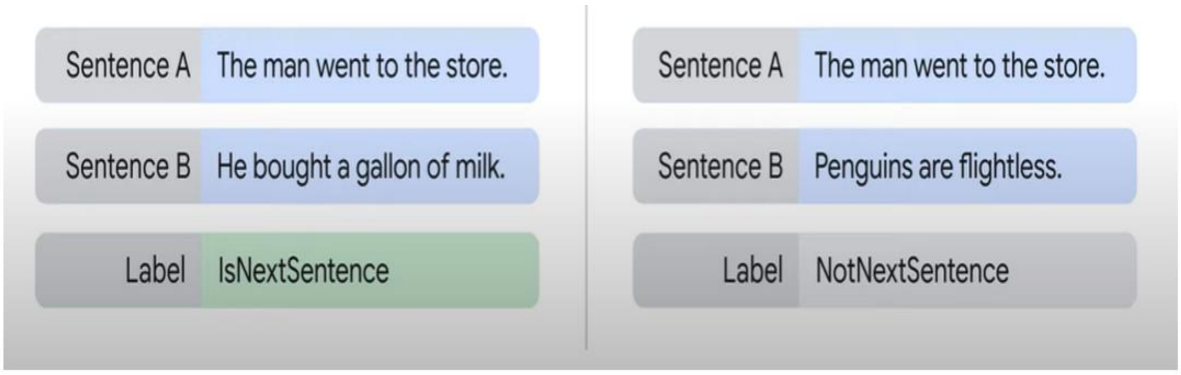
store gallon

Too little masking → Too expensive to train

Too much masking → Not enough context

(ii) Next Sentence Prediction

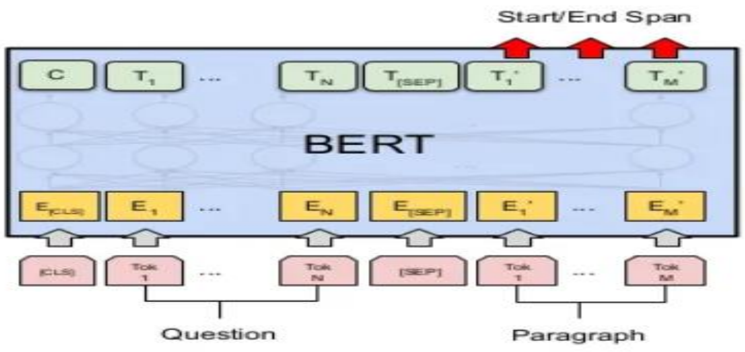
- Masked Language Models (MLMs) learn to understand the relationship between words.
- Additionally, **BERT is also trained on the task of Next Sentence Prediction for tasks that require an understanding of the relationship between sentences.**



Sentence A	The man went to the store.
Sentence B	He bought a gallon of milk.
Label	IsNextSentence

Sentence A	The man went to the store.
Sentence B	Penguins are flightless.
Label	NotNextSentence

Fine tuning tasks



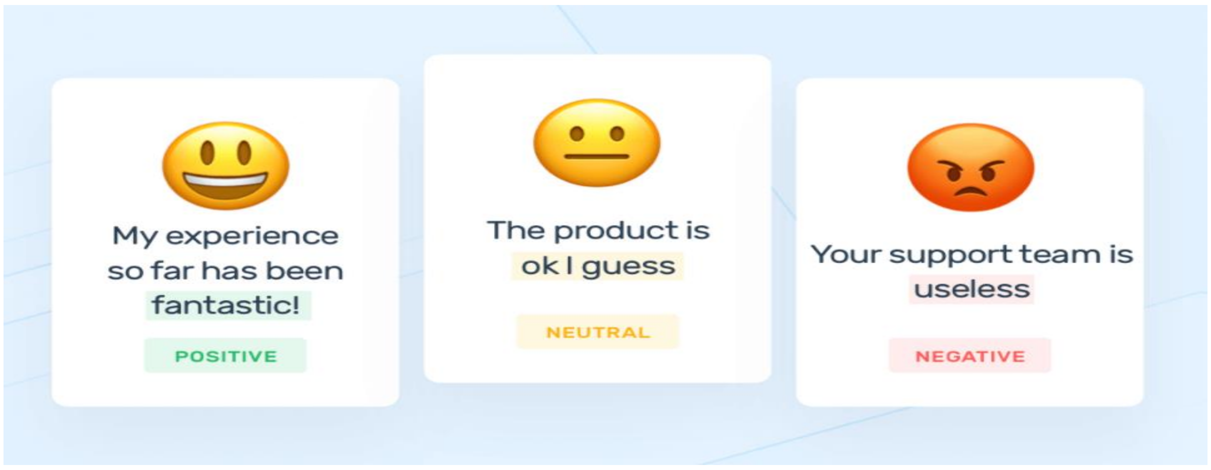
For the Question Answering System, BERT takes two parameters, the input question, and passage as a single packed sequence. The input embeddings are the sum of the token embeddings and the segment embeddings.

1. Token embeddings: A [CLS] token is added to the input word tokens at the beginning of the question and a [SEP] token is inserted at the end of both the question and the paragraph.
2. Segment embeddings: A marker indicating Sentence A or Sentence B is added to each token. This allows the model to distinguish between sentences. In the below example, all tokens marked as A belong to the question, and those marked as B belong to the paragraph.

BERT is undoubtedly a breakthrough in the use of Machine Learning for Natural Language Processing.

The fact that it's approachable and allows fast fine-tuning will likely allow a wide range of practical applications in the future.

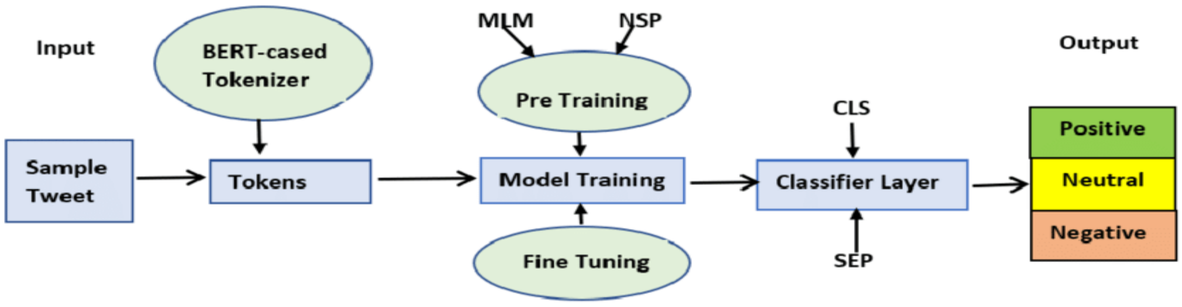
Sentiment Analysis


My experience so far has been fantastic!
POSITIVE
The product is ok I guess
NEUTRAL
Your support team is useless
NEGATIVE

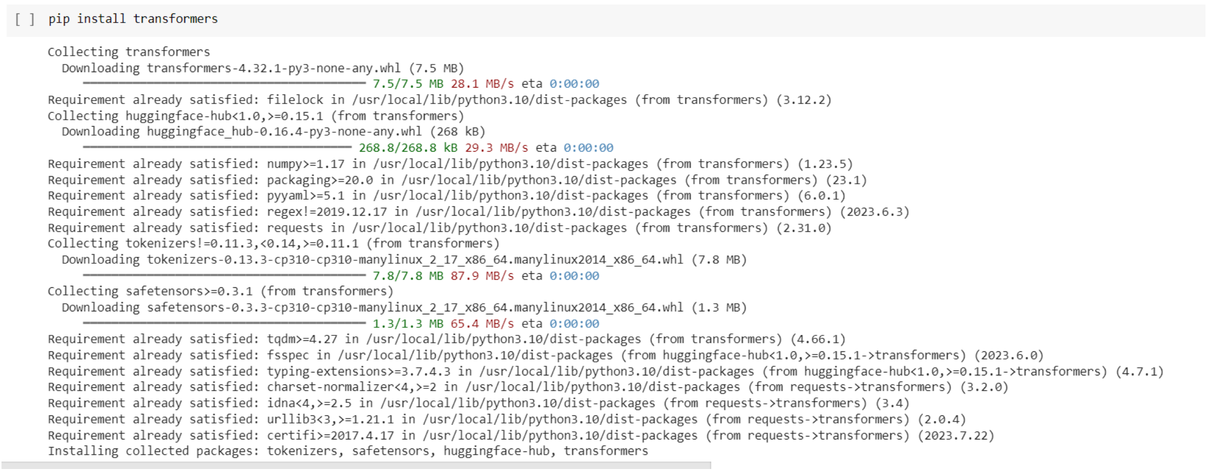
We will do the following operations to train a sentiment analysis model:

- ❑ Install Transformers library
- ❑ Load the BERT Classifier and Tokenizer along with Input modules

- ❑ Download the IMDB Reviews Data and create a processed dataset .
- ❑ Configure the Loaded BERT model and Train for Fine-tuning
- ❑ Make Predictions with the Fine-tuned Model

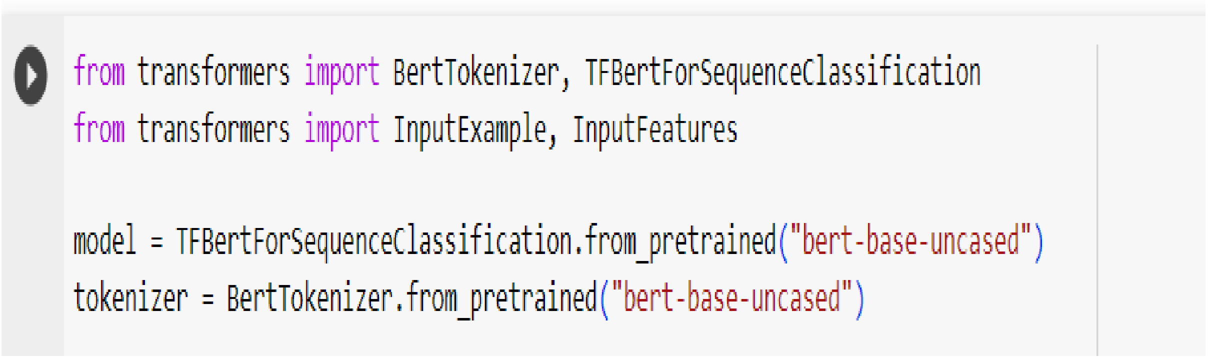


STEP1:install transformers

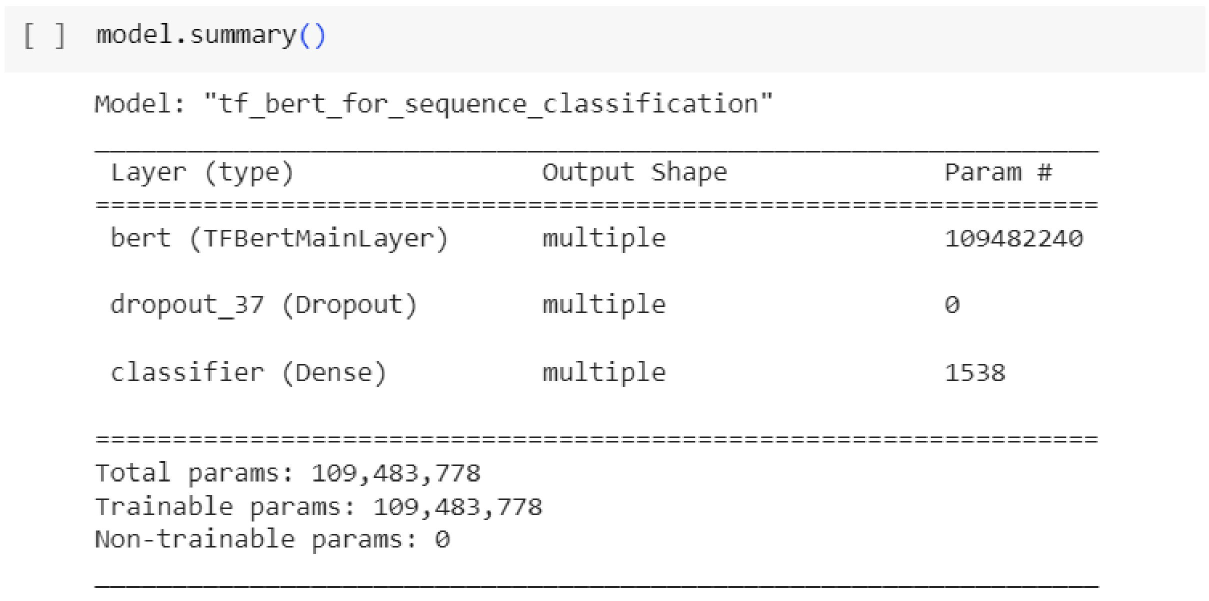


```
[ ]  pip install transformers

     Collecting transformers
       Downloading transformers-4.32.1-py3-none-any.whl (7.5 MB)
          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.5/7.5 MB 28.1 MB/s eta 0:00:00
     Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers) (3.12.2)
     Collecting huggingface-hub<1.0,>=0.15.1 (from transformers)
       Downloading huggingface_hub-0.16.4-py3-none-any.whl (268 kB)
          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 268.8/268.8 kB 29.3 MB/s eta 0:00:00
     Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (1.23.5)
     Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers) (23.1)
     Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (6.0.1)
     Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (2023.6.3)
     Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers) (2.31.0)
     Collecting tokenizers!=0.11.3,<0.14,>=0.11.1 (from transformers)
       Downloading tokenizers-0.13.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.8 MB)
          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 87.9 MB/s eta 0:00:00
     Collecting safetensors>=0.3.1 (from transformers)
       Downloading safetensors-0.3.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 65.4 MB/s eta 0:00:00
     Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-packages (from transformers) (4.66.1)
     Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.15.1->transformers) (2023.6.0)
     Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.15.1->transformers) (4.7.1)
     Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.2.0)
     Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.4)
     Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.4)
     Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2023.7.22)
     Installing collected packages: tokenizers, safetensors, huggingface-hub, transformers
```

STEP 2:load the pretrained BERT model



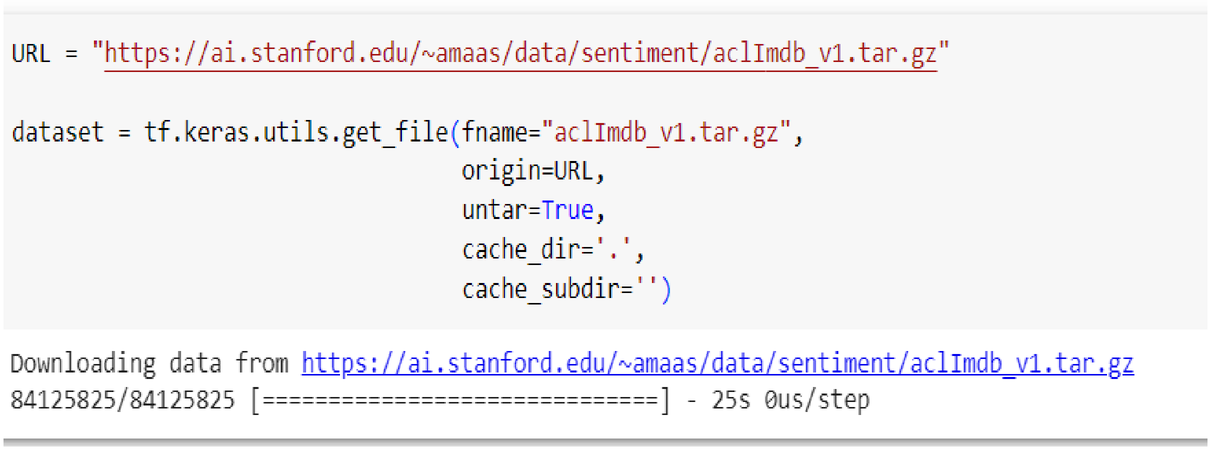
```python
from transformers import BertTokenizer, TFBertForSequenceClassification
from transformers import InputExample, InputFeatures

model = TFBertForSequenceClassification.from_pretrained("bert-base-uncased")
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
```



```
[ ]  model.summary()

     Model: "tf_bert_for_sequence_classification"
     _________________________________________________________________
      Layer (type)                Output Shape              Param #
     =================================================================
      bert (TFBertMainLayer)      multiple                  109482240

      dropout_37 (Dropout)        multiple                  0

      classifier (Dense)          multiple                  1538

     =================================================================
     Total params: 109,483,778
     Trainable params: 109,483,778
     Non-trainable params: 0
     _________________________________________________________________
```

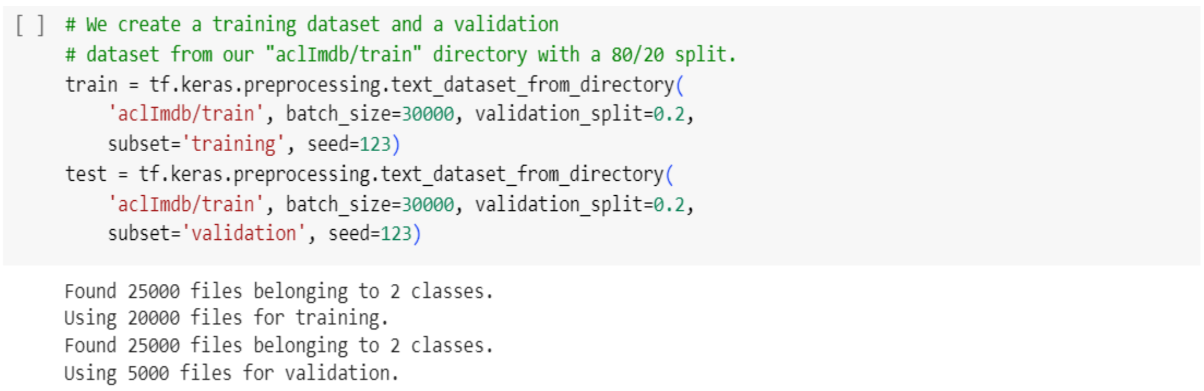
STEP 3: Load the dataset

- ❑ Now that we have our model, let’s create our input sequences from the IMDB reviews dataset:
- ❑ IMDB Dataset IMDB Reviews Dataset is a large movie review dataset collected and prepared by Andrew L. Maas from the popular movie rating service, IMDB.
- ❑ The IMDB Reviews dataset is used for binary sentiment classification, whether a review is positive or negative.
- ❑ It contains 25,000 movie reviews for training and 25,000 for testing. All these 50,000 reviews are labeled data that may be used for supervised deep learning.



```
URL = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"

dataset = tf.keras.utils.get_file(fname="aclImdb_v1.tar.gz",
                                  origin=URL,
                                  untar=True,
                                  cache_dir='.',
                                  cache_subdir='')

Downloading data from https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
84125825/84125825 [==============================] - 25s 0us/step
```

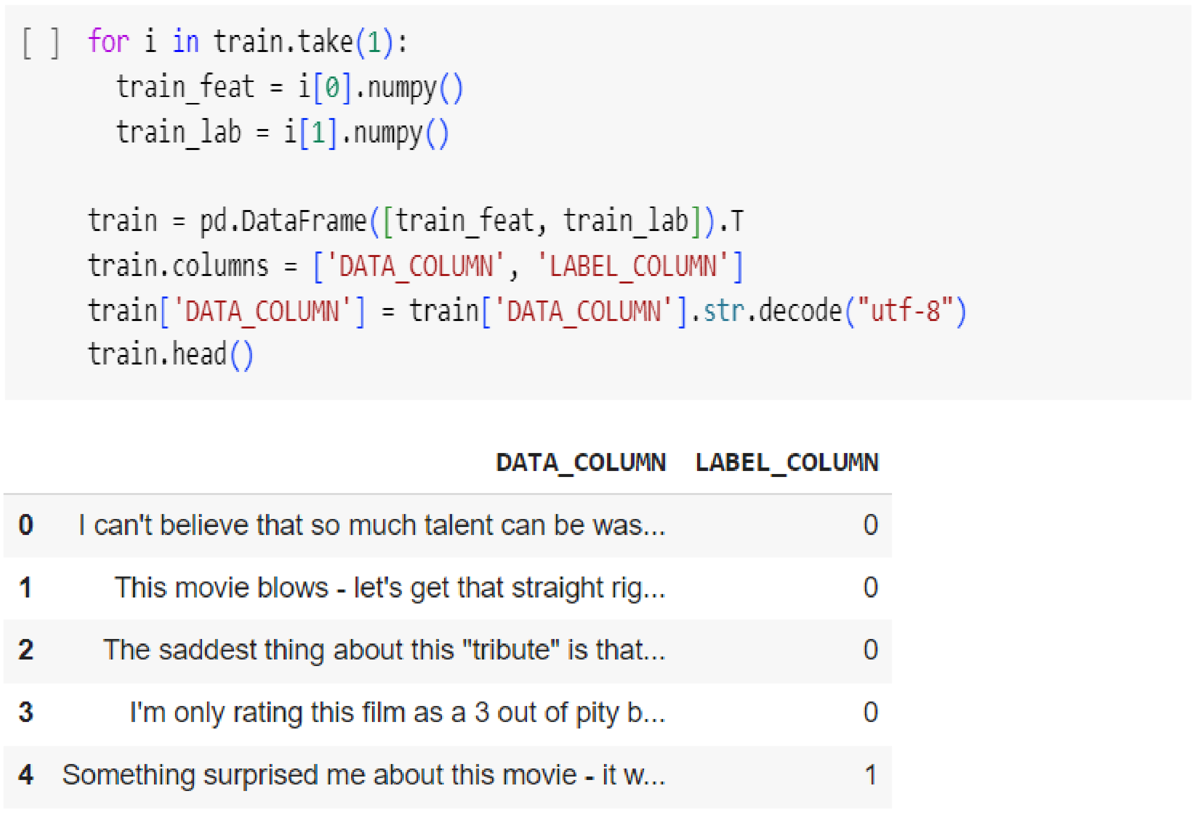
STEP 4:Train and Test Split



```
[ ] # We create a training dataset and a validation
    # dataset from our "aclImdb/train" directory with a 80/20 split.
    train = tf.keras.preprocessing.text_dataset_from_directory(
        'aclImdb/train', batch_size=30000, validation_split=0.2,
        subset='training', seed=123)
    test = tf.keras.preprocessing.text_dataset_from_directory(
        'aclImdb/train', batch_size=30000, validation_split=0.2,
        subset='validation', seed=123)
```

```
Found 25000 files belonging to 2 classes.
Using 20000 files for training.
Found 25000 files belonging to 2 classes.
Using 5000 files for validation.
```

Convert to Pandas to View and Process

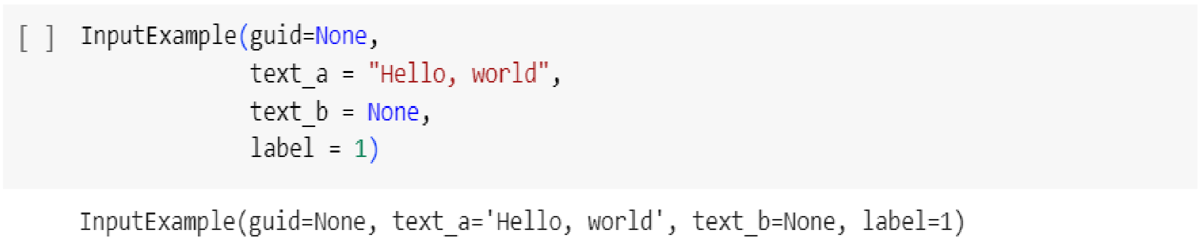


```
[ ] for i in train.take(1):
      train_feat = i[0].numpy()
      train_lab = i[1].numpy()

    train = pd.DataFrame([train_feat, train_lab]).T
    train.columns = ['DATA_COLUMN', 'LABEL_COLUMN']
    train['DATA_COLUMN'] = train['DATA_COLUMN'].str.decode("utf-8")
    train.head()
```

	DATA_COLUMN	LABEL_COLUMN
0	I can't believe that so much talent can be was...	0
1	This movie blows - let's get that straight rig...	0
2	The saddest thing about this "tribute" is that...	0
3	I'm only rating this film as a 3 out of pity b...	0
4	Something surprised me about this movie - it w...	1

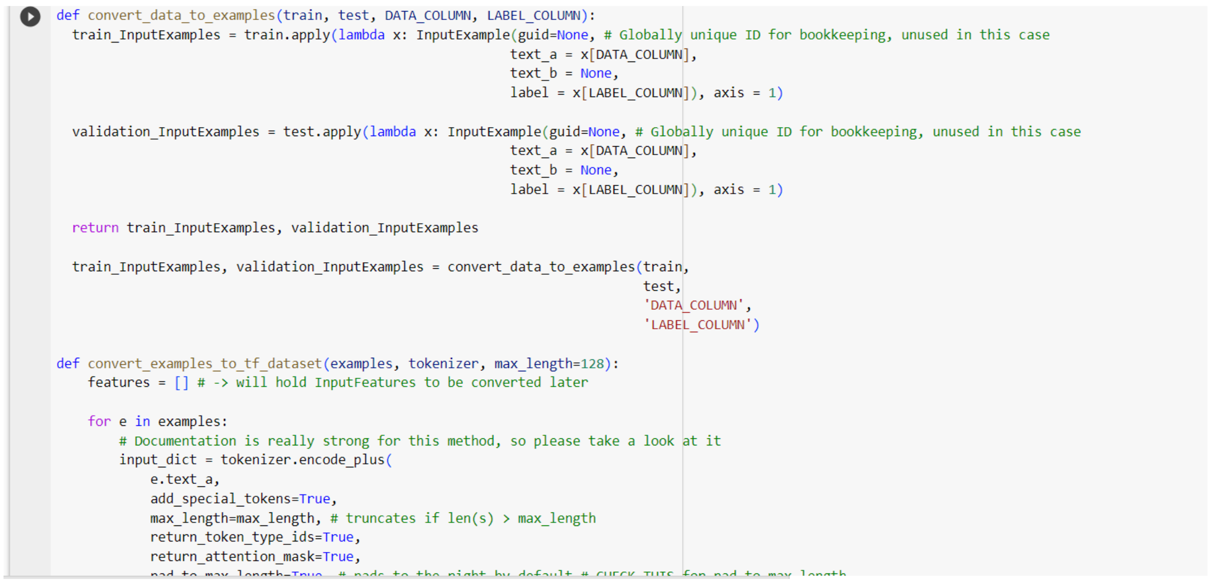
Creating Input Sequences

- We will take advantage of the Input Example function that helps us to create sequences from our dataset. The Input Example function can be called as follows:



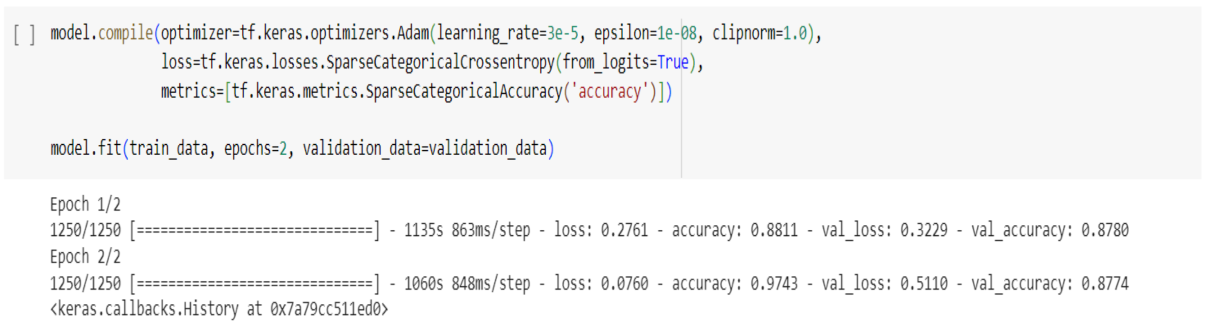
```
[ ] InputExample(guid=None,
                 text_a = "Hello, world",
                 text_b = None,
                 label = 1)
```

```
InputExample(guid=None, text_a='Hello, world', text_b=None, label=1)
```



```
def convert_data_to_examples(train, test, DATA_COLUMN, LABEL_COLUMN):
  train_InputExamples = train.apply(lambda x: InputExample(guid=None, # Globally unique ID for bookkeeping, unused in this case
                                                          text_a = x[DATA_COLUMN],
                                                          text_b = None,
                                                          label = x[LABEL_COLUMN]), axis = 1)

  validation_InputExamples = test.apply(lambda x: InputExample(guid=None, # Globally unique ID for bookkeeping, unused in this case
                                                          text_a = x[DATA_COLUMN],
                                                          text_b = None,
                                                          label = x[LABEL_COLUMN]), axis = 1)

  return train_InputExamples, validation_InputExamples

  train_InputExamples, validation_InputExamples = convert_data_to_examples(train,
                                                                           test,
                                                                           'DATA_COLUMN',
                                                                           'LABEL_COLUMN')

def convert_examples_to_tf_dataset(examples, tokenizer, max_length=128):
    features = [] # -> will hold InputFeatures to be converted later

    for e in examples:
        # Documentation is really strong for this method, so please take a look at it
        input_dict = tokenizer.encode_plus(
            e.text_a,
            add_special_tokens=True,
            max_length=max_length, # truncates if len(s) > max_length
            return_token_type_ids=True,
            return_attention_mask=True,
            pad_to_max_length=True, # pads to the right by default # CHECK THIS for pad to max length
```

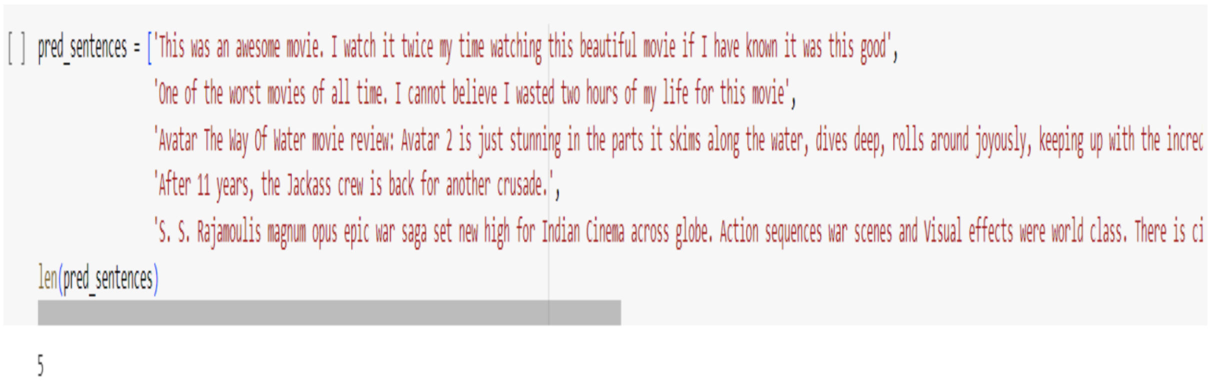
STEP 5:compile



```
[ ] model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=3e-5, epsilon=1e-08, clipnorm=1.0),
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=[tf.keras.metrics.SparseCategoricalAccuracy('accuracy')])

    model.fit(train_data, epochs=2, validation_data=validation_data)

    Epoch 1/2
    1250/1250 [==============================] - 1135s 863ms/step - loss: 0.2761 - accuracy: 0.8811 - val_loss: 0.3229 - val_accuracy: 0.8780
    Epoch 2/2
    1250/1250 [==============================] - 1060s 848ms/step - loss: 0.0760 - accuracy: 0.9743 - val_loss: 0.5110 - val_accuracy: 0.8774
    <keras.callbacks.History at 0x7a79cc511ed0>
```

Configuring the BERT model and Fine-tuning We will use Adam as our optimizer, CategoricalCrossentropy as our loss function, and SparseCategoricalAccuracy as our accuracy metric. Fine-tuning the model for 2 epochs will give us around 95% accuracy, which is great.

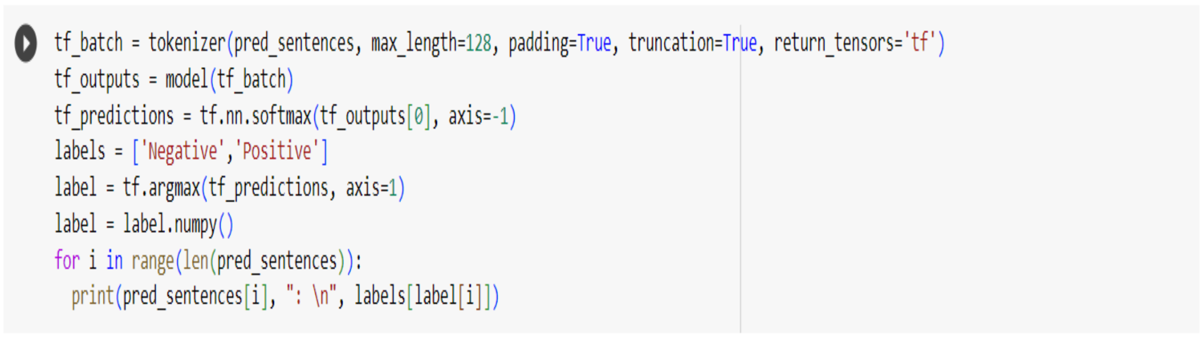
STEP 6: predictions



```
[ ] pred_sentences = ['This was an awesome movie. I watch it twice my time watching this beautiful movie if I have known it was this good',
                      'One of the worst movies of all time. I cannot believe I wasted two hours of my life for this movie',
                      'Avatar The Way Of Water movie review: Avatar 2 is just stunning in the parts it skims along the water, dives deep, rolls around joyously, keeping up with the increc
                      'After 11 years, the Jackass crew is back for another crusade.',
                      'S. S. Rajamoulis magnum opus epic war saga set new high for Indian Cinema across globe. Action sequences war scenes and Visual effects were world class. There is ci
    len(pred_sentences)

    5
```

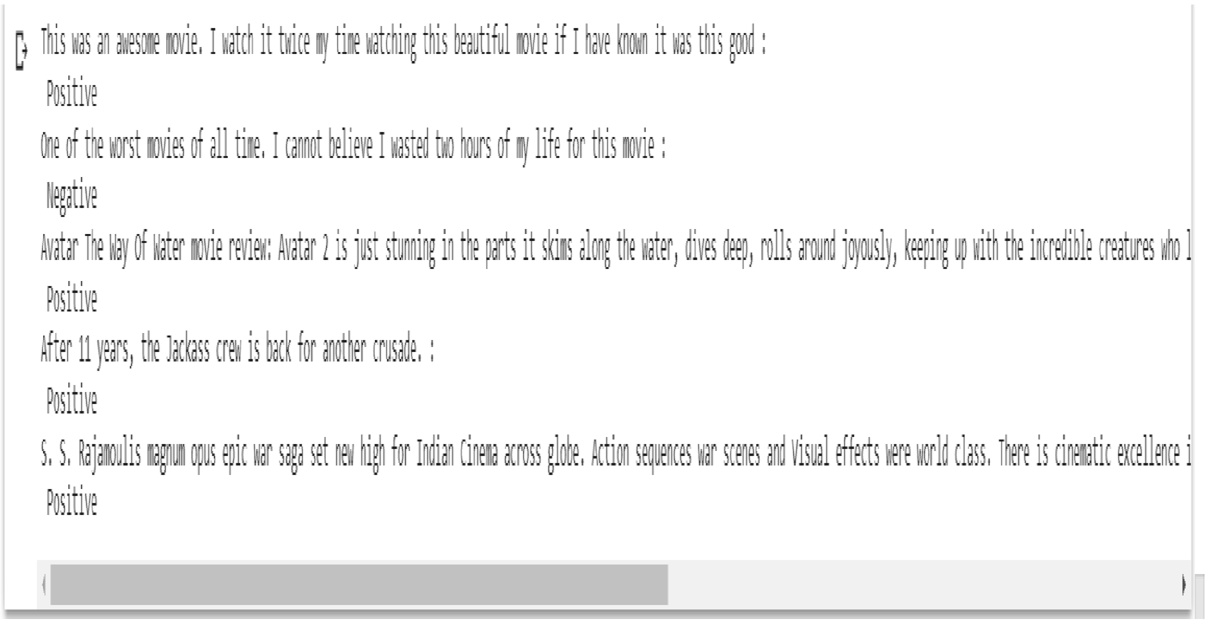
Making Predictions lets created a list of 5 with positive reviews and negative reviews.

STEP 7:results



```
tf_batch = tokenizer(pred_sentences, max_length=128, padding=True, truncation=True, return_tensors='tf')
tf_outputs = model(tf_batch)
tf_predictions = tf.nn.softmax(tf_outputs[0], axis=-1)
labels = ['Negative','Positive']
label = tf.argmax(tf_predictions, axis=1)
label = label.numpy()
for i in range(len(pred_sentences)):
  print(pred_sentences[i], ": \n", labels[label[i]])
```

We will then feed these tokenized sequences to our model and run a final softmax layer to get the predictions. We can then use the argmax function to determine whether our sentiment prediction for the review is positive or negative. Finally, we will print out the results with a simple for loop



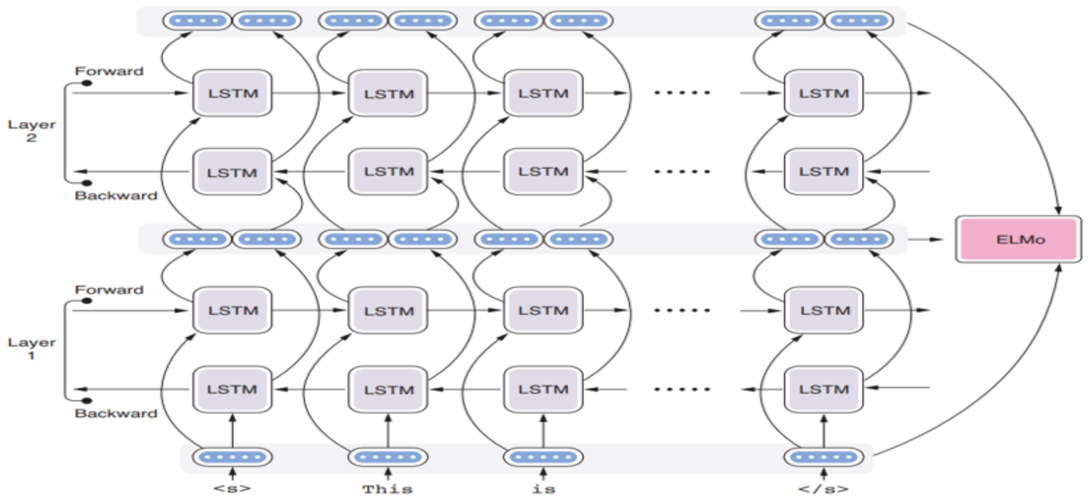
```
This was an awesome movie. I watch it twice my time watching this beautiful movie if I have known it was this good :
 Positive
One of the worst movies of all time. I cannot believe I wasted two hours of my life for this movie :
 Negative
Avatar The Way Of Water movie review: Avatar 2 is just stunning in the parts it skims along the water, dives deep, rolls around joyously, keeping up with the incredible creatures who l
 Positive
After 11 years, the Jackass crew is back for another crusade. :
 Positive
S. S. Rajamoulis magnum opus epic war saga set new high for Indian Cinema across globe. Action sequences war scenes and Visual effects were world class. There is cinematic excellence i
 Positive
```

Other Pretrained Language Models

- BERT is not the only popular pretrained language models (PLMs) commonly used in the NLP community nowadays. There are several other popular PLMs. Most of these models are already implemented and publicly available from the Transformers library, so you can integrate them with your NLP application by changing just a couple of lines of your code.

1.ELMo

- ELMo (Embeddings from Language Models), proposed in early 2018, is one of the earliest PLMs for deriving contextualized embeddings using unlabeled texts.
- Its core idea is simple—train an LSTM-based language model and use its hidden states as additional "features" for downstream NLP tasks. Because the language model is trained to predict the next token given the previous context, the hidden states can encode the information needed to "understand the language."
- ELMo does the same with another, backward LM and combines the embeddings from both directions so that it can also encode the information in both directions. See figure 9.10 for an illustration

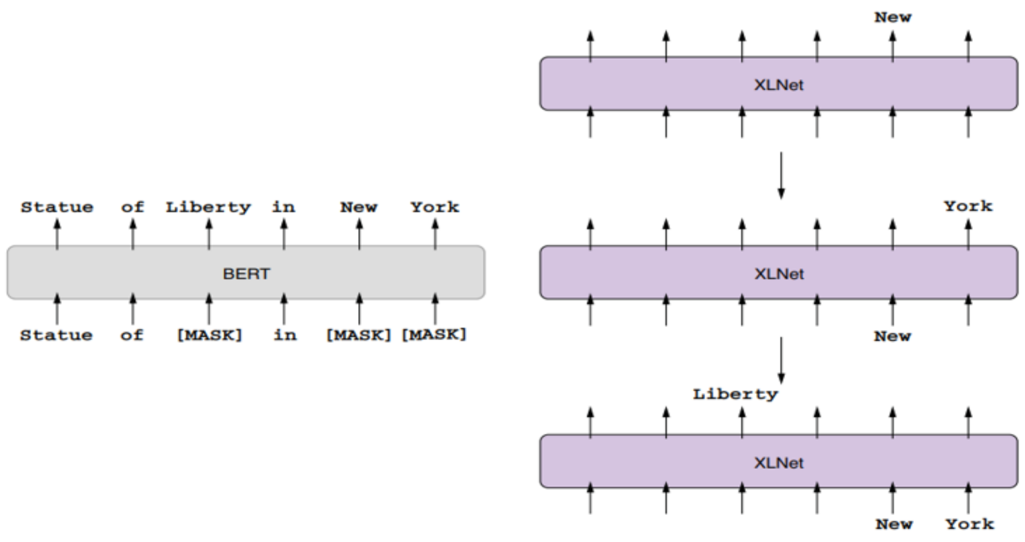


- After pretraining LMs in both directions, downstream NLP tasks can simply use the ELMo embeddings as features.
- Note : ELMo uses multilayer LSTM, so the features are the sum of hidden states taken from different layers, weighted in a task-specific way. The inventors of ELMo showed that adding these features improves the performance

2. XLNet

- XLNet, proposed in 2019, is an important successor of BERT and often referenced as one of the most powerful PLMs as of today.
- XLNet addresses two main issues of how BERT is trained: train-test skew and the independence of masks.

- The first issue has to do with how BERT is pretrained using the masked language model (MLM) objective. During training time, BERT is trained so that it can accurately predict masked tokens, whereas during prediction, it just sees the input sentence, which does not contain any masks. This means that there's a discrepancy of information to which BERT is exposed to between training and testing, and that creates the train-test skew problem.
- The second issue has to do with how BERT makes predictions for masked tokens. If there is more than one [MASK] token in the input, BERT makes predictions for them in parallel.


Statue of Liberty in New York
BERT
Statue of [MASK] in [MASK] [MASK]
New
XLNet
York
XLNet
New
Liberty
XLNet
New York

- For example, you can choose to generate "New" first, which gives a strong clue for the next words, "York" and "Liberty," and so on. Note that prediction is still made based on all the tokens generated previously.
- If the model chose to generate "Washington" first, then the model would proceed to generate "DC" and "Lincoln" and would never mix up these two.
- XLNet is already implemented in the Transformers library, and you can use the model with only a few lines of code change.

3. RoBERTa

- RoBERTa (from "robustly optimized BERT") is another important PLM that is commonly used in research and industry.

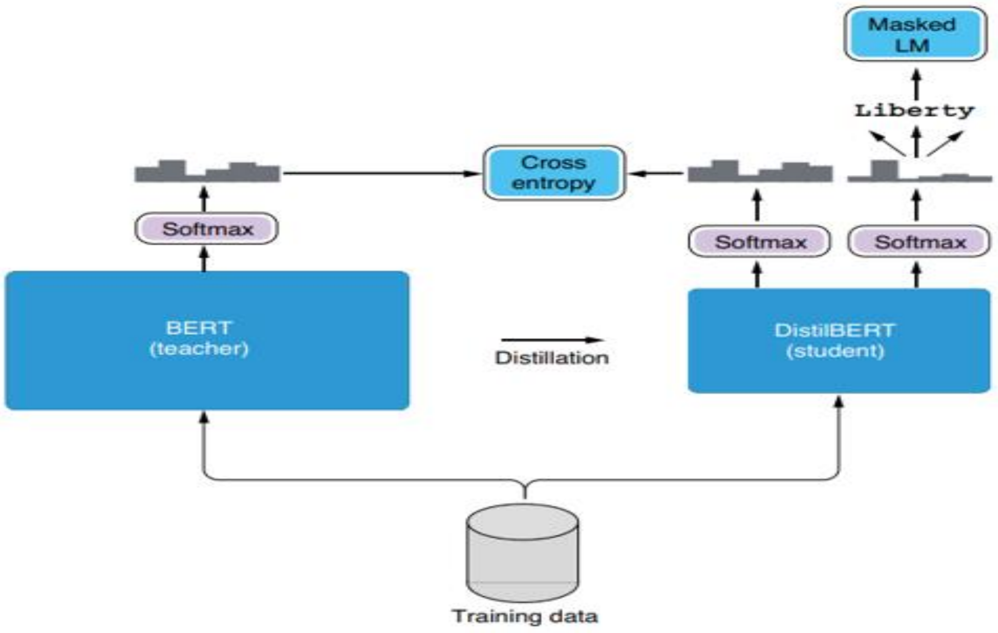
- RoBERTa revisits and modifies many training decisions of BERT, which makes it match or even exceed the performance of postBERT PLMs, including XLNet.
- RoBERTa makes several improvements over BERT, but the most important (and the most straightforward) is the amount of its training data. The developers of RoBERTa collected five English corpora of varying sizes and domains, which total over 160 GB of text (versus 16 GB used for training BERT). Simply by using a lot more data for training, RoBERTa overperforms some of the other powerful PLMs, including XLNet, in downstream tasks after fine-tuning. The second modification has to do with the next-sentence prediction (NSP) objective where BERT is pretrained to classify whether the second sentence is the "true" sentence that follows the first one in a corpus.
- The developers of RoBERTa found that, by removing NSP (and training with the MLM objective only), the performance of downstream tasks stays about the same or slightly improves.
- In addition to these, they also revisited the batch size and the way masking is done for MLM. Combined, the new pretrained language model achieved the state-of-the-art results on downstream tasks such as question answering and reading comprehension.
- Because RoBERTa uses the identical architecture to BERT and both are implemented in Transformers, switching to RoBERTa is extremely easy if your application already uses BERT.

4. DistilBERT

- Although pretrained models such as BERT and RoBERTa are powerful, they are computationally expensive, not just for pretraining but also for tuning and making predictions.
- For example, BERT-base (the regular-sized BERT) and BERT-large (the larger counterpart) have 110 million and 340 million parameters, respectively, and virtually every input has to go through this huge network to get predictions. If you were to finetune and make predictions with a BERT-based model , you'd most certainly need a GPU, which is not always available, depending on your computational environment.

For example, if you'd like to run some real-time text analytics on a mobile phone, BERT wouldn't be a great choice (and it might not even fit in the memory).

- To reduce the computational requirement of modern large neural networks, knowledge distillation (or simply distillation) is often used.
- This is a machine learning technique where, given a large pretrained model (called the teacher model), a smaller model (called the student model) is trained to mimic the behavior of the larger model.

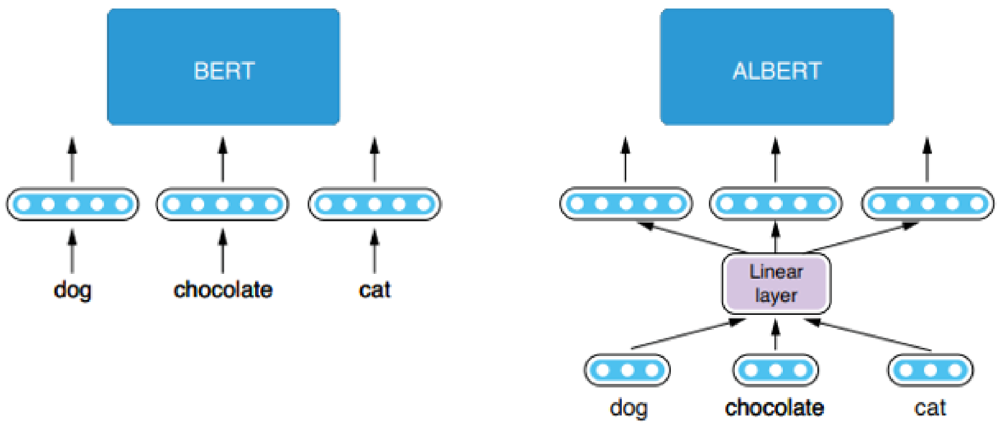


- The student model is trained with the masked language model (MLM) loss (same as BERT), as well as the cross-entropy loss between the teacher and the student. This pushes the student model to produce the probability distribution over predicted tokens that are as similar to the teacher as possible.
- Researchers at Hugging Face developed a distilled version of BERT called DistilBERT, which is 40% smaller and 60% faster while retraining 97% of task performance compared to BERT.

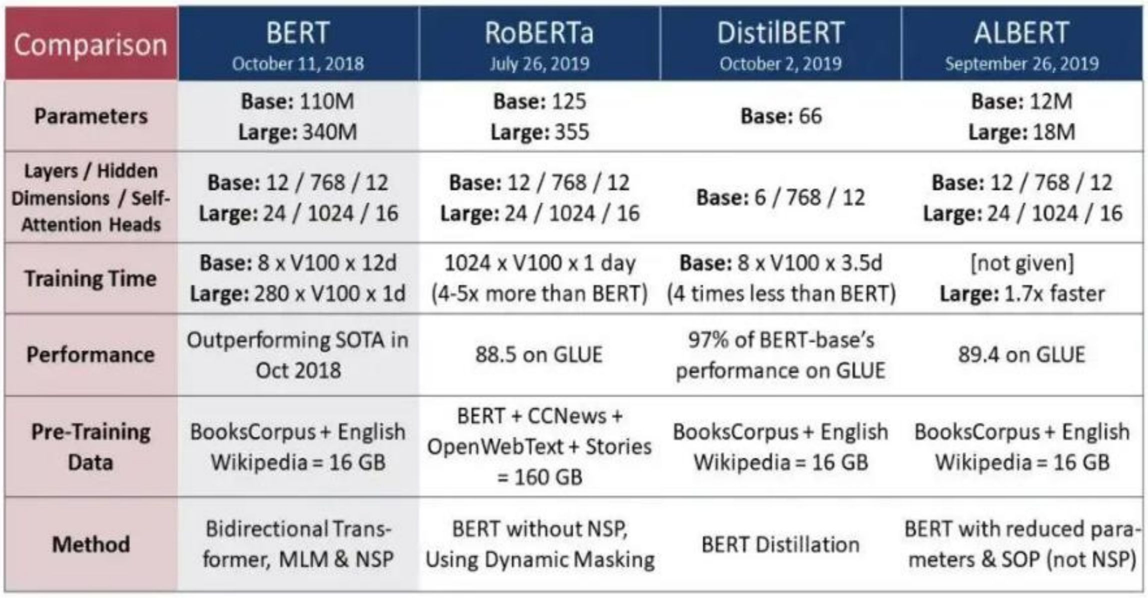
5. ALBERT

- Another pretrained language model that addresses the computational complexity problem of BERT is ALBERT, short for "A Lite BERT."

- Instead of resorting to knowledge distillation, ALBERT makes a few changes to its model and the training procedure. One design change ALBERT makes to its model is how it handles word embeddings.
- In most deep NLP models, word embeddings are represented by and stored in a big lookup table that contains one word embedding vector per word. This way of managing embeddings is usually fine for smaller models such as RNNs and CNNs.
- However, for Transformer-based models such as BERT, the dimensionality (i.e., the length) of input needs to match that of the hidden states, which is usually as big as 768 dimensions. This means that the model needs to maintain a big lookup table of size V times 768, where V is the number of unique vocabulary items. Because in many NLP models V is also large (e.g., 30,000), the resulting lookup table becomes huge and takes up a lot of memory and computation.
- ALBERT addresses this issue by decomposing word embedding lookup into two stages, as shown in figure below



- The first stage is similar to how word embeddings are retrieved from a mapping table, except that the output dimensionality of word embedding vectors is smaller (say, 128 dimensions).
- In the next stage, these shorter vectors are expanded using a linear layer so that they match the desired input dimensionality of the model (say, 768).
- This is similar to how we expanded word embeddings with the Skip-gram model.Thanks to this decomposition, ALBERT needs to store only two smaller lookup tables (V × 128, plus 128 × 768) instead of one big look-up table (V × 768).



Comparison	BERT October 11, 2018	RoBERTa July 26, 2019	DistilBERT October 2, 2019	ALBERT September 26, 2019
Parameters	**Base:** 110M **Large:** 340M	**Base:** 125 **Large:** 355	**Base:** 66	**Base:** 12M **Large:** 18M
Layers / Hidden Dimensions / Self-Attention Heads	**Base:** 12 / 768 / 12 **Large:** 24 / 1024 / 16	**Base:** 12 / 768 / 12 **Large:** 24 / 1024 / 16	**Base:** 6 / 768 / 12	**Base:** 12 / 768 / 12 **Large:** 24 / 1024 / 16
Training Time	**Base:** 8 x V100 x 12d **Large:** 280 x V100 x 1d	1024 x V100 x 1 day (4-5x more than BERT)	**Base:** 8 x V100 x 3.5d (4 times less than BERT)	[not given] **Large:** 1.7x faster
Performance	Outperforming SOTA in Oct 2018	88.5 on GLUE	97% of BERT-base's performance on GLUE	89.4 on GLUE
Pre-Training Data	BooksCorpus + English Wikipedia = 16 GB	BERT + CCNews + OpenWebText + Stories = 160 GB	BooksCorpus + English Wikipedia = 16 GB	BooksCorpus + English Wikipedia = 16 GB
Method	Bidirectional Transformer, MLM & NSP	BERT without NSP, Using Dynamic Masking	BERT Distillation	BERT with reduced parameters & SOP (not NSP)

Case study: Spell Checker with BERT

BERT (Bidirectional Encoder Representations from Transformers) is a powerful transformer-based model that excels in understanding contextual information in natural language text. Here's how BERT can be utilized in a spell checker:

Masked Language Modeling (MLM): BERT can predict missing or masked words in a sentence. This capability is crucial in a spell checker where BERT can predict the correct spelling of misspelled words based on the context of the surrounding words.

Integration with SpellChecker: While BERT itself does not correct spelling mistakes directly, it can provide predictions for masked tokens. These predictions can be used as suggestions for correcting misspelled words identified by another spell checking component like pyspellchecker.

- **pyspellchecker Library:**

The pyspellchecker library provides a straightforward way to identify and correct misspelled words using pre-built dictionaries.

Here's how it complements BERT in the spell checker implementation:

Basic Spell Checking: pyspellchecker uses dictionary-based checks to identify words that do not exist in its dictionary or are not spelled correctly. It can provide initial corrections for obvious spelling mistakes.

Pre-processing: Before passing text to BERT for contextual correction, pyspellchecker can pre-process text to identify and correct straightforward spelling errors. This reduces the complexity and improves the efficiency of the BERT-based correction process.

- **Integration Strategy:**

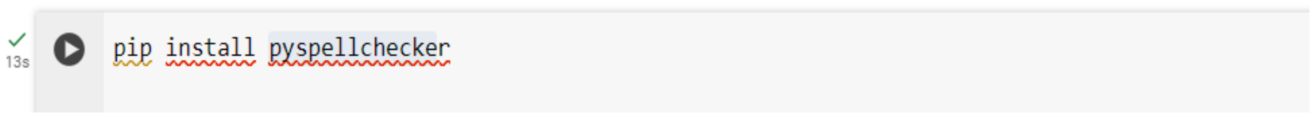
To implement a spell checker using BERT and pyspellchecker, you can follow this integration strategy:

Text Pre-processing: Use pyspellchecker to identify and correct obvious spelling mistakes in the text. This includes corrections for words not found in the dictionary or those with simple typographical errors.

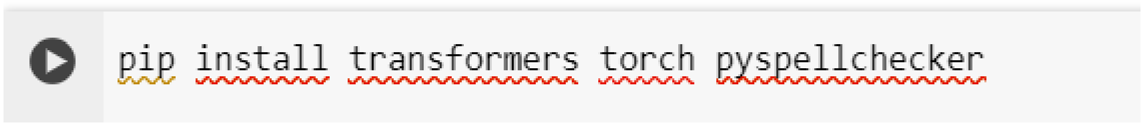
Contextual Correction with BERT: For remaining or complex misspellings identified by pyspellchecker, use BERT to predict the correct spelling based on the surrounding context. BERT's MLM capability allows it to provide accurate predictions for masked tokens.

Evaluation and Refinement: Evaluate the corrected text against expected results or ground truth to measure accuracy. Refine the integration by adjusting parameters such as threshold for BERT predictions or incorporating feedback loops to improve correction suggestions.

Step 1: Install Necessary Libraries Make sure to install transformers, torch, and pyspellchecker:

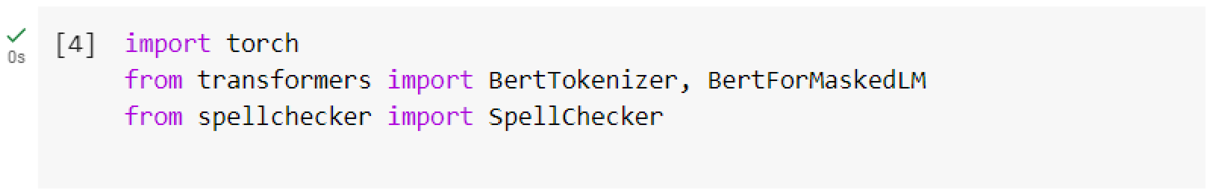


```
pip install pyspellchecker
```



```
pip install transformers torch pyspellchecker
```

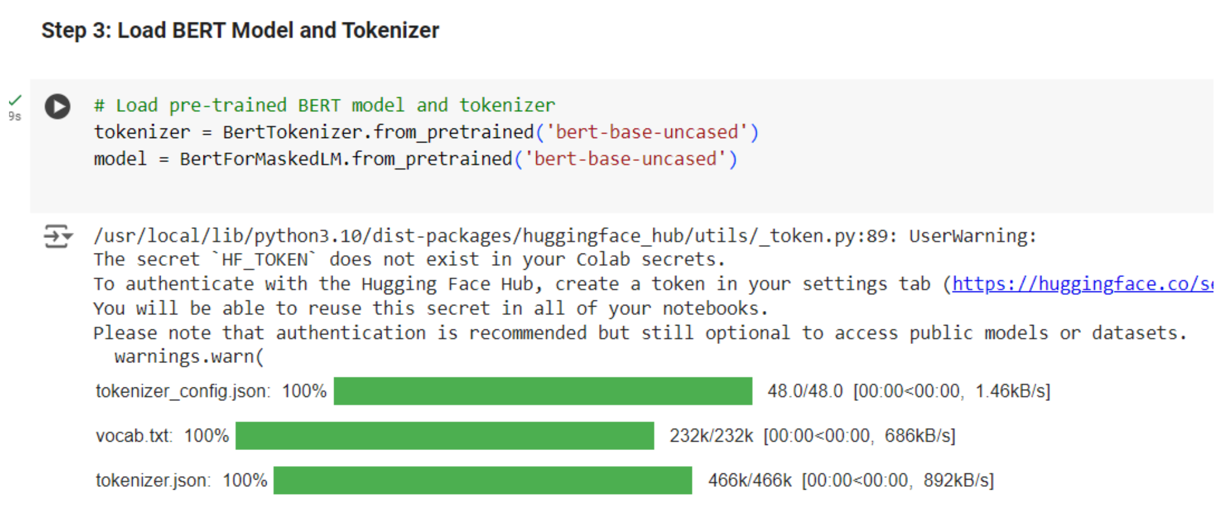
Step 2: Import Libraries



```
[4] import torch
    from transformers import BertTokenizer, BertForMaskedLM
    from spellchecker import SpellChecker
```

Imports: We import necessary libraries:

- torch: PyTorch library for deep learning.
- BertTokenizer and BertForMaskedLM from transformers: These are components from Hugging Face's transformers library for working with BERT models.
- SpellChecker from spellchecker: A library for spell checking



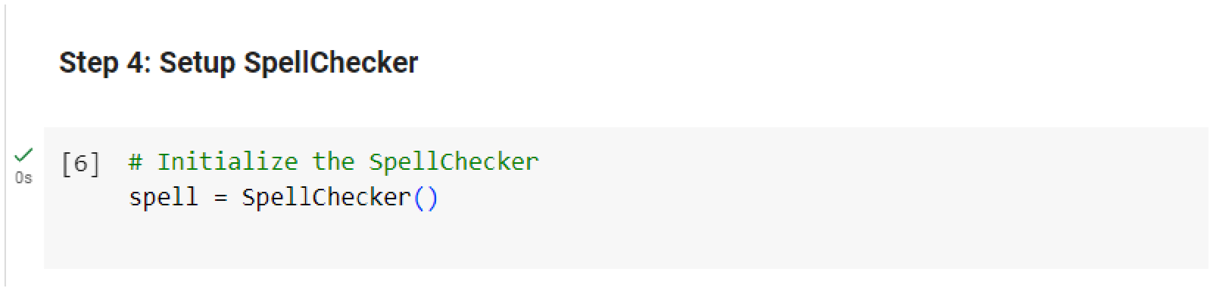
Step 3: Load BERT Model and Tokenizer

```
# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/s
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 1.46kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 686kB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 892kB/s]
```

Load BERT Model:

- tokenizer: Loads the BERT tokenizer for tokenizing text inputs.
- model: Loads the BERT model configured for Masked Language Modeling (MLM), which predicts masked words in a sentence.

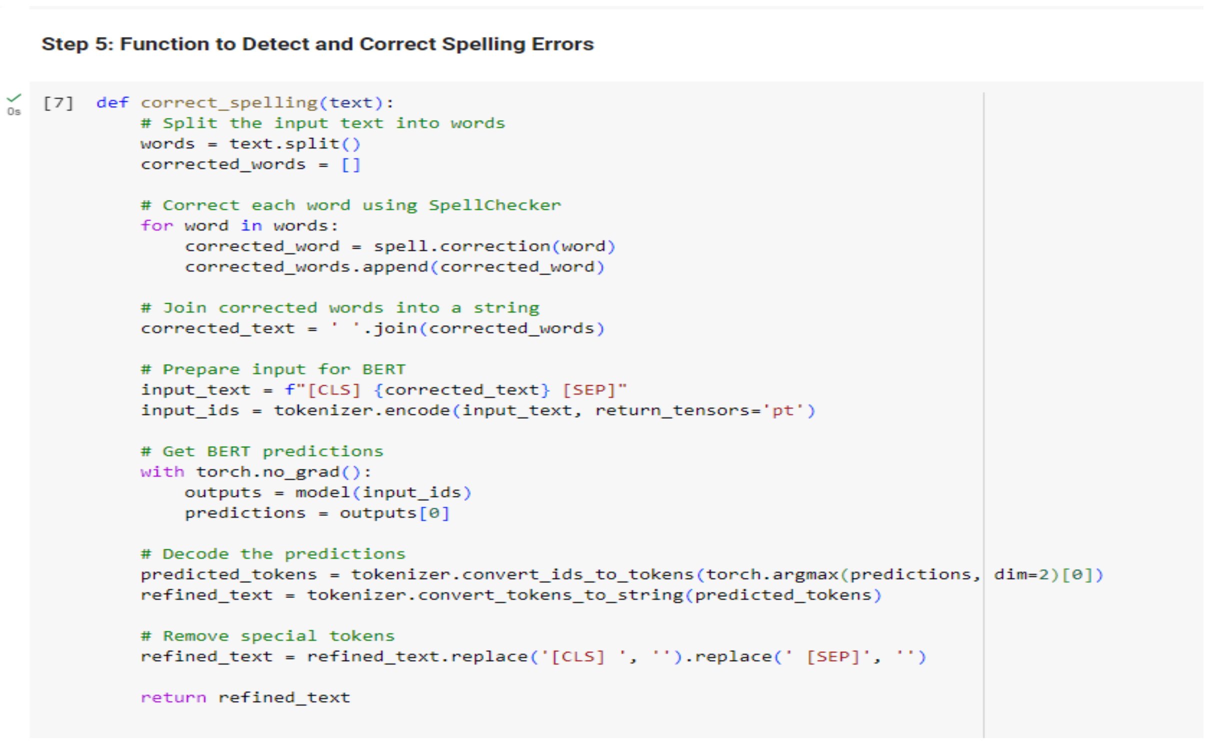


Step 4: Setup SpellChecker

```
[6] # Initialize the SpellChecker
    spell = SpellChecker()
```

pellChecker Initialization:

- spell: Initializes the SpellChecker object for correcting misspelled words.


Step 5: Function to Detect and Correct Spelling Errors

```
[7] def correct_spelling(text):
        # Split the input text into words
        words = text.split()
        corrected_words = []

        # Correct each word using SpellChecker
        for word in words:
            corrected_word = spell.correction(word)
            corrected_words.append(corrected_word)

        # Join corrected words into a string
        corrected_text = ' '.join(corrected_words)

        # Prepare input for BERT
        input_text = f"[CLS] {corrected_text} [SEP]"
        input_ids = tokenizer.encode(input_text, return_tensors='pt')

        # Get BERT predictions
        with torch.no_grad():
            outputs = model(input_ids)
            predictions = outputs[0]

        # Decode the predictions
        predicted_tokens = tokenizer.convert_ids_to_tokens(torch.argmax(predictions, dim=2)[0])
        refined_text = tokenizer.convert_tokens_to_string(predicted_tokens)

        # Remove special tokens
        refined_text = refined_text.replace('[CLS] ', '').replace(' [SEP]', '')

        return refined_text
```

correct_spelling Function:

- **Input Processing**:
    - text: Accepts a string text as input.
- **Spell Checking**:
    - words = text.split(): Splits the input text into individual words.
    - corrected_words = []: Initializes an empty list to store corrected words.
    - for word in words:: Iterates through each word in the input text.
        - spell.correction(word): Uses SpellChecker to correct each word. Appends the corrected word to corrected_words.
- **Preparing Input for BERT**:
    - corrected_text = ' '.join(corrected_words): Joins corrected words into a single string.
    - input_text = f"[CLS] {corrected_text} [SEP]": Prepares the input text for BERT, adding [CLS] (beginning of sequence) and [SEP] (end of sequence) tokens.
    - input_ids = tokenizer.encode(input_text, return_tensors='pt'): Tokenizes input_text using BERT's tokenizer and converts it into PyTorch tensors ('pt').

 BERT Prediction:

- outputs = model(input_ids): Feeds input_ids into the BERT model.
- predictions = outputs[0][0]: Retrieves the predictions from the model's output. predictions has shape (seq_length, vocab_size).

 Processing BERT Predictions:

- predicted_token_ids = torch.argmax(predictions, dim=1): Converts predicted probabilities into token IDs using torch.argmax.
- predicted_tokens = tokenizer.convert_ids_to_tokens(predicted_token_ids.tolist()): Converts token IDs back into tokens using BERT's tokenizer.

☐ **Replacing [MASK] Tokens**:

- refined_tokens = []: Initializes an empty list to store refined tokens.
- for token in predicted_tokens:: Iterates through each predicted token.
  - if token == '[MASK]': Checks if the token is [MASK].
    - if corrected_words:: If there are corrected words left:
      - refined_tokens.append(corrected_words.pop(0)): Appends the next corrected word.
    - else:: Handles the case where there are more [MASK] tokens than corrected words:
      - refined_tokens.append("[MASK]"): Appends [MASK].

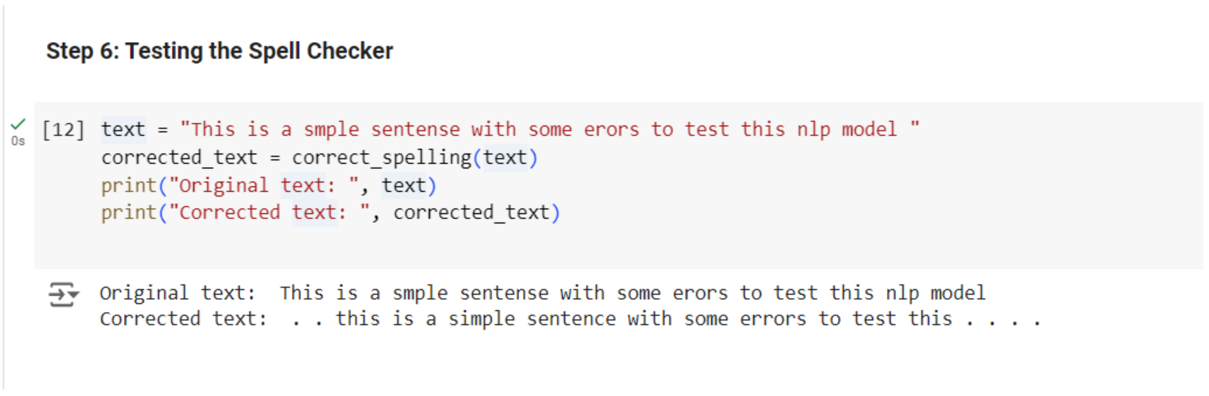
☐ **Converting Tokens to String**:

- refined_text = tokenizer.convert_tokens_to_string(refined_tokens): Converts refined_tokens back into a string.

☐ **Cleaning Up Special Tokens**:

- refined_text = refined_text.replace('[CLS] ', '').replace(' [SEP]', ''): Removes special tokens ([CLS] and [SEP]) from the final refined text.

☐ **Return**: Returns refined_text, the corrected and refined version of the input text.



Step 6: Testing the Spell Checker

```
[12] text = "This is a smple sentense with some erors to test this nlp model "
     corrected_text = correct_spelling(text)
     print("Original text: ", text)
     print("Corrected text: ", corrected_text)
```

```
Original text:  This is a smple sentense with some erors to test this nlp model
Corrected text:  . . this is a simple sentence with some errors to test this . . . .
```

Testing:

text = "This is a smple sentence with som erors.": Defines a test string with intentional spelling errors.

corrected_text = correct_spelling(text): Calls correct_spelling function to correct the text.

print("Original text: ", text): Prints the original text.

print("Corrected text: ", corrected_text): Prints the corrected text returned by the function.

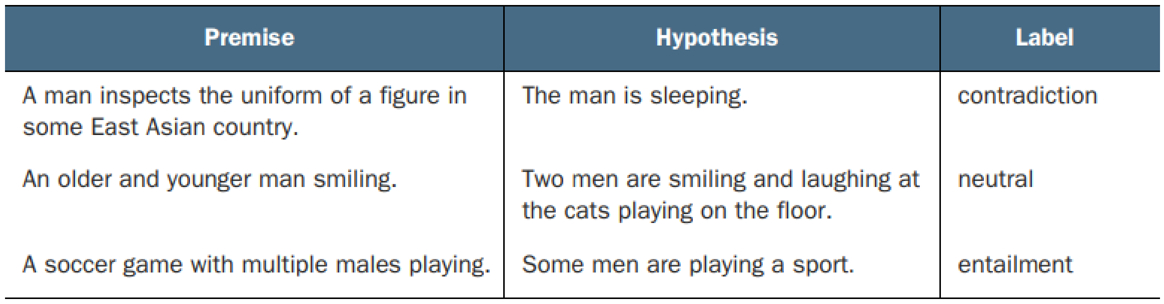
Summary

- This code integrates SpellChecker for initial spelling correction and BERT's masked language model for contextual refinement. It corrects misspelled words, generates a refined version of the input text using BERT's predictions, and handles cases where multiple [MASK] tokens might be present due to spelling corrections. Adjustments can be made based on specific needs and further refinement of the text processing pipeline.
- Spell checking in NLP is a fundamental preprocessing step that ensures the accuracy and quality of text data in various applications.
- As NLP techniques evolve, spell checkers continue to improve through the integration of advanced algorithms and models that enhance their ability to handle complex language issues and provide accurate corrections based on context.

Case study: Natural language inference with BERT

What is natural language inference?

- Natural language inference (or NLI, for short) is the task of determining the logical relationship between a pair of sentences. Specifically, given one sentence (called premise) and another sentence (called hypothesis), you need to determine whether the hypothesis is logically inferred from the premise. This is easier to see in the following examples.



Premise	Hypothesis	Label
A man inspects the uniform of a figure in some East Asian country.	The man is sleeping.	contradiction
An older and younger man smiling.	Two men are smiling and laughing at the cats playing on the floor.	neutral
A soccer game with multiple males playing.	Some men are playing a sport.	entailment

- Natural language inference (NLI) is a task in natural language processing (NLP) where the goal is to determine whether a statement (referred to as the "premise") entails another statement (referred to as the "hypothesis"), contradicts it, or is neutral with respect to it.

- Natural Language Inference (NLI) it involves determining the relationship between a given pair of sentences: a "premise" and a "hypothesis." The objective is to classify the relationship into one of three categories:

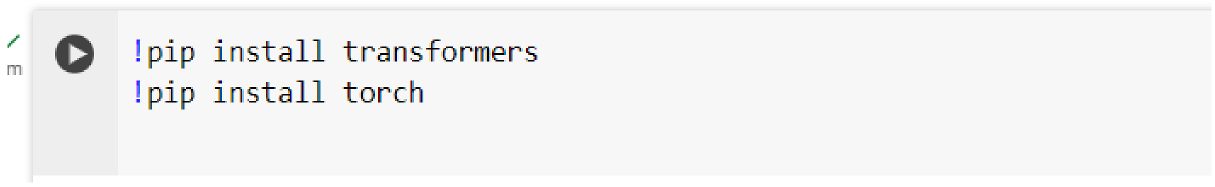
1. **Entailment**: The hypothesis logically follows from the premise. If the premise is true, the hypothesis must also be true.

2. **Neutral**: There is no clear logical relationship between the premise and the hypothesis. The truth of the hypothesis is uncertain given the premise.
3. **Contradiction**: The hypothesis logically contradicts the premise. If the premise is true, the hypothesis must be false.

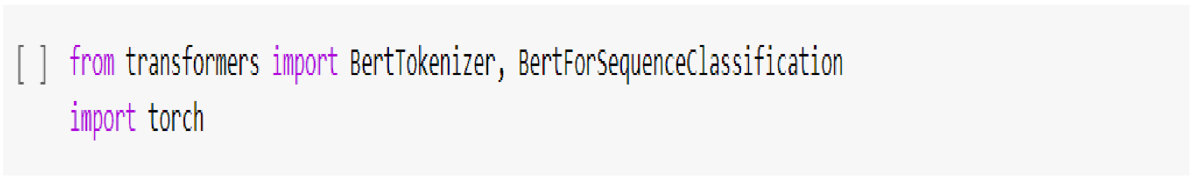
   NLI is typically approached as a supervised learning problem, where the model is trained on a dataset of premise-hypothesis pairs labeled with their corresponding relationship (entailment, contradiction, or neutral). The model learns to make predictions by mapping the input (premise and hypothesis) to an output (entailment, contradiction, or neutral) using various techniques such as neural networks or tree-based models.

   NLI has many applications in NLP, including question answering, sentiment analysis, and dialogue systems. It is also a fundamental building block for more complex tasks such as natural language understanding and generation.

Step 1: Import Libraries



```
!pip install transformers
!pip install torch
```



```
from transformers import BertTokenizer, BertForSequenceClassification
import torch
```

Step 2: Load the Tokenizer and Model

BertTokenizer.from_pretrained: Loads a pre-trained tokenizer for tokenizing input text into the format required by the BERT model.

BertForSequenceClassification.from_pretrained: Loads a pre-trained BERT model specifically fine-tuned for sequence classification tasks like NLI. The model will output logits for each class (entailment, neutral, contradiction).

model.eval(): Puts the model in evaluation mode, which disables dropout and other training-specific features.

```
[ ] # Load the tokenizer and the NLI model
    tokenizer = BertTokenizer.from_pretrained('textattack/bert-base-uncased-snli')
    model = BertForSequenceClassification.from_pretrained('textattack/bert-base-uncased-snli')

    # Ensure the model is in evaluation mode
    model.eval()
```

Step 3: Prepare the Input Data

prepare_data function: Takes a premise and a hypothesis as input.

tokenizer.encode_plus: Tokenizes the input text (premise and hypothesis) and returns it in the format required by BERT (including special tokens like [CLS] and [SEP]).

return_tensors='pt': Ensures the output is a PyTorch tensor.

max_length=512: Sets the maximum length for the input sequence (BERT can handle up to 512 tokens).

truncation=True: Truncates sequences longer than the maximum length.

padding='max_length': Pads sequences shorter than the maximum length with zeros.

```
[ ] def prepare_data(premise, hypothesis):
        # Tokenize the inputs
        inputs = tokenizer.encode_plus(premise, hypothesis, return_tensors='pt', max_length=512, truncation=True, padding='max_length')
        return inputs
```

Step 4: Define the Prediction Function

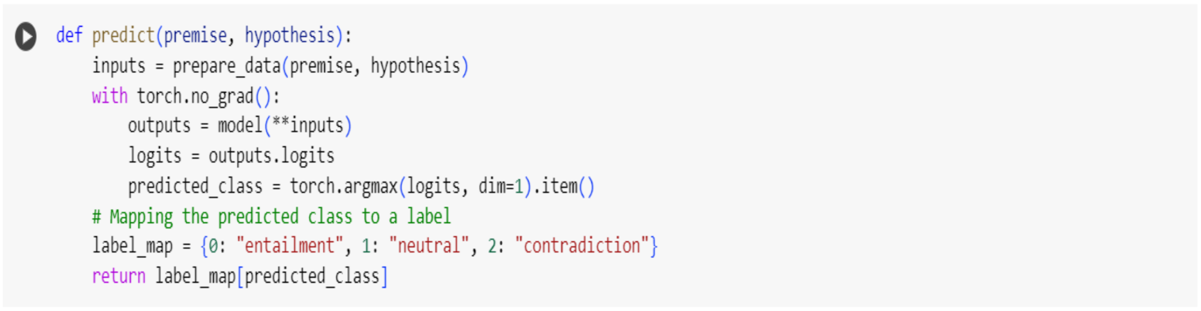
predict function: Takes a premise and a hypothesis, prepares the data, and returns the predicted NLI label.

prepare_data(premise, hypothesis): Prepares the input data using the prepare_data function.

with torch.no_grad(): Disables gradient calculation, reducing memory usage and speeding up computations.

model(**inputs): Feeds the prepared input data into the model and gets the output logits.

torch.argmax(logits, dim=1).item(): Gets the index of the highest logit (i.e., the predicted class). label_map: Maps the predicted class index to the corresponding label (entailment, neutral, contradiction).



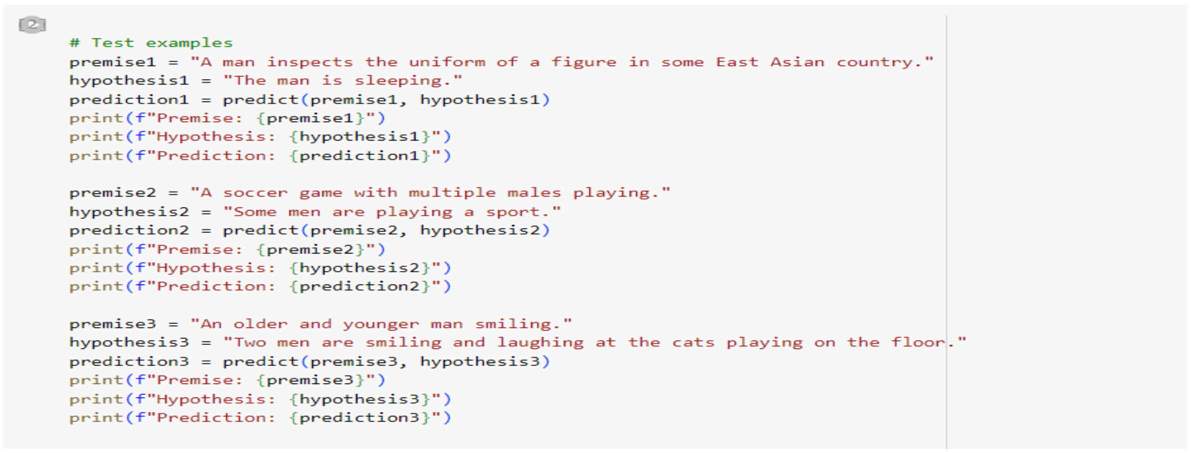
```python
def predict(premise, hypothesis):
    inputs = prepare_data(premise, hypothesis)
    with torch.no_grad():
        outputs = model(**inputs)
        logits = outputs.logits
        predicted_class = torch.argmax(logits, dim=1).item()
    # Mapping the predicted class to a label
    label_map = {0: "entailment", 1: "neutral", 2: "contradiction"}
    return label_map[predicted_class]
```

Step 5: Test the Prediction Function

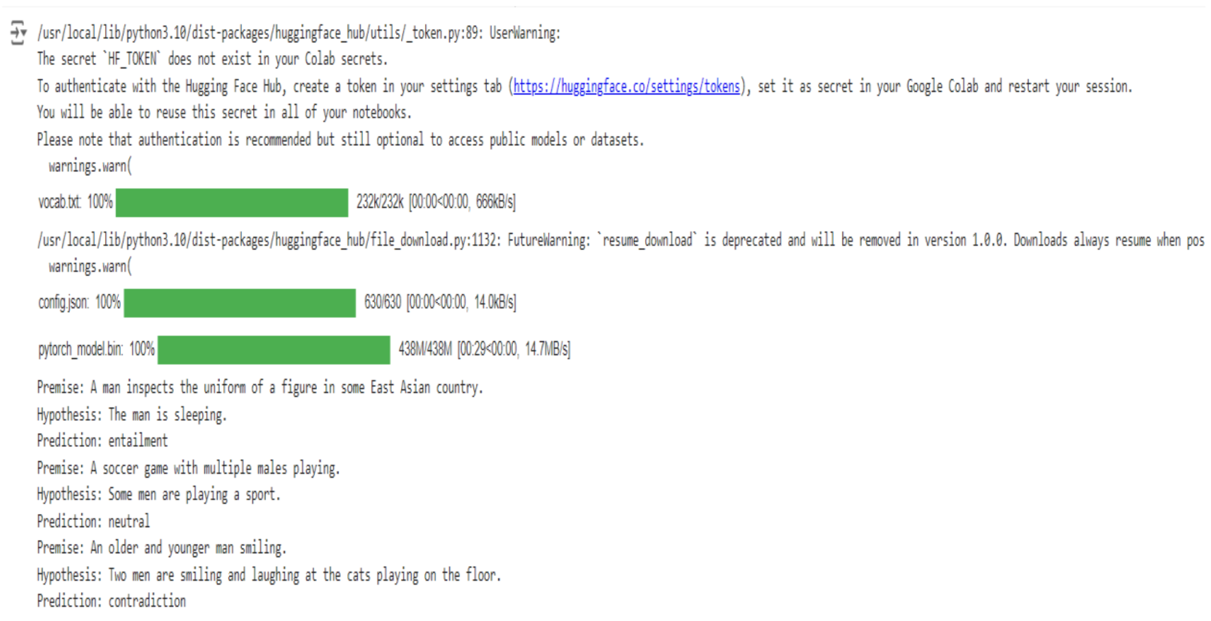
Testing: Three examples are tested with different premises and hypotheses.

predict function: Called for each pair of premise and hypothesis to get the prediction.

print statements: Print the premise, hypothesis, and the predicted NLI label.



```python
# Test examples
premise1 = "A man inspects the uniform of a figure in some East Asian country."
hypothesis1 = "The man is sleeping."
prediction1 = predict(premise1, hypothesis1)
print(f"Premise: {premise1}")
print(f"Hypothesis: {hypothesis1}")
print(f"Prediction: {prediction1}")

premise2 = "A soccer game with multiple males playing."
hypothesis2 = "Some men are playing a sport."
prediction2 = predict(premise2, hypothesis2)
print(f"Premise: {premise2}")
print(f"Hypothesis: {hypothesis2}")
print(f"Prediction: {prediction2}")

premise3 = "An older and younger man smiling."
hypothesis3 = "Two men are smiling and laughing at the cats playing on the floor."
prediction3 = predict(premise3, hypothesis3)
print(f"Premise: {premise3}")
print(f"Hypothesis: {hypothesis3}")
print(f"Prediction: {prediction3}")
```



```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
vocab.txt: 100%  232k/232k [00:00<00:00, 666kB/s]
/usr/local/lib/python3.10/dist-packages/huggingface_hub/file_download.py:1132: FutureWarning: `resume_download` is deprecated and will be removed in version 1.0.0. Downloads always resume when pos
  warnings.warn(
config.json: 100%  630/630 [00:00<00:00, 14.0kB/s]
pytorch_model.bin: 100%  438M/438M [00:29<00:00, 14.7MB/s]
Premise: A man inspects the uniform of a figure in some East Asian country.
Hypothesis: The man is sleeping.
Prediction: entailment
Premise: A soccer game with multiple males playing.
Hypothesis: Some men are playing a sport.
Prediction: neutral
Premise: An older and younger man smiling.
Hypothesis: Two men are smiling and laughing at the cats playing on the floor.
Prediction: contradiction
```