Ever since the Transformer architecture was introduced by Google brain in 2017, it has been the standard for solving NLP tasks. Even Google uses BERT to enhance its search engine to understand users' search queries. OpenAI's ChatGPT model has also received all the attention for their ability to generate human-like texts. These Transformers now power applications such as GitHub's Copilot which makes coding easier and faster.

What are Large Language Models (LLMs)?

LLMs are AI models usually derived from the transformer architecture to understand and generate human language, code , etc. They are trained on large amounts of text data, allowing them to capture complexities of human language. They can also perform a wide range of language tasks like text classification, text generation with great degree of accuracy.

LLMs has seen applications in various fields like healthcare for drug discovery and clinical trials; finance for fraud detection and sentiment analysis of financial news, also in customer service like chatbots and virtual assistants.

Since the advent of transformers, using and deploying transformers has exploded. The transformers library and its supporting packages have made using, training and sharing models easy. LLM repositories like hugging face have popped up, making powerful open-source models accessible to the masses.

This course will guide you on how to use, train and optimize LLMs for practical applications.

Language modeling is a subfield of NLP that involves the creation of statistical models for predicting the likelihood of a sequence of tokens in a specified vocabulary(set of tokens).

Tokens are the smallest unit of semantic meaning created by breaking down a piece of text into smaller units and are inputed to a LLM. They can be words or sub-words.

Divided into 2: autoencoding and autoregressive tasks.

Autoregressive LMs are trained to predict the next token in a sentence, based on the previous tokens in the phrase.

Autoencoding LMs are trained to reconstruct the original sentence from a corrupted version of the input.

LLMs are either autoregressive, autoencoding or a combination of the two. Modern LLMs are usually based on transformer architecture although they can be based on another architecture. The key feature of LLMs is their large size and training datasets.

Characteristics of LLMs

The original transformer architecture was a sequence-to-sequence (seq-seq) model which means it has 2 components:

The encoder takes in raw text, splits them,converts them into vectors and uses attention to understand the context of the text.

The decoder excels at generating text by using attention to predict the next best token.The transformer has many other sub-components that makes training faster, generalizable and better performant. Models like BERT and GPT dissect the transformer into only an encoder and decoder respectively in order to build models.

How an LLM is pre-trained and fine-tuned makes all the difference. We will take a quick look into how LLMs are pre-trained to understand what they are good at, what they are bad at and whether we need to update them with our own custom data.

Pretraining

Every LLM has been pre-trained on a large corpus of text data and on specific language modeling related tasks. During pre-training, the LLM tries to learn and understand general language and relationships between words. Every LLM is trained on different corpora and task. Example:

BERT was originally pre-trained on:

English wikipedia 2.5 billion words at the time and

The BookCorpus 800 words and on two specific language modeling tasks:

The masked language modeling task (autoencoding) helps BERT recognize token interactions within a sentence.

The next sentence prediction task helps BERT understand how tokens interact with each other between sentences.

How LLMs Work

How an LLM is pre-trained and fine-tuned makes all the difference. We will take a quick look into how LLMs are pre-trained to understand what they are good at, what they are bad at and whether we need to update them with our own custom data.

Pretraining

Every LLM has been pre-trained on a large corpus of text data and on specific language modeling related tasks. During pre-training, the LLM tries to learn and understand general language and relationships between words. Every LLM is trained on different corpora and task. Example:

BERT was originally pre-trained on:

English wikipedia 2.5 billion words at the time and

The BookCorpus 800 words and on two specific language modeling tasks:

The masked language modeling task (autoencoding) helps BERT recognize token interactions within a sentence.

The next sentence prediction task helps BERT understand how tokens interact with each other between sentences.

Transfer Learning / Fine Tuning

Transfer learning is a technique used in machine learning to leverage the knowledge gained from one task to improve the performance on another related task. Transfer learning for LLMs involves taking a pre-trained LLM on a corpus of text data and finetuning it for a specific “downstream” task such as text classification, by updating the model’s parameters.The idea is the LLM has already learned a lot of information about the language and relationships between word which is a starting point to improve performance on a new task.The data for the downstream task can be of much smaller size.

Fine-tuning has been shown to drastically improve performance on domain-specific and task-specific tasks and lets LLMs adapt quickly to a wide variety of NLP applications.

The basic fine-tuning loop for models are:

Define the model as well as any fine-tuning parameters e.g learning rate

Aggregate the training data (the format depends on the model)

Compute the losses and gradients

Update the model through backpropagation.

However, there are pre-built tools to abstract away the complexities such as Hugging Face’s Transformers package and OpenAI’s Fine-tuning API.

Embeddings

Embeddings are the mathematical representations of words, phrases or tokens in a large-dimensional space. In NLP, embeddings are used to represent words, phrases or tokens in a way that captures their semantic meaning and relationships with other words. Other types of embeddings like position embeddings encode the position of a token in a sentence and token embeddings encode the semantic meaning of a token.

Tokenization

Tokenization involves breaking text down into the smallest unit of understanding i.e tokens. These tokens are the pieces of information that are embedded into semantic meaning and act as inputs to the attention calculations. Tokens make up an LLMs static vocabulary and don't always represent entire words. Tokens can represent punctuation, individual characters, or even a sub-word if a word is not known to the LLM. Nearly all LLMs also have special tokens that have specific meaning to the model. For example, the BERT model has a few special tokens including the [CLS] token which BERT automatically injects as the first token of every input and is meant to represent an encoded semantic meaning for the entire input sequence.

Tokenization can also involve several preprocessing steps like casing, which is the capitalization of the tokens. There are 2 types of casing: uncased and cased. In uncased tokenization, all the tokens are lowercased and usually accents from letters are stripped, while in cased tokenization, the capitalization of the tokens is preserved. The choice of casing can impact the performance of the model, as capitalization can provide important information about the meaning of a token

Out of Vocabulary (OOV) phrases

OOV phrases are phrases/words that the LLM does not recognize as a token and has to split up into smaller sub-words. For example, the name Sinan is not a token in most LLMs so in BERT, the tokenization scheme will split up the name into 2 tokens;

Sin - the first part of the name

##an - a special sub-word token that is different from the word "an" and is used as a means to split up unknown words.

Some LLMs limit the number of tokens we can input at once so how an LLM tokenizes text can matter if we are trying to be mindful about this limit.

Attention

The name of the original paper that introduced the transformer was called “Attention is all you need”. Attention is a mechanism used in DL models that assigns different weights to different parts of the input, allowing the model to prioritize and emphasize (focus) the most important information while performing tasks like translation, leading to improved performance. Unlike

word2vec or gloVe, the attention mechanism takes into account all the words in a sentence during the process of creating a word embedding for a given word in a given sentence.

For example, consider the following list of sentences:

I went to the bank.

I sat on the river bank.

The road will bank to the right.

As you can see, the word “bank” is overloaded in the sense that it has a different meaning in each of the three sentences (and twice as a noun and once as a verb). The attention mechanism in the transformer architecture generates a different context vector (i.e., a one-dimensional vector with floating point numbers). Hence, the same word in different

(and distinct) sentences will have a different word embedding in each of those sentences.

Transformers are a type of deep learning model architecture introduced in the paper "Attention is All You Need" by Vaswani et al. in 2017. They have since become a cornerstone in various natural language processing (NLP) and machine learning tasks. Transformers work through a mechanism called the "self-attention mechanism" and consist of several key components:

Input Embedding: Transformers take a sequence of tokens as input. Each token (e.g., word or subword) is converted into a fixed-dimensional vector representation, often called word embeddings or token embeddings. These embeddings capture the semantic meaning of the tokens.

Positional Encoding: Unlike traditional recurrent models, transformers do not have inherent notions of the order of tokens in a sequence. To address this, positional encodings are added to the token embeddings. These positional encodings provide information about the position of tokens within the sequence.

Self-Attention Mechanism: This is the core of the transformer architecture. Self-attention allows each token to consider the relationships and dependencies between all other tokens in the sequence, which is crucial for understanding the context. The self-attention mechanism computes weighted sums of all tokens, where the weights are learned during training and are based on the similarity between tokens.

The self-attention mechanism operates on a sequence of token embeddings and computes different attention weights for each token based on how it relates to other tokens in the sequence.

The attention weights are used to generate weighted representations for each token, considering its relationships with all other tokens. This enables the model to focus on relevant context and ignore irrelevant information.

4. Multiple Layers: Transformers consist of multiple layers, typically referred to as "transformer blocks" or "encoder-decoder layers." Each layer consists of a stack of self-attention mechanisms and feedforward neural networks.

5. Encoder-Decoder Architecture (for tasks like translation): In sequence-to-sequence tasks, such as language translation, transformers use an encoder-decoder architecture. The encoder processes the input sequence, while the decoder generates the output sequence.

6. Masking: In tasks where it's essential to process sequences with variable lengths (e.g., machine translation), transformers use masking to ensure that each position in the output sequence only depends on positions in the input sequence with valid information.

7. Position-wise Feedforward Networks: After self-attention, each token representation goes through a position-wise feedforward network. This network applies a set of fully connected layers separately to each position, enhancing the model's capacity to capture complex relationships between tokens.

8. Residual Connections and Layer Normalization: Transformers use residual connections and layer normalization to facilitate training deeper networks and improve gradient flow.

9. Output Layer: The final layer in the transformer produces the model's predictions. For sequence-to-sequence tasks, the decoder uses self-attention mechanisms in addition to source-target attention mechanisms to generate the output sequence step by step.

10. Training and Fine-Tuning: Transformers are typically pre-trained on large corpora of text data using objectives like language modeling or masked language modeling (as seen in models like BERT and GPT). After pre-training, models can be fine-tuned on specific tasks using task-specific labeled data.

Transformers have demonstrated remarkable capabilities in various NLP tasks and beyond, including machine translation, text classification, question answering, text generation, and more. Their self-attention mechanism, which allows them to capture complex and long-range dependencies in data, has contributed to their success in understanding and generating natural language text.

Encoder models, in the context of natural language processing (NLP) and machine learning, are a class of deep learning architectures that focus on encoding and understanding input data, particularly in the form of text or sequences. These models are widely used for a range of NLP tasks, including text classification, sentiment analysis, machine translation, question answering, and more. Encoder models are the foundation of many state-of-the-art NLP systems. Here's an overview of how encoder models work and their key features:

1. Input Data Encoding: Encoder models take input data in the form of sequences, such as text. The input sequence is typically tokenized and embedded into continuous vector representations. Each token in the sequence is transformed into a vector through an embedding layer.

2. Deep Neural Networks: Encoder models often employ deep neural networks, such as recurrent neural networks (RNNs), long short-term memory networks (LSTMs), or more commonly, transformers, to process the token embeddings. Transformers, in particular, have gained prominence due to their effectiveness and parallel processing capabilities.

3. Sequence Processing: The primary task of an encoder is to process the input sequence while capturing meaningful contextual information. In the case of transformers, this is achieved through self-attention mechanisms, which allow each token to attend to all other tokens in the sequence, enabling the model to capture dependencies and context effectively.

4. Contextual Representations: As the model processes the input sequence, it updates the token embeddings to produce contextual representations. These representations encode not only the content of the tokens themselves but also their relationships with other tokens in the sequence.

5. Layer Stacking: Many encoder models consist of multiple layers, which enables them to capture increasingly abstract and complex patterns in the input data. In the case of transformers, these layers can be stacked to create deep models.

6. Residual Connections and Layer Normalization: Encoder models often incorporate residual connections and layer normalization between layers to stabilize training and facilitate gradient flow in deep networks.

7. Dimension Reduction: Encoder models may reduce the dimensionality of the representations in higher layers to focus on the most relevant information and reduce computational complexity.

8. Pre-training and Fine-Tuning: Many encoder models are pre-trained on large corpora of text data using objectives like language modeling or masked language modeling. After pre-training, they can be fine-tuned on specific downstream tasks using task-specific labeled data.

9. Adaptability: Encoder models are versatile and can be adapted to various NLP tasks. By fine-tuning the final layers and output, they can be tailored to specific tasks like text classification, sentiment analysis, or named entity recognition.

10. Multimodal Input: While encoder models are often associated with processing text, they can also be extended to handle multimodal data, combining text with other modalities like images, audio, or structured data.

Notable encoder models include BERT (Bidirectional Encoder Representations from Transformers), GPT (Generative Pre-trained Transformer), and various other variants. These models have achieved state-of-the-art performance in a wide range of NLP tasks and continue to advance the field of natural language understanding and processing.

Decoder models, in the context of natural language processing (NLP) and machine learning, are a class of deep learning architectures designed for generating sequences of data. Unlike encoder models, which focus on encoding input data, decoder models specialize in producing sequential output, making them well-suited for tasks like language generation, machine translation, text summarization, and more. Here's an overview of how decoder models work and their key features:

1. Input-Encoding and Context: In many cases, decoder models work in conjunction with encoder models. The encoder processes the input sequence and generates contextual representations, often referred to as the "context" or "thought vector." This context encodes relevant information from the input data.

2. Sequential Output: The primary task of a decoder is to generate a sequence of data. This sequence could be text, translation, summarization, or any other task that requires generating ordered data. For language generation, each step involves producing a token or word one at a time.

3. Deep Neural Networks: Decoder models typically use deep neural networks to generate the output sequence. These networks are often designed as recurrent neural networks (RNNs), long short-term memory networks (LSTMs), transformers, or other sequence-to-sequence models.

4. Autoregressive Generation: Many decoder models use autoregressive generation, where the model produces one token at a time while taking into account previously generated tokens. The model's hidden state or context evolves with each generated token, and the generated token is used as input for the next step.

5. Layer Stacking: Decoder models can consist of multiple layers, allowing them to capture increasingly complex patterns in the output sequence and enhance the quality of the generated content.

6. Attention Mechanisms: Attention mechanisms are commonly used in decoder models to focus on different parts of the input context or previously generated tokens when generating the current token. This enables the model to capture dependencies and context effectively.

7. Residual Connections and Layer Normalization: Similar to encoder models, decoder models may incorporate residual connections and layer normalization between layers to improve training stability and facilitate gradient flow in deep networks.

8. Dimension Reduction: Some decoder models reduce the dimensionality of the hidden representations in higher layers to focus on the most relevant information and reduce computational complexity.

9. Training and Fine-Tuning: Decoder models are often trained with supervised learning, where they are provided with input data and target sequences to generate. After pre-training, they can be fine-tuned on specific tasks with task-specific labeled data.

10. Conditional Generation: Decoder models can perform conditional generation, taking additional context or conditioning information to influence the output sequence. For example, in machine translation, the decoder takes the source language context as a condition to generate the target language translation.

Notable decoder models include models like the GPT (Generative Pre-trained Transformer) series, which are capable of tasks like text generation, language translation, text completion, and text summarization. Decoder models are essential for applications requiring the generation of structured and ordered sequences of data, and their versatility makes them valuable in various NLP tasks and beyond.

Sequence-to-sequence (Seq2Seq) models are a class of deep learning models used for various natural language processing (NLP) and machine learning tasks. They are designed to transform an input sequence into an output sequence, making them versatile for a wide range of applications. Seq2Seq models consist of two main components: an encoder and a decoder. Here's an overview of how they work:

Encoder:

Input Encoding: The encoder takes an input sequence (e.g., a sentence in one language) and processes it step by step. Each element of the input sequence (e.g., a word or token) is embedded into a continuous vector representation, capturing its semantic meaning.

Hidden States: The encoder maintains a set of hidden states that evolve as it processes each element of the input sequence. These hidden states capture the contextual information and dependencies between the input elements.

Context Vector: At the end of the encoding process, the encoder typically generates a single context vector or hidden state that summarizes the entire input sequence. This context vector is a high-level representation of the input and contains essential information.

Decoder:

Initial State: The decoder starts with an initial hidden state or context vector, which is often the context vector generated by the encoder.

Generating Output: The decoder generates the output sequence (e.g., a translated sentence in another language) one element at a time. It predicts each element based on the input context and the elements it has generated so far.

Hidden States: Similar to the encoder, the decoder maintains a set of hidden states that evolve as it generates each element of the output sequence. These hidden states capture the context and dependencies between the output elements.

Generating Output Tokens: For each time step, the decoder uses its hidden state and the previously generated tokens to make predictions for the next token in the output sequence. This can involve using a softmax layer to select the most likely next token from a vocabulary

Recurrent or Transformer Architectures: Seq2Seq models can be built with recurrent neural networks (RNNs), long short-term memory networks (LSTMs), or transformer architectures. Transformer-based Seq2Seq models, like the Transformer and its variants, have gained prominence for their effectiveness in capturing long-range dependencies

Training:

Seq2Seq models are typically trained using paired input-output sequences. The model's parameters are optimized to minimize the difference between the predicted output sequence and the target output sequence. Common loss functions for training Seq2Seq models include cross-entropy loss.

Applications:

Seq2Seq models have found application in various NLP tasks and beyond, including:

Machine Translation: Translating text from one language to another.

Text Summarization: Generating concise summaries of long documents.

Speech Recognition: Converting spoken language into text.

Text-to-Speech Synthesis: Generating natural-sounding speech from text.

Chatbots and Virtual Assistants: Responding to user queries and generating human-like text.

Overall, Seq2Seq models are a versatile class of models that excel in tasks where sequences of data need to be transformed or generated, making them valuable in various real-world applications.

BERT is an acronym for Bidirectional Encoder Representations from Transformers. This is an autoencoding model that uses attention to build a bidirectional representation of a sentence, making it ideal for sentence classification. BERT achieved new SOTA results on eleven natural language processing tasks. Also, BERT is computationally expensive.

BERT Training

BERT underwent pre-training using an extensive dataset without labels, sourced from Wikipedia. Apart from handling natural language processing tasks, BERT offers the flexibility of fine-tuning for diverse objectives. Utilizing the attention mechanism, BERT produces word embeddings that consider both preceding and subsequent words in relation to a given word. In contrast, context-free algorithms like word2vec lack contextual information, resulting in word embeddings devoid of contextual awareness.

BERT Models

The two most renowned variations of BERT are named BERT-Base and BERT-Large. BERT-Base comprises 12 layers, 12 attention heads, and 110 million parameters, while BERT-Large is a more extensive pre-trained model with 24 layers, 16 attention heads, and 340 million parameters. Integration with the Hugging Face transformers library enables the utilization of BERT for diverse tasks, such as question-answering and sequence classification.

Due to its absence of a decoder component, BERT is unsuitable for text generation. Additionally, BERT is not well-suited for unsupervised tasks like clustering or semantic similarity.

BERT Size & Architecture

BERT is built upon the transformer architecture, developed by Google in 2017. Interestingly, BERT only utilizes the encoder part of this architecture to create a language representation model. With a vocabulary size of approximately 30,000 words, each word embedding in BERT comprises 768 features, forming a 1x768 vector of floating-point numbers.

The BERT model specifically employs the encoder component of the transformer architecture, making it an encoder-only design. The core of BERT is a substantial 12-layer neural network designed for text processing. Notably, the final classifier in BERT can be substituted with a task-specific model, allowing the utilization of BERT's acquired knowledge for various tasks.

This process, known as "fine-tuning," involves adapting BERT to a specific task with minimal training compared to the extensive pre-training done by Google. While the fine-tuning step is relatively smaller in scale, it still requires a notable amount of computational resources.

Limitations.

While BERT excels at tasks involving text interpretation and prediction, there are certain limitations to its capabilities. BERT cannot perform the following tasks:

Generate new text

Translate text between languages

Generate responses to questions

These limitations arise because BERT solely relies on the encoder component of the transformer architecture. In contrast, Language Models (LMs) based on the decoder component, such as GPT-based models, have the ability to accomplish all the tasks listed above.

The Generalized Pre-trained Transformer (GPT) marked the pioneering autoregressive model based on the transformer architecture. OpenAI, the organization behind GPT, introduced GPT, GPT-2, and GPT-3 in 2018, 2019, and 2020, with parameters numbering 117 million, 1.5 billion, and 175 billion, respectively. A more recent release, GPT-4, emerged in March 2023, boasting a substantial 1.76 trillion parameters, which is ten times larger than its predecessor, GPT-3.

The original GPT model underwent pre-training using the BookCorpus dataset. The library offers different versions of the model for language modeling and multitask language modeling/multiple-choice classification.

GPT models have exhibited a trend of increasing architectural complexity, coupled with training on progressively larger datasets. Notably, the maximum sequence length for GPT, GPT-2, and GPT-3 is 1 K, 1 K, and 2 K, respectively. The word embedding length for these models also sees a substantial increase from 768 for GPT to 1600 for GPT-2 and a significant leap to 12888 for GPT-3.

The strength of GPT models lies in their proficiency in processing text-based input and generating text-based output. However, it's important to note that GPT models do not support non-text input, such as audio or video, as they are not designed as multi-modal models.

The Main GPT Engines

The GPT APIs offer various language models or engines with distinct capabilities. The four prominent GPT-3 engines, listed from largest to smallest, are Davinci, Curie, Babbage, and Ada. Davinci stands out with the most functionality, making it an excellent option for experimentation, although it requires more resources. Ada, on the other hand, excels in terms of speed, making it the engine of choice for swift performance, while Curie and Babbage are considered viable options for production environments.

Each engine caters to specific use cases and has its own set of advantages:

Davinci is particularly well-suited for creative and analytical outputs.

Curie is optimal for tasks involving classification, language translation, and summarization.

Babbage is designed for semantic search classification.

Ada is tailored for tasks related to parsing text and simple classification.

GPT-2

GPT-2 represents a larger and enhanced iteration of the original GPT model. Its pre-training data comes from WebText, comprising web pages linked from Reddit with 3 or more "karmas" (i.e., upvotes). It's worth noting that GPT and GPT-2 were trained on different datasets, with the former primarily trained on BookCorpus and the latter on web pages, blogs, and news articles linked from Reddit.

GPT-2 operates as a transformer-based decoder-only Language Model (LLM) capable of generating high-quality text. It supports up to 2048 tokens, although this capacity has been surpassed by subsequent models like GPT-3 and GPT-4.

GPT-2 stands out for its ability to produce high-quality output through random sampling. Additionally, it has achieved long-range coherence in terms of the distance between tokens, a feat that traditional models like RNNs and LSTMs struggle to accomplish.

GPT-3

Generative Pre-trained Transformer 3, or GPT-3, stands out as one of the most advanced language models developed by OpenAI, representing a significant advancement in AI language capabilities. Released as the third iteration in the GPT series, GPT-3 is particularly notable for its remarkable ability to generalize from minimal examples and generate coherent text across a diverse array of topics and tasks. This breakthrough, however, comes with a responsibility to consider the ethical use and potential implications of its outputs.

GPT-3 is a monumental leap from its predecessor, GPT-2, boasting an impressive 175 billion parameters compared to GPT-2's 1.5 billion. The model retains the transformer architecture, recognized for its proficiency in handling long-range dependencies in text and widely adopted in state-of-the-art language models.

Trained on a combination of licensed data, human-created training data, and publicly available data in multiple languages, GPT-3's vast dataset, coupled with its extensive parameter count, enables exceptional performance across a diverse range of tasks without the need for task-specific training data.

One groundbreaking feature of GPT-3 is its ability to engage in few-shot learning. This means that by providing GPT-3 with just a few examples of a task, it can generalize and perform that task, eliminating the need for extensive fine-tuning on specific tasks. For instance, GPT-3 can be given a few examples of

translating English to French and then attempt to translate new sentences, even without explicit fine-tuning for translation.

Given its versatility, GPT-3 finds applications in various domains, including text generation (such as essay or poetry writing), answering questions based on provided information, language translation, simulating characters within video games, and generating code based on a description.

T5, or Text-to-Text Transfer Transformer, is a transformer model designed with both encoder and decoder components, making it versatile for various Natural Language Processing (NLP) tasks, ranging from text classification to summarization. Derived architectures from T5 are well-suited for applications that demand the processing and understanding of text, as well as the generation of text.

The source code for T5 is available online and can be downloaded from the GitHub repository: https://github.com/google-research/text-to-text-transfer-transformer. Additionally, T5 can be easily installed using the following command:

Pre-trained on a mixture of unsupervised and supervised tasks, T5 demonstrates effectiveness in tasks like translation. The training process involves a technique known as "teacher forcing," where both an input sequence (designated with input_ids) and a target sequence (designated with output_ids) are necessary for training. The input sequence is processed by the encoder, and the target sequence is passed to the decoder.

A notable feature of T5 is its consistent input/output mechanism across various tasks, including classification, question-answering, and translation. This design allows the same model to be seamlessly applied to multiple tasks without requiring task-specific modifications.

The text classification task assigns a label to a given piece of text. This task is commonly used in sentiment analysis, where the goal is to classify a piece of text as positive, negative, or neutral, or in topic classification, where the goal is to classify a piece of text into one or more predefined categories.

Human Language to Human Language

One of the first applications of attention even before Transformers was for machine translation tasks where AI models were expected to translate from one human language to another. T5 was one of the first LLMs to tout the ability to perform multiple tasks off the shelf . One of these tasks was the ability to translate English into a few languages and back.

Since T5, language translation in LLMs has only gotten better and more diverse. Models like GPT-3 and the latest T5 models can translate between dozens of languages with relative ease.

SQL Generation

If we consider SQL as a language, then converting English to SQL is really not that different from converting English to French . Modern LLMs can already do this at a basic level off the shelf, but more advanced SQL queries often require some fine-tuning.

Free Text Generation

What first caught the world's eye in terms of modern LLMs like ChatGPT was their ability to freely write blogs, emails, and even academic papers. This notion of text generation is why many LLMs are affectionately referred to as "Generative AI", although that term is a bit reductive and imprecise Chatbots etc.

Transformers, what can they do?

Transformers are a type of deep learning model architecture that has had a significant impact on various natural language processing (NLP) and machine learning tasks. Originally introduced in the paper "Attention is All You Need" by Vaswani et al. in 2017, Transformers have since become the foundation for a wide range of applications and have demonstrated remarkable capabilities. Here's what Transformers can do:

Sequence-to-Sequence Tasks: Transformers can perform a wide array of sequence-to-sequence tasks, including machine translation, text summarization, and language generation. Models like the Transformer and its variants, including BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer), have achieved state-of-the-art results in these areas.

Text Classification: Transformers are excellent at text classification tasks, such as sentiment analysis, spam detection, and topic categorization. They can learn to represent and classify text effectively, often outperforming traditional machine learning models.

Named Entity Recognition (NER): Transformers can be used for NER tasks, where they identify and classify named entities like people, organizations, and locations in text.

Text Generation: Transformers are capable of generating human-like text. GPT-3 and GPT-4, for example, have been used to create content, write code, and even engage in natural conversations with users.

Question Answering: Transformers can be used in question-answering systems that can extract answers from text or knowledge bases. For instance, models like BERT have been fine-tuned for this purpose.

Language Understanding: Transformers are essential for language understanding tasks, as they can capture the nuances and context of language. This is crucial for chatbots, virtual assistants, and other applications where understanding user input is vital.

Image Captioning: Transformers can be combined with computer vision models to generate textual descriptions or captions for images. This enables applications like automated image tagging and assistive technologies for the visually impaired.

Speech Recognition: Transformers are used in automatic speech recognition (ASR) systems to transcribe spoken language into text. They help improve the accuracy of speech-to-text conversion.

Text Summarization: Transformers can generate concise summaries of long text documents, making it easier to digest large amounts of information.

Language Translation: Transformers are the foundation of many machine translation systems, like Google Translate, that enable the translation of text from one language to another.

Chatbots and Virtual Assistants: Transformers have been employed in developing conversational agents and virtual assistants like Siri, Alexa, and chatbots that can understand and generate human-like text in real-time conversations.

Sentiment Analysis: Transformers are widely used for sentiment analysis tasks, helping determine the emotional tone of a piece of text, such as whether a review is positive or negative.

Recommendation Systems: Transformers can be used to build recommendation systems by processing user interactions and content to provide personalized recommendations, as seen in platforms like Netflix and Amazon.

Language Understanding and Generation Across Languages: Transformers can be fine-tuned for multiple languages and support multilingual applications, making them versatile for global use.

Document Classification: Transformers are employed in document categorization tasks, such as classifying articles, legal documents, or research papers into specific categories.

Transformers have become the backbone of many NLP applications and have demonstrated the ability to understand and generate text in a human-like manner. Their pre-trained models can be fine-tuned for specific tasks, reducing the need for extensive labeled data and making them highly adaptable to a wide range of applications across various domains. They continue to be a driving force in the advancement of NLP and machine learning.

Working with pipelines in natural language processing (NLP) typically involves using predefined sequences of NLP tasks or components to process and analyze text data efficiently. Pipelines simplify the development process by automating many of the common tasks. Below is a brief overview of how to work with pipelines in NLP:

1. Select an NLP Library or Framework: Choose an NLP library or framework that provides pipeline capabilities. Some popular choices include spaCy, Hugging Face Transformers, NLTK (Natural Language Toolkit), and Gensim.

2. Define the Pipeline: Create a pipeline by specifying the sequence of NLP tasks you want to perform on your text data. Common tasks in a pipeline may include tokenization, part-of-speech tagging, named entity recognition, sentiment analysis, and more.

3. Load or Preprocess Data: Prepare your text data by loading it into the NLP framework or library. This may involve reading text from files, databases, or web sources and performing any necessary preprocessing steps, such as cleaning, lowercasing, or encoding.

4. Instantiate the Pipeline: In your chosen NLP library, instantiate a pipeline object and configure it with the tasks and components you defined in step 2. This sets up the sequence of NLP operations to be executed.

5. Process Data: Apply the pipeline to your text data. This will automatically run the predefined NLP tasks on your text, producing the desired output. The pipeline takes care of passing data between tasks and handling intermediate results.

6. Access Results: After processing your text data with the pipeline, you can access the results of each task. These results may include tokenized text, part-of-speech tags, named entities, sentiment scores, or any other information generated by the pipeline components.

7. Customization: Some NLP libraries allow you to customize or extend the pipeline by adding or replacing components to tailor the processing to your specific needs. You can add custom functions or components for tasks like domain-specific entity recognition.

8. Post-Processing: Depending on your application, you may need to perform additional post-processing or analysis on the pipeline's output. This can include aggregating information, generating reports, or integrating the results into other applications.

9. Evaluation and Fine-Tuning: Evaluate the pipeline's performance on your specific tasks or datasets. If necessary, fine-tune the pipeline by adjusting configurations, component choices, or training on custom data.

Using pipelines in NLP can save you time and effort by automating common text processing tasks and allowing you to focus on specific analysis or application-related tasks. Depending on the NLP library or framework you choose, the specific steps and capabilities may vary, but the general process is similar. For the rest of the course, we will be using Hugging Face Transformer Pipeline

Hugging Face has developed a Python-based transformers library and an open-source repository dedicated to building models based on the transformer architecture, which is accessible at https://github.com/huggingface/transformers.

The Hugging Face open-source transformers library consists of three main components:

1. Model Classes: This component encompasses more than two dozen pre-trained transformer models, providing a wide range of options for different NLP tasks and architectures.

2. Configuration Classes: Configuration classes offer various classes that contain the parameters essential for instantiating a model. These classes allow users to customize and fine-tune the behavior of the models according to their specific needs.

3. Tokenizer Classes: Tokenizer classes are crucial components that endow each model with a vocabulary and methods for encoding/decoding strings. They manage token embeddings and indices, facilitating the conversion between raw text and the tokenized input required by transformer models.

The combination of these three components in the Hugging Face transformers library provides a comprehensive and flexible toolkit for working with transformer-based models, enabling users to leverage pre-trained models, customize configurations, and handle tokenization efficiently.

The transformers library and NLP Tasks

Indeed, the transformers library by Hugging Face is a versatile and comprehensive resource for pre-trained models in the field of Natural Language Processing (NLP). It supports a wide array of models for various NLP tasks and languages, and it ensures compatibility with popular deep learning frameworks such as TensorFlow 2 and PyTorch.

Among the notable pre-trained models supported by Hugging Face are:

1. BART (from Facebook)

2. BERT (from Google)

3. DeBERTa (from Microsoft Research)

4. DistilBERT (from Hugging Face)

5. GPT-2 (from OpenAI)

6. Longformer (from AllenAI)

7. Pegasus (from Google)

8. Reformer (from Google Research)

9. RoBERTa (from Facebook)

10. SqueezeBert

11. T5 (from Google AI)

12. XLM-RoBERTa (from Facebook AI)

13. XLNet (from Google/CMU)

These models cover a broad spectrum of NLP tasks and are trained on diverse datasets, making them valuable resources for researchers, developers, and practitioners working in the field of natural language understanding and generation. The flexibility to work with different models and the support for various languages contribute to the popularity and usability of the transformers library by Hugging Face.

The Transformer Pipeline Class

The transformers library provides a standard interface to multiple models which simplifies the task of integrating language models into custom applications. Moreover, the transformers library supports popular deep learning frameworks, including:

JAX: A library for high-performance numerical computing developed by Google, especially suited for machine learning and deep learning tasks.

PyTorch: An open-source deep learning framework developed by Facebook that provides dynamic computational graphs.

TensorFlow: An open-source machine learning framework developed by the Google Brain team, known for its flexibility and scalability.

By supporting these frameworks, the transformers library ensures that users can choose the deep learning framework that best fits their needs or aligns with their existing infrastructure. This flexibility is crucial for researchers and practitioners who may have preferences or specific requirements related to the deep learning framework they want to use in their projects.

The transformer library also provides a pipeline() class that you can use in a pipeline-based mechanism. The following code block illustrates how to instantiate a transformer pipeline for various NLP tasks that are specified inside double quotes (complete code samples are discussed later):

As you can see in the preceding code block, the pipeline() class supports NER, sentiment analysis. In addition, the transformers library supports the preceding NLP tasks in more than 100 languages. The Hugging Face Hub contains numerous models:

https://huggingface.co/models

Some of the currently available pipelines are:

feature-extraction (get the vector representation of a text)

fill-mask

ner (named entity recognition)

question-answering

sentiment-analysis

summarization

text-generation

translation

zero-shot-classification

By default, this pipeline selects a particular pretrained model that has been fine-tuned for sentiment analysis in English. The model is downloaded and cached when you create the classifier object. If you rerun the command, the cached model will be used instead and there is no need to download the model again.

There are three main steps involved when you pass some text to a pipeline:

The text is preprocessed into a format the model can understand.

The preprocessed inputs are passed to the model.

The predictions of the model are post-processed, so you can make sense of them