**Implementation of transformers using Huggingface**

**Index**

1. **Abstract**
2. **Introduction**
   1. **What is Hugging Face?**
   2. **The Hugging Face ecosystem**
   3. **Basic tasks supported by Hugging Face**
3. **Methodology**
   1. **Hugging Face transformers and how to use them**
   2. **Encoder models**
   3. **Decoder models**
   4. **Transformers and pipelines**
   5. **Using pre-defined pipelines**
   6. **Create your own pipeline**
   7. **Some pre-trained transformer models** 
      1. **mBART model**
      2. **T5**
      3. **MarianMT model**
   8. **Create your own machine learning translator & fine tune them**
   9. **Standard Transformer 2017**
      1. **Create Word Embeddings**
      2. **Positional Encoding**
      3. **Self-Attention**
      4. **Encoder**
      5. **Decoder**
4. **Conclusion**
5. **References**
6. **Abstract**

When you are working on a Machine learning problem, **adapting an existing solution and re purposing it can help you get to a solution much faster.** **Using existing models, not just aid machine learning engineers or data scientists but also helps companies to save computational costs as it requires less training.** There are many companies that provide open source libraries containing pre-trained models and Hugging Face is one of them.

Hugging Face first launched its chat platform back **in 2017.** **To normalize NLP and make models accessible to all, they created an NLP library that provides various resources like datasets, transformers, and tokenizers**, etc. On releasing NLP libraries called Transformers and a wide variety of tools, Hugging Face instantly became very popular among big tech companies.

Hugging Face is focused on Natural Language **Processing (NLP) tasks and the idea is not to just recognize words but to understand the meaning and context of those words**. Computers do not process the information in the same way as humans and which is why need **a pipeline – a flow of steps to process the texts.**

Many companies are now adding NLP technologies into their systems for enhanced interaction experience and having communication close to human experience as much as possible is becoming more important than ever. This is where Hugging Face comes into the picture. In the upcoming sections, will be covering Hugging Face and its transformers in detail.

1. **Introduction**

A transformer is a deep learning model. It is distinguished by its adoption of self-attention, differentially weighting the significance of each part of the input (which includes the recursive output) data. It is used primarily in the fields of natural language processing (NLP) and computer vision (CV).

Like recurrent neural networks (RNNs), transformers are designed to process sequential input data, such as natural language, with applications towards tasks such as translation and text summarization. However, unlike RNNs, transformers process the entire input all at once. The attention mechanism provides context for any position in the input sequence. For example, if the input data is a natural language sentence, the transformer does not have to process one word at a time. This allows for more parallelization than RNNs and therefore reduces training times.

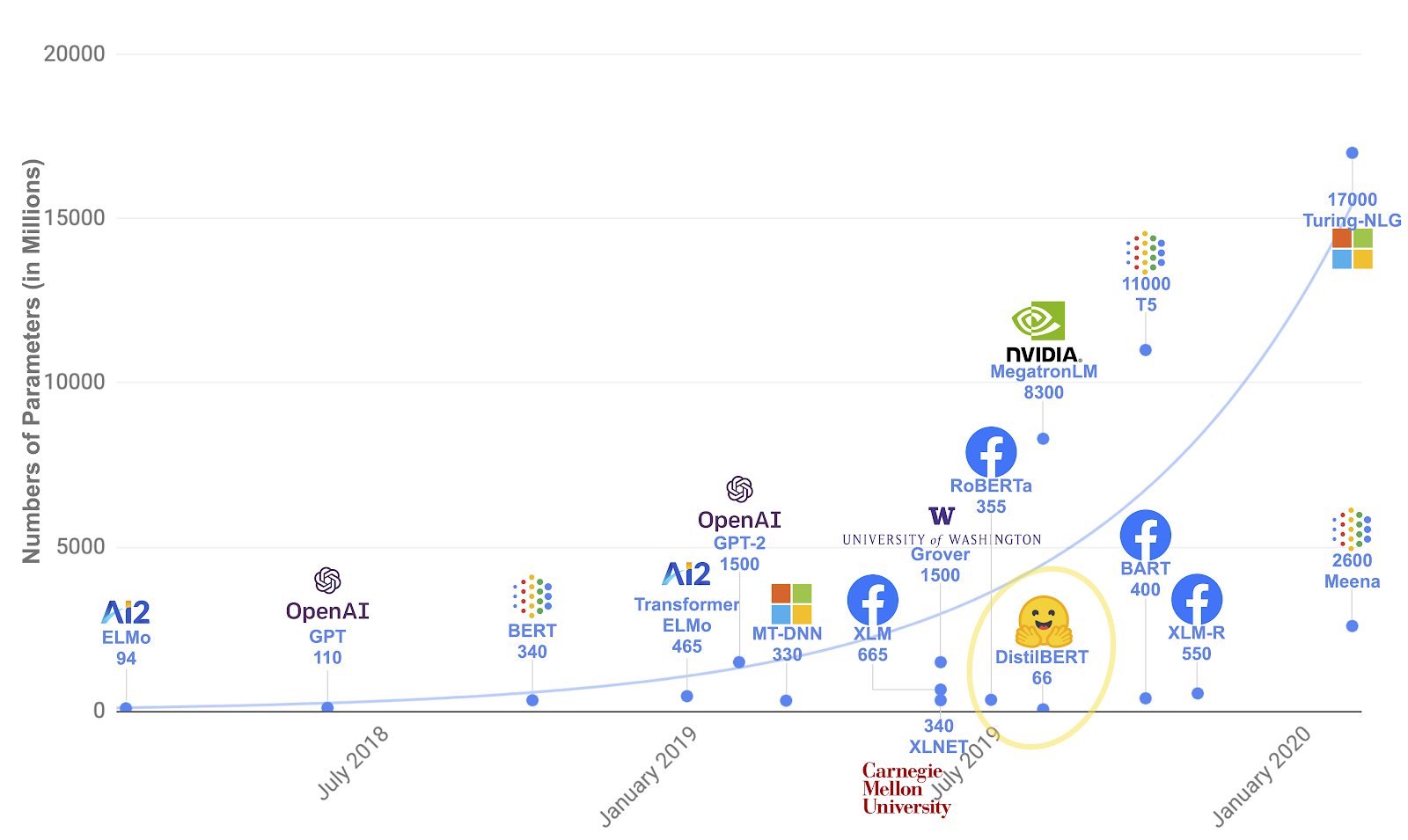
Transformers were introduced in 2017 by a team at Google Brain and are increasingly becoming the model of choice for NLP problems, replacing RNN models such as long short-term memory (LSTM). Compared to RNN models, transformers are more amenable to parallelization, allowing training on larger datasets. This led to the development of pre-trained systems such as BERT (Bidirectional Encoder Representations from Transformers) and the original GPT (generative pre-trained transformer), which were trained with large language datasets, such as the Wikipedia Corpus and Common Crawl, and can be fine-tuned for specific tasks.

**2.1 What is Hugging Face?**

In in just a few short years, **with more than 1,219 contributors, 25,800 users, 61,000 stars, and 14,700 forks on GitHub**, AI community Hugging Face’s transformers has established itself as the go-to provider for all things NLP.

Hugging Face is a start-up, AI community, and the self-described “**home of machine learning**” that was initially founded as a messaging app.

Now focusing exclusively on transformers, the company provides open-source NLP technologies and thousands of **pre-trained models to perform tasks on different modalities such as text, vision, and audio. It also provides courses and datasets and has a large community following.** In **2019, it raised US$15 million in venture funding to build a definitive NLP library** before raising a further **US$40 million in a 2021 Series B funding round.**



Some of the benefits of using the Hugging Face transformers library include:

* **Easy-to-use, state-of-the-art models**
* **High-performance natural language understanding and generation**
* **High-performance computer vision and audio tasks**
* **Much lower computing costs and smaller carbon footprint due to model sharing**
* **Choose the proper framework for every part of a model’s lifetime**
* **Models are easily customizable and adaptable to different use cases**

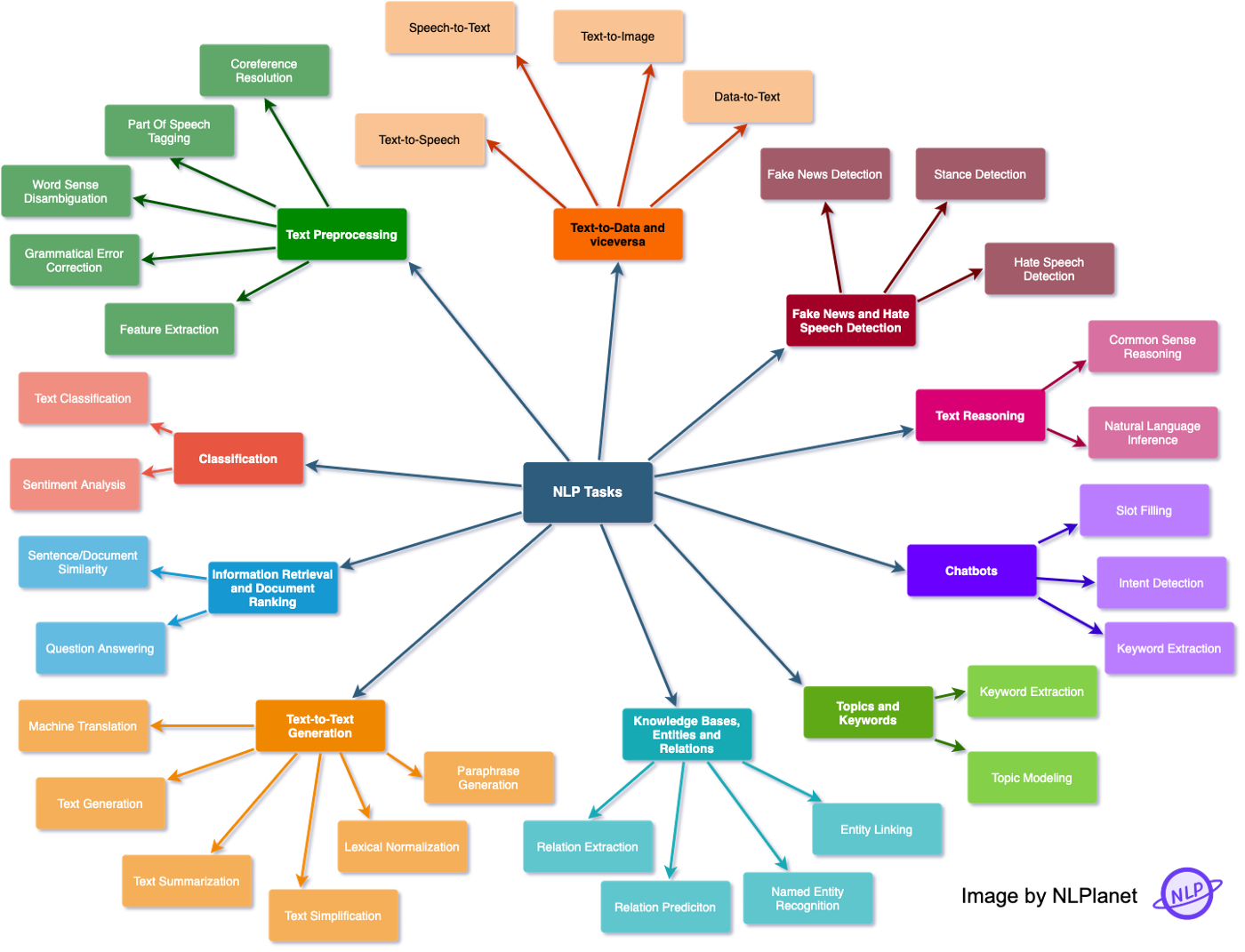
**2.2 The Hugging Face ecosystem**

Hugging Face has been **built around the concept of attention-based transformer models**. At the core of its ecosystem, then, is its transformers library which is supported by its datasets and tokenizers libraries.

Since transformer models don’t understand text sequences in their native form of a string of characters, **they must be converted into vectors, matrices, and tensors, and thus a tokenizer is a core component of the Hugging Face transformer ecosystem and its pipelines.**

Hugging Face also **comes with the accelerate library.** This integrates with existing Hugging Face training flows and generic PyTorch training scripts in order to easily **empower distributed training with various hardware acceleration devices like GPUs and TPUs. This means that the same training script can be used on a dedicated training run with multiple GPUs or on a laptop CPU.**

Supporting all of Hugging Face’s libraries is a dedicated community, the Hugging Face Hub, which creates and shares community resources. The Hub adds value to projects with tools for versioning and an API for hosted inference.

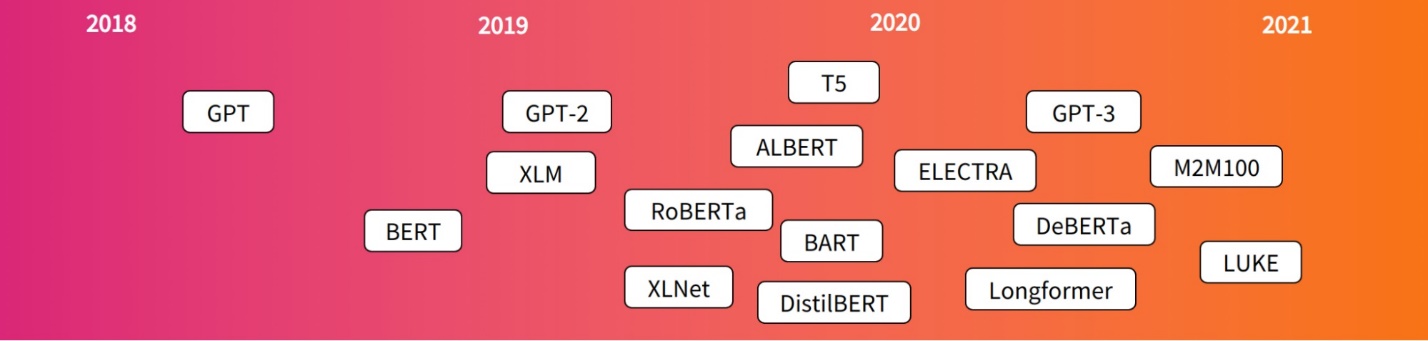


* 1. **Basic tasks supported by Hugging Face**

Before learn how a hugging face model can be used to implement NLP solutions, need to know what the basic NLP tasks that Hugging Face supports are and why do care about them. Hugging Face models provide many different configurations and great support for a variety of use cases, but here are some of the basic tasks that it is widely used for:

1. **Sequence classification: -** Given a number of classes, **the task is to predict the category of a sequence of inputs.** It is a predictive modeling problem and has a broad range of applications. Some real-world use cases are **(Understanding the sentiment behind a review, detecting spam emails, correcting grammatical mistakes, etc.)**
2. **Question & answer: -** **Providing an answer for a given contextual question.** The idea is to build a system **that can automatically answer questions posed by humans**. The questions can be **open or close-ended and the system should be designed to be compatible with both.** The answers can be constructed either **by querying a structured database or searching through an unstructured collection of documents.**
3. **Named entity recognition: -** Named entity recognition is the **task of identifying a token as a person, place, or organization.** It is being used in many fields in NLP and helps solve many real-world problems. In this technique, **can scan articles and extract fundamental entities and categorize them into defined classes.**
4. **Summarization: -** Do you remember writing a summary report in school or college? Will **this task is the same, given a document, with the help of NLP, it can be converted into a concise text.** **It is a process of creating a short, coherent, and fluent version of a longer text.** There are two approaches that can be used for text **summarization – Extractive and Abstractive.** In the **extractive approach, extract the important sentences and phrases whereas**, during **the abstractive approach, are required to interpret the context and reproduce the text keeping core information intact.**
5. **Translation: -** It is the **task of translating a text from one language to another.** Replacing atomic words is not enough as want to create a system that is able to translate the text like a human translator. **Need a corpus that takes into account phonetic typology, translations of idioms, etc. to conduct complicated translations.**
6. **Language modelling: -** Language modeling involves **generating text to make sense of a sequence of tokens or predicting some phrases that can be used to complete a text.** These tasks can be categorized as **– Masked Language Modelling and Casual Language modeling.**

There is more to NLP tasks other than just working with written text, it also **covers solutions related to Speech Recognition, Computer Vision, Generating Transcripts**, etc. NLP tasks are difficult to handle with Machine Learning and a lot of research has been done to improve the accuracy of these models.



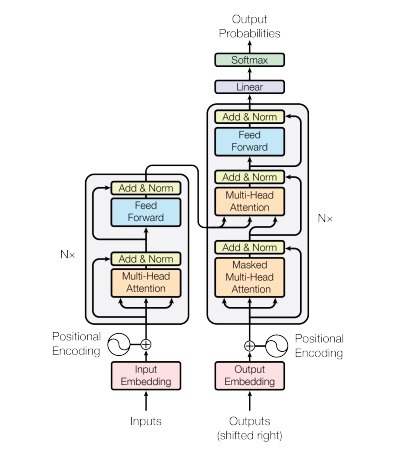
1. **Methodology**

**3.1 Hugging Face transformers and how to use them**

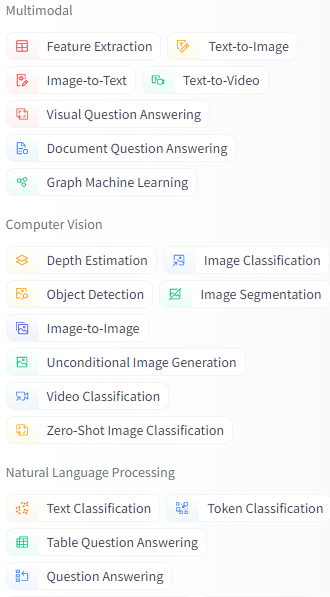
“**Transformer in NLP is a novel architecture that aims to solve sequence to sequence tasks while handling long range dependencies with ease**.”

The concept of transformers **was introduced in 2017** and was influenced by many researchers who introduced several models later.

The transformer language model is composed of **encoder-decoder architecture. These components are connected to each other in the core architecture** but can be used independently as.



* **The encoder receives inputs and iteratively processes the inputs to generate information about which parts of inputs are relevant to each other.** The model will be optimized to get the best understanding from the input.
* **The decoder generates a target sequence using representation from the encoder and uses the contextual information to generate outputs.**

**A key feature of transformer models architecture here is the Attention Layer**. This layer will tell the model to **pay attention to specific details and words.** It can be described as a mapping of a key and a set of key-value pairs to an output, where the **query, keys, values, and output are all vectors.**

Transformers are increasingly the model of choice for NLP problems and this is the reason there have been many developments in this area. Many models which Hugging Face supports are based on the transformer’s architecture. There are a number of **“Translation” and “text2text” based models available in hugging face**.

**3.2 Encoder models**

Use only the encoder of a Transformer model. At each stage**, the attention layers can access all the words in the initial sentence.** These models are often **characterized as having “bi-directional” attention, and are often called auto-encoding models.**

The pre-training of these models usually **revolves around somehow corrupting a given sentence (for instance, by masking random words in it) and tasking the model with finding or reconstructing the initial sentence.**

**Encoder models are best suited for tasks requiring an understanding of the full sentence, such as sentence classification, named entity recognition (and more generally word classification), and extractive question answering.**

Representatives of this family of models include: **ALBERT, BERT, DistilBERT, ELECTRA, RoBERTa**

**3.3 Decoder models**

Use only the decoder of a Transformer model. At each stage, **for a given word the attention layers can only access the words positioned before it in the sentence. These models are often called auto-regressive models.**

The pre-training of decoder models usually **revolves around predicting the next word in the sentence.**

These models are best suited for tasks involving text generation. Representatives of this family of models include: **CTRL, GPT, GPT-2, Transformer XL**

**3.4 Transformers and pipelines**

Hugging Face transformer library consists of different models for different tasks and is accessible through high-level APIs. **Transformer models are complex to build as they would require fine-tuning of tens of billions of parameters and intense training**. The hugging Face transformer library was created to **provide ease, flexibility, and simplicity to use these complex models by accessing one single API.** The models can be loaded, trained, and saved without any hassle.

A typical NLP solution consists of multiple steps from getting the data to fine-tuning a model.



* 1. **Using pre-defined pipelines**

Hugging Face Transformer pipeline performs all pre and post-processing steps on the given input text data**. The overall process of every NLP solution is encapsulated within these pipelines which are the most basic object in the Transformer library.** This helps to connect a model with required pre and post-processing steps and only have to provide input texts.

A typical NLP solution consists of multiple steps from getting the data to fine-tuning a model.



While using pipelines you don’t have to worry **about implementing each of these steps separately.** You can just choose a pipeline that is relevant for **your use case and create a machine translator with a few lines** of code as below:

* 1. **Create your own pipeline**

The **default pipelines only support a few scenarios** for these basic tasks. **Like translation pipeline “English to German”** but what if you want to translate to a different language. For these scenarios, **you will have to create a pipeline using fine-tuned trained models.**

Fortunately, **hugging face has a model hub, a collection of pre-trained and fine-tuned models** for all the tasks mentioned above. These models are based on a variety of transformer architecture – GPT, T5, BERT, etc. If you filter for translation, you will see there are **1423 models as of Nov 2021.** In this section, will see how you can use these models and translate the texts. Let’s create a machine learning translator:

1. **Import and Initialize**

**Transformer models can’t process the raw text and would need to be converted into numbers for models to make sense of the data.**

1. **Import the model**

Can download pre-trained models the same as downloaded the tokenizer in the above step. Here will instantiate a model that contains a base transformer module, given inputs, it will produce outputs a high dimensional vector.

1. **Tokenize and encode the text in seq2seq manner**

For the model to make sense of the data, use a tokenizer that can help with:

* **Splitting the text into words and sub-words**
* **Mapping each token to an integer**

Initialized the tokenizer to get the tokens **for input text.** **The output of tokenizer is a dictionary containing two keys – input ids and attention mask.** **Input ids are the unique identifiers of the tokens in a sentence. Attention mask is used to batch the input sequence together and indicate whether the token should be attended by our model or not.** Token with attention mask **value 0 means token will be ignored and 1 means tokens are important and will be taken for further processing.**

1. **Translate and decode the elements in batch**

Will feed the preprocessed input to the model and the model generates an output vector

**3.7 Some pre-trained transformer models**

**3.7.1 mBART model**

mBART follows **the concept of BART** and is a **sequence-to-sequence de-noising auto-encoder** that was **pre-trained on large-scale monolingual corpora in many languages.** The actual BART maps a corrupted document to the original document it was derived from by randomly shuffling the order of original sentences and replacing the texts with a single mask token. BART was trained by corrupting documents and optimizing the loss between the decoder’s output and the main document. **The corrupted document can be encoded using a directional model and the original document produced using an autoregressive decoder.**

**Pre-train on a subset of 25 languages – CC25 – extracted from the Common Crawl (CC).** **Rebalanced the corpus by up/down-sampling text from each language i with a ratio where pi is the percentage of each language in CC25. Used the smoothing parameter α = 0.7.**

**Pre-processing tokenize with a sentence piece model (SPM**) learned on the full CC data that includes **250, 000 sub word tokens.** While not all of these languages are used for pre-training, this tokenization supports **fine-tuning on additional languages.**  Do not apply additional preprocessing, such as true casing or normalizing punctuation/characters.

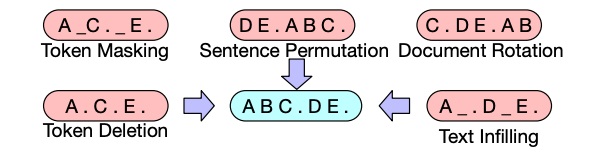
**Used a standard sequence-to-sequence Transformer architecture**, **with 12 layers of encoder and 12 layers of decoder with model dimension of 1024 on 16 heads (∼ 680M parameters).** Include an additional layer-normalization layer on top of both the encoder and decoder, which found stabilized training at FP16 precision. Learning Our training data covers K languages: D = {D1, ..., DK} where each Di is a collection of monolingual documents in language i.

(1) Assume access to a noising function g, defined below, that corrupts text

(2) Train the model to predict the original text X given g(X). More formally, aim to maximize

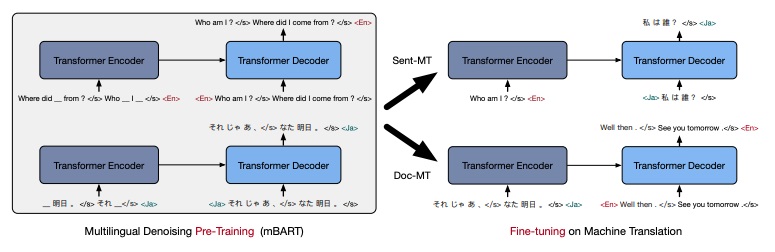
Where X is an instance in language i and the distribution P is defined by the Seq2Seq model.

**For input encoder or transformation,** it allows you to apply any kind of corruption to documents **such as token masking, deletion, infilling permutation, and detection**. BART supports a variety of downstream applications like **Sequence classification, token classification, sequence generation, and machine translation.**



**mBART model was proposed in multilingual de-noising pre-training for neural machine translation.** **While previous models and methods re only focused on encoder, decoder, and transforming the part of the text**, researchers suggested that **using mBART model for the first time can de-noise the full text in multiple languages.**

mBART is a multilingual encoder-decoder (sequence-to-sequence) model primarily intended for translation tasks. As the **model is multilingual it expects the sequences in a different format. A special language id token is added in both the source and target text to identify the language of the text.**



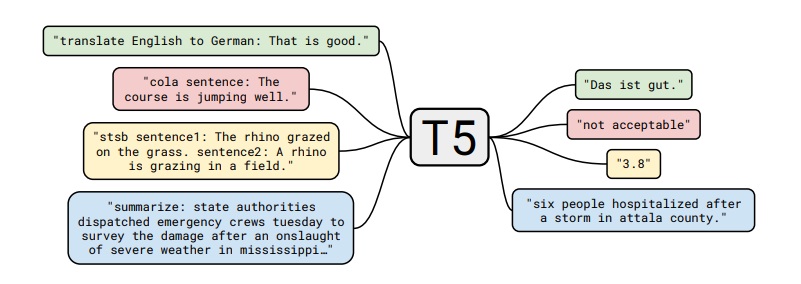
The model is trained **once for all the languages and provides a set of parameters that can be fine-tuned** for **supervised** (Sentence and document level) and **unsupervised machine translation** without any task or language-specific modifications. Let’s see how it handles machine translation in these scenarios:

* **Sentence-level translation:** mBART was evaluated on sentence-level machine translation tasks with a **goal to minimize the difference between source and target sentence representation**. mBART’s pre-training settings provide significant performance improvements compared to other available methods. The model was pre trained using **bi-text (word alignment to identify translation relationship between languages)** only and then combined with back translation.
* **Document-level translation:** **This deals with learning about dependency between different sentences and then translating a paragraph or a whole document.** For pre-processing document level data, **in each block, the sentences should be separated with sentence symbols and an entire training example should end with language id**. While translating **the model will stop when it finds language id as it will not have any knowledge about the number of sentences.**
* **Unsupervised translation:** mBART also supports **unsupervised translation** i.e. no bi-text available. **When there is no bi-text, the model uses back translation and where no bi-text is available but the target language is found in other language pairs in the document collection**, it uses language transfer.

Due to its unique framework, **it doesn’t require parallel data across multiple languages but targeted direction**. **This helps to improve scalability even with a language where do not have enough resources or those resources are domain-specific.**

**3.7.2 T5 model**

This model was introduced in the exploring the limits of transfer learning with a unified text-to-text transfer paper. The researchers re able to explore the effectiveness of transfer learning by introducing a unified framework that can convert all text-based language problems into a text-to-text format. **T5 is based on encoder-decoder architecture and the basic idea is to take text as input and produce a new text as output.**



It follows the same concept as **the original transformer idea. The sequence of input text’s tokens is mapped to a sequence of embeddings to pass it to the encoder**. The encoder consists of blocks, each of them comprising **two parts: a self-attention layer followed by a small feed-forward network. The decoder is similar in structure to the encoder except that it includes a standard attention mechanism after each self-attention layer that attends to the output of the encoder**. **It also uses a form of autoregressive or causal self-attention, which allows the model to attend to past outputs.**

The T5 model was **trained on unlabeled data which was generated using a cleaner version of common crawl, ​​Colossal Clean Crawled Corpus (C4).** With the help of a text-to-text transformer and a new pre-training dataset, the T5 model helped in surveying the vast landscape of ideas.

**There are 5 T5 variants with varying parameters and model sizes.**

Base: Comparable to that of BERT\_base. It is a baseline model with **222 million parameters.**

Small: It is a scaled-down version of the Base model. It only **has 60 million parameters with only 6 layers of encoder and decoders.**

Large: Scaled-up version of the base with **770 million parameters.**

3B: Scaled up version of the base with **3 Billion parameters.**

11B: Scaled up version of the base with **11 Billion parameters.**

T5 model **works will with a wide range of tasks out-of-the-box by prepending a prefix of these tasks to the input sequence e.g. for translation- translate English to French and for summarization- summarize.**

**3.7.3 MarianMT model**

MarianMT is also based on encoder-decoder architecture and was originally trained **using the Marian library.** **Marian, an efficient and self-contained Neural Machine Translation framework consists of an integrated automatic differentiation engine based on dynamic computation graphs**.

Marian is **written entirely in C++.** This library **supports faster training and translation.** As it has minimal dependencies, **it provides support to optimize MPI-based multi-node training, efficient batched beam search, compact implementations of new models, etc.** The Marian toolkit can be used to solve many NLP problems:

* **Automatic post-editing: Focusing on end-to-end neural models to automatically edit the machine translated output**, researchers found that dual-attention mechanisms over two encoders re able to recover missing words from the raw MT output.
* **Grammatical error correction: Marian was also used to produce a set of models for GEC (Grammatical error correction) settings.** The idea was to use low-resource neural machine translation for automatic grammatical error correction(GEC). These methods re used as an extension for Marian to induce noise in the source text, specify lighted training objectives, pre-trained embeddings, transfer learning using pre-trained models etc.

With the Marian framework, it was possible to combine different encoders and decoders and create Marian MT to reduce the implementation effort. **MarianMT model was trained on Open Parallel Corpus(OPUS) which is a collection of translated texts from the b.**

**There are around 1300 models which support multiple language pairs**. All these models follow **the same naming convention – Helsinki-NLP/opus-mt-{src}-{target}**, where **src and target are the two-character language codes. Each model is about 298 MB on disk**, which means it is smaller compared to other models and can be useful for experiments, fine-tuning, and integrating tests. New multi-lingual models in Marian require three-character language codes.

**3.8 Create your own machine learning translator & fine tune them**

Will be using the Hugging face dataset library to find the data need for our modelling**. There are around 1,800 datasets and are specific to different NLP tasks.**

1. Load the data set

Use the load\_dataset function to download and cache the dataset in our notebook.

1. Pre-process the data set

To preprocess the NLP data need to tokenize it using predefined tokenizers.

**T5 model requires a special prefix** to put before the inputs, you should adopt the following code for defining the prefix. **For mBART and MarianMT prefixes will remain blank.**

1. **Train and fine-tune the model**

For our training, will need a few more things.

* **The training attributes that are needed to customize our training.**
* **Will define a data collator to pad the inputs and label them.**
* **And one last thing is to compute the metrics while train the models.**

After fine-tuning the model, the model can be saved in the directory and should be **able to use it like a pre-trained model. Can also push the model to Hugging Face hub and share.**

**3.9 Transformer**

**Basic components**

1. **Create Word Embeddings**

First of all need to convert each word in the input sequence to an embedding vector. Embedding vectors will create a more semantic representation of each word.

Suppose each **embedding vector is of 512 dimension** and suppose our **vocab size is 100,** then our embedding matrix will be of size **100x512**. These matrix will be learned on training and during inference each word will be mapped to corresponding 512 d vector. Suppose have **batch size of 32 and sequence** length of 10(10 words). T**he output will be 32x10x512.**

### **Positional Encoding**

Next step is to generate positional encoding. In order for the model to make sense of the sentence, it needs to know two things about the each word.

**In "attention is all you need paper" author used the following functions to create positional encoding. On odd time steps a cosine function** is used and in even time steps a sine function is used.

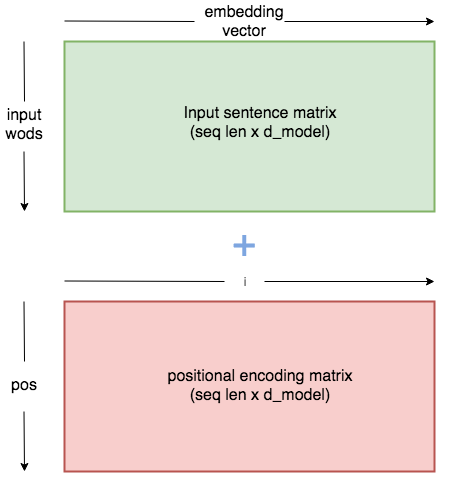


**pos -> refers to order in the sentence**

**i -> refers to position along embedding vector dimension**

Positional embedding will generate a matrix **of similar to embedding matrix**. It will create a matrix of dimension sequence length x embedding dimension. For each token(word) in sequence, will find the embedding vector which is of dimension 1 x 512 and it is added **with the corresponding positional vector which is of dimension 1 x 512 to get 1 x 512 dim out for each word/token.**

if have batch size of 32 and seq length of 10 and let embedding dimension be 512. Then will have embedding vector of dimension 32 x 10 x 512. **Similarly will have positional encoding vector of dimension 32 x 10 x 512. Then add both.**



## Self-Attention

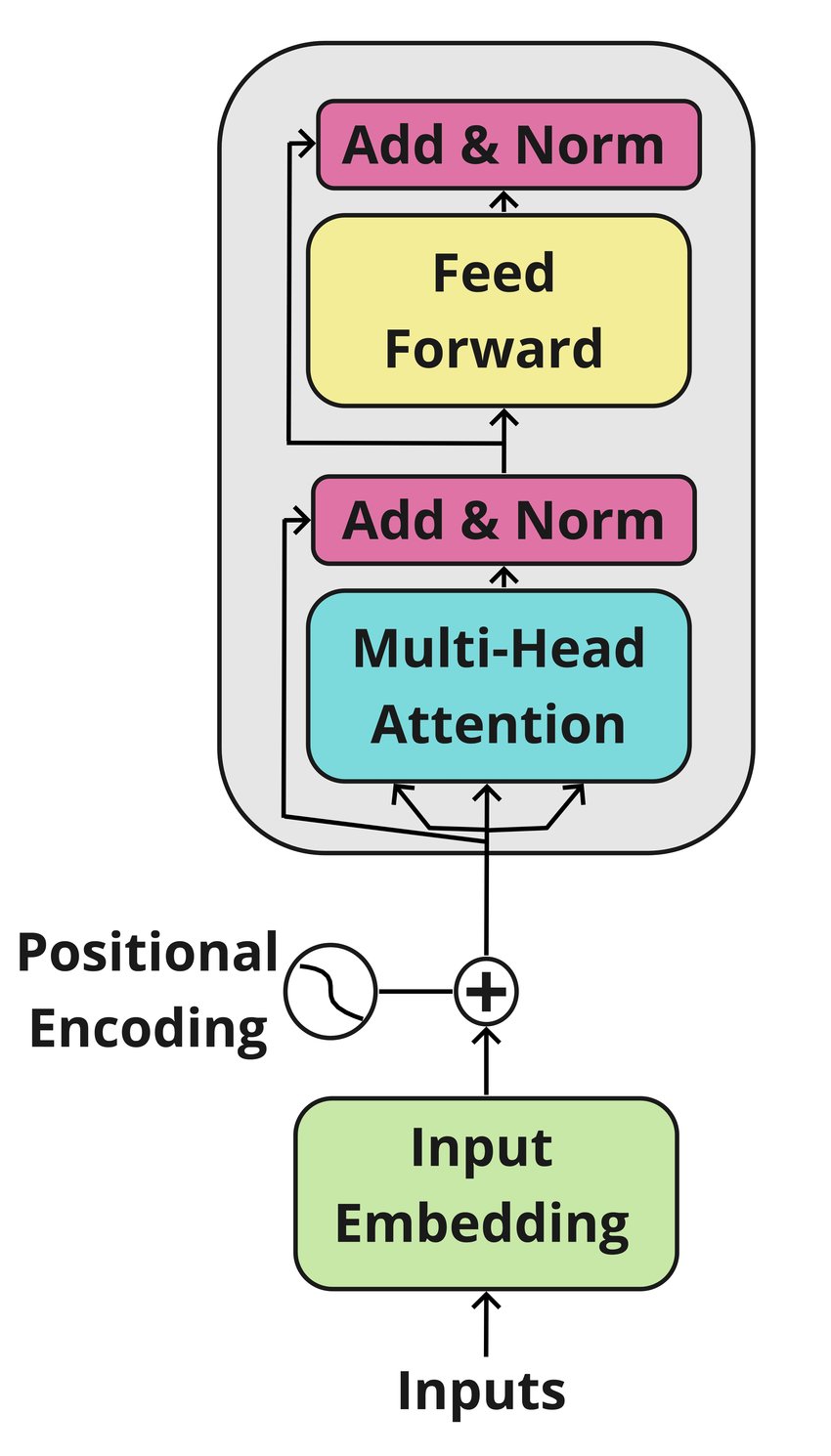
As model processes each word, **self-attention allows it to look at other positions in the input sequence for clues. It will creates a vector based on dependency of each word with the other.**

Let us go through a step by step illustration of self-attention.

* **Step 1: The first step in calculating self-attention is** to create three vectors from each of the encoder’s input vectors (in this case, the embedding of each word). So for each word, **create a Query vector, a Key vector, and a Value vector. Each of the vector will be of dimension 1x64.** Since have a **multi-head attention will have 8 self-attention heads**. Will explain the code with 8 attention head in mind. Suppose have batch\_size=32, sequence\_length=10, embedding dimension=512. So after embedding and positional encoding our output will be of dimension 32x10x512. will resize it to 32x10x8x64.(About 8, it is the number of heads in multi-head attention)
* **Step 2:** **Second step is to calculate the score, will multiply query matrix with key matrix. [Q x K.t].** Suppose our key, query and value dimension be 32x10x8x64. Before proceeding further, **will transpose each of them for multiplication convenience** (32x8x10x64). **Now multiply query matrix with transpose key matrix. ie (32x8x10x64) x (32x8x64x10) -> (32x8x10x10).**
* **Step 3: Now divide the output matrix with square root of dimension of key matrix and then apply Softmax over it**. will divide 32x8x10x10 vector by 8 by square root of 64 (dimension of key matrix)
* **Step 4:** Then this gets multiply it with value matrix. After step 3 our output will be of dimension 32x8x10x10. Now multiply it with value matrix (32x8x10x64) to get output of dimension (32x8x10x64).Here 8 is the number of attention heads and 10 is the sequence length. Thus for each word have 64 dim vector.
* **Step 5:** Once have this will pass this through a linear layer. This forms the output of multi-head attention. **(32x8x10x64) vector gets transposed to (32x10x8x64) and then reshaped as (32x10x512).Then it is passed through a linear layer to get output of (32x10x512).**

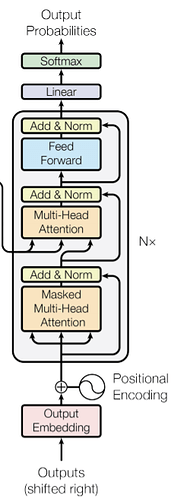
## 4. Encoder

In the encoder section -



* **Step 1:**  **input (padded tokens corresponding to the sentence)** get passes through embedding layer and positional encoding layer. Suppose have input of **32x10 (batch size=32 and sequence length=10).** Once it passes through **embedding layer it becomes 32x10x512.** Then it gets **added with corresponding positional encoding vector and produces output of 32x10x512.** This gets passed to the multi-head attention
* **Step 2:** it will passed through the multi-head attention layer and creates useful representational matrix as output. input to multi-head attention will be a **32x10x512 from which key, query and value vectors are generated** as above and finally produces a 32x10x512 output.
* **Step 3:** Next **normalization and residual connection.** The output from multi-head attention is added with its input and then normalized. **Output of multi-head attention which is 32x10x512 gets added with 32x10x512 input (which is output created by embedding vector) and then the layer is normalized.**
* **Step 4:** Next have **a feed forward layer and a then normalization layer with residual connection from input (input of feed forward layer) where passes the output after normalization though it and finally gets the output of encoder.** The normalized output will be of dimension 32x10x512. This gets passed through **2 linear layers: 32x10x512 -> 32x10x2048 -> 32x10x512. Finally have a residual connection which gets added with the output and the layer is normalized. Thus a 32x10x512 dimensional vector is created as output for the encoder.**

## 5. Decoder



Now have gone through most parts of the encoder. Let us get in to the components of the decoder. Will use the output of encoder to generate key and value vectors for the decoder. There are two kinds of multi head attention in the decoder. One is the decoder attention and other is the encoder decoder attention.

Let us explain with respect to the training phase.

* **Step 1:** First the output gets passed through the **embedding and positional encoding to create an embedding vector of dimension 1x512 corresponding to each word in the target sequence.** Suppose have a sequence length of 10. Batch size of 32 and embedding vector dimension of 512. have input of size 32x10 to the **embedding matrix which produces and output of dimension 32x10x512 which gets added with the positional encoding of same dimension and produces a 32x10x512 out**
* **Step 2:** The embedding output gets passed through a multi-head attention layers as before (creating key, query and value matrixes from the target input) and produces an output vector. This time the major difference is that uses a mask with multi-head attention.

**Why mask?**

Mask is used because while creating attention of target words, **do not need a word to look in to the future words to check the dependency. ie, already learned that why create attention because need to know contribution of each word with the other word. Since are creating attention for words in target sequence**, **For example in word "I am a student", do not need the word "a" to look word "student". For creating attention created a triangular matrix with 1 and 0.eg: triangular matrix for seq length 5**. After the key gets multiplied with query, fill all zero positions with negative infinity, will fill it with a very small number to avoid division errors. (With -1e 20)

* **Step 3:** As before have add and norm layer where add with output of embedding with attention out and normalized it.
* **Step 4:** Next have **another multi-head attention and then add and norm layer. This multi-head attention is called encoder-decoder multi-head attention.** For this multi-head attention create key and value vectors from the encoder output. Query is created from the output of previous decoder layer. Thus **have 32x10x512 out from encoder out. Key and value for all words are generated from it.** Similar query matrix is generated from output from previous layer of decoder (32x10x512). Thus it is passed through a multi-head attention (used number of heads = 8) the through an Add and Norm layer. Here the output from previous encoder layer (previous add and norm layer) gets added with encoder-decoder attention output and then normalized.
* **Step 5:** Next have a feed forward layer (linear layer) with add and nom which is similar to that of present in the encoder.
* **Step 6:** Finally create a linear layer with length equal to number of words in total target corpus and a softmax function with it to get probability of each word.

**4. Conclusion**

Transformers model is a deep learning model that has been in the field for Six years now, and that has led to several top performing and state of the art models such as the BERT model. Giving its dominance in the field of NLP and its expanding usage in other fields such as computer vision, it is important to understand its architecture. This article covers the different components of the transformer and highlights their functionalities.

And have come to revolutionize the world of sequence-to-sequence modelling and started to percolate into a number of exciting applications where their encoder-decoder model provides advantages over other neural network architectures, Also help with dealing large NLP Corpus, We don’t need to retrain from the begin and use pre-trained models.

Hinging on the concepts of attention and self-attention, transformers are able to provide wider range of context in a given sequence, improving results for important natural language processing tasks such as speech recognition or machine translation.

**5.References**

1. [Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, Attention Is All You Need, 2017, arxiv](https://arxiv.org/abs/1706.03762)
2. [Jacob Devlin, Ming-i Chang, Kenton Lee, Kristina Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018 , arxiv](https://arxiv.org/abs/1810.04805v2)
3. [Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, i Li, Peter J. Liu , T5, 2019, arxiv](https://arxiv.org/abs/1910.10683v3)
4. [Lewis Tunstall, Leandro von rra, Thomas Wolf, natural language processing transformers, 2022, oreilly](https://www.oreilly.com/library/view/natural-language-processing/9781098136789/)