Large Language Models from Scratch

Author: Victor Dantas
Version: April 2023

Introduction

To learn how large language models (like GPT, LamDA, and others) work, it's useful to understand how transformers work. To understand how transformers work, it's good to understand how the attention mechanism works. To understand how the attention mechanism works, it's useful to understand how sequence-to-sequence (seq2seq) models work. To understand how seq2seq models work, it's useful to understand how recurrent neural networks (RNNs) work. To have a better understanding of RNNs, one should have a foundational knowledge of neural networks and deep learning. To learn about neural networks and deep learning, you probably first need to understand some basic machine learning concepts.

Most learning materials on LLMs out there are either too shallow or too deep, not bringing all the building blocks together, and often assuming some existing knowledge. This document is an attempt to take you on a linear path "from zero to hero", starting from the basics. With the help of some linked videos and other resources, it will hopefully help you build an intuition around the building blocks that make up LLMs.

Encoding Text

Machine learning (ML) algorithms are fundamentally based on math and statistics. They work with numbers, not words. So, for natural language processing (NLP) systems, we need a way to encode words into numbers.

A naive approach would be to assign each word from a dictionary a unique number. For example:

Word	**Number**
aardvark	1
…	…
king	2551
…	…
queen	3122
…	…

The first major problem with this approach is that large numbers (like the ones shown in the preceding table) may cause issues during the gradient descent steps (more on that later) of the ML training process. In short, such numbers may prevent the training process from converging to an optimal solution. A solution for this would be to apply the one-hot encoding technique.

In one-hot encoding, words are mapped to a vector of the same length as the dictionary. Each index in the vector represents a word in the dictionary, and to map a word to its vector representation, we put “1” in the position corresponding to the word’s index, and “0” everywhere else. For example:

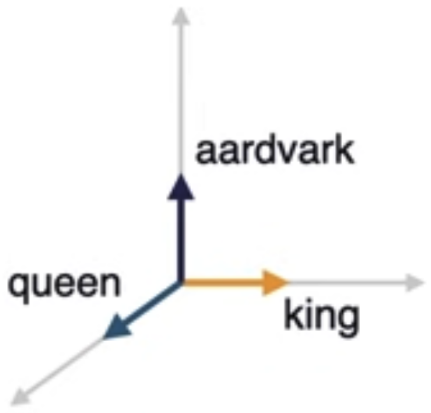
Word	**One-hot vector**
aardvark	[1, 0, 0, …]
…	…
king	[0, 0, 0, …, 1, 0, …]
…	…
queen	[0, 0, 0, …, 0, …, 1, 0, …]
…	…

Note: the one-hot vectors all have the same size.

While one-hot embeddings do use values that are amenable to machine learning (0s and 1s, which are great), they have a few drawbacks:

- One-hot encoding is a sparse representation that requires a very large vector size to capture all words in a large vocabulary (such as the English language).
- They don’t encode any meaning and knowledge about words or their relationships. For example, the words “king” and “queen” are closer to each other in meaning than they are to “aardvark”. And yet, these one-hot vectors are all orthogonal to each other and separate by the same distance.

To illustrate both of these points, let’s consider a dictionary that only contains these three words: aardvark, king, and queen. Their vector representations can therefore be mapped to a 3D space as shown in the following image:



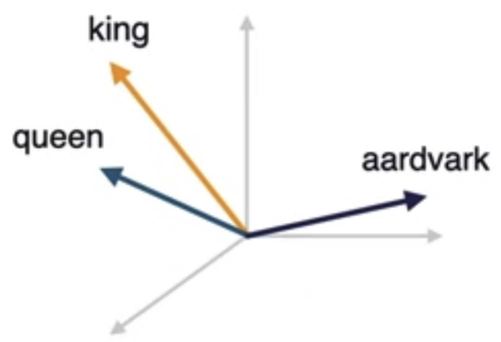
A simple example of one-hot encoding applied to a 3-word vocabulary (source: [Transfer learning and Transformer models (ML Tech Talks)](#))

As you can see, there’s clearly a wasted opportunity by not “filling up” the space (i.e., using a denser representation as opposed to the sparse one we see). And, because of that, we cannot use this to infer anything about the meaning of the words. The word “king” here is about as distant to the word “queen” as it is to “aardvark”.

A more ideal solution then is to map words to a continuous vector, allowing words to occupy the entire vector space. Building on the same example, such a representation would be as follows:

Word	Continuous vector
aardvark	[0.3, 1.9, -0.4]
king	[2.1, -0.7, 0.2]
queen	[1.5, -1.3, 0.9]

These continuous vector representations can be visualized as follows:



An example of continuous vector representation applied to a 3-word vocabulary (source: Transfer learning and Transformer models (ML Tech Talks))

Such a technique is what is referred to as **word embeddings** and the numbers in the vector - in case you were wondering where they come from - are not predefined, but *learned*.

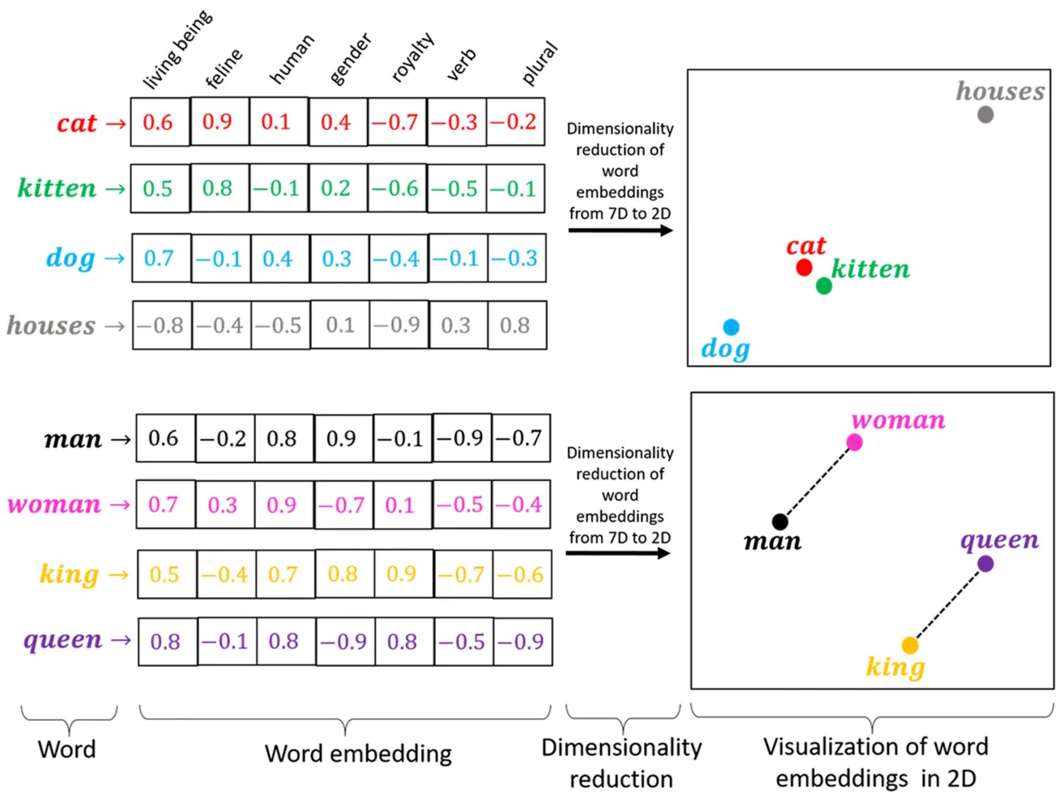
Word Embeddings

A word embedding is a learned representation of a word as a real-valued vector. That vector encodes the meaning of the word in a way that words that are close to each other in their vector representation are expected to have similar or related meanings, or at least similar syntactic functions.

Think about it this way: you map each word in a vocabulary to a vector in a multi-dimensional space. And, just like you can use algebra to calculate the distance between two vectors, or to add two vectors to obtain a third vector, you can do similar operations on these word vectors. The distance between two vectors can be used as a measure of similarity between words (in terms of semantics, or meaning). For example, if you were to subtract “man” from “king”, and then add “woman”, you obtain a vector that is very close to the “queen” vector.

In practice, there are many dimensions to this vector space which makes them impossible to visualize in our minds. But in the world of math, there's no limit to how many dimensions a vector space can have and the underlying algebra works out in the same way.

One intuitive way to think about the "dimensions" of this space is that they represent word categories - or attributes we would use to classify words. So, in an embedded word vector, each index will correspond to a category, and the value in that index corresponds to how much that word is associated with that category (and again, these values are learned, not programmed into the algorithm). The following image illustrates this:



	living being	feline	human	gender	royalty	verb	plural
cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8
man →	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman →	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king →	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen →	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

An illustration of word embedding (source: Word Embedding: Basics, Medium article)

And how are the meanings of words learned? The representations are learned based on their usage. This results in words that are used in similar ways having similar representations, "similar" here meaning that the vectors are close together in space.

This notion of letting the usage of a word define its meaning can be summarized by a quote by John Firth:

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

Example techniques for implementing word embeddings include Word2Vec, Continuous Bag-of-Words (CBOW), skip-gram, and Global Vectors for Word Representation (GloVe).

Note: Models such as GPT-3 and similar do not use traditional word embedding algorithms like Word2Vec or GloVe. Instead, they use a transformer architecture with self-attention mechanisms to learn contextualized token representations which take into account each token's relationships with other tokens in the input sequence. The concept is the same -

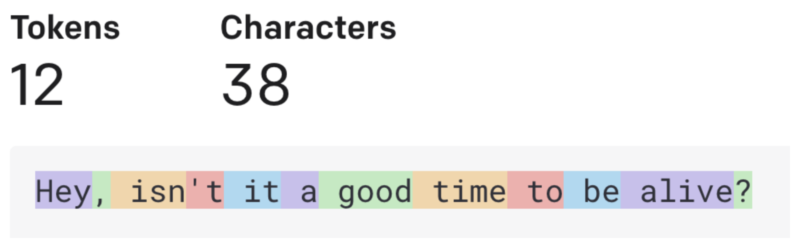
mapping words to a dense vector representation - but the representations are learned differently. More on that later.

Tokenization

Typically, models that process language don't actually use words. Instead, they use what are referred to as *tokens*.

There are different ways to *tokenize* words and sentences. For example, a simple and naive way is to split a sentence by whitespaces. This way, in the sentence "Hey, isn't it a good time to be alive?", we end up with the tokens: "Hey,", "isn't", "it", "a", "good", "time", "to", "be", "alive?". The problem with this approach is that "Hey,", "isn't", and "alive?" will include punctuation in them. So, if these words appear in a different sentence but without the punctuation, the model would learn different representations for them, which is not ideal.

In practice, models like GPT use a tokenizer that takes this and other such nuances into account and they'll often split words into pieces, or "subwords". The above sentence is tokenized by the GPT family of models as follows:



Tokens 12 Characters 38

```
Hey, isn't it a good time to be alive?
```

Tokenization of the sentence "Hey, isn't it a good time to be alive?". Different colors demark different tokens. Source: OpenAI Tokenizer

Open AI offers a tool to explore their tokenizer and see what results you get for different input text: https://platform.openai.com/tokenizer.

So, in summary: the way sentences are "read" and processed by NLP models is by first tokenizing the sentence, then embedding each token using one of the word embedding techniques we discussed.

Neural Networks, Deep Learning, Gradient Descent, and Backpropagation

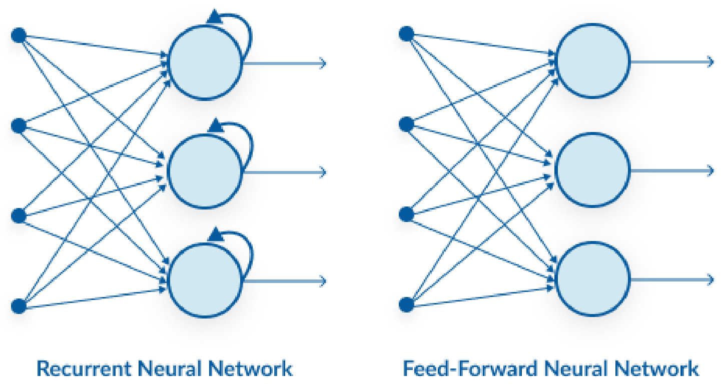
The following is a Youtube playlist from *3Blue1Brown* containing four videos that give a nice introduction (with great visualizations) to neural networks and concepts such as gradient descent and backpropagation (total watch time: **64min**):

But what is a neural network? | Chapter 1, Deep learning

If you're not already familiar with these concepts, I'd recommend watching this video series before continuing. These videos probably do a better job than any written content you'd spend the same amount of time reading would.

Recurrent Neural Networks

While traditional neural networks were capable of doing many things, they didn't have a good way to store some kind of state or internal memory, which is useful for processing variable-length sequences of inputs. A recurrent neural network (RNN) solves this problem by introducing a cycle in the connections between nodes, allowing output from the network to affect subsequent input, effectively creating an internal memory (as the previous output is "remembered" each time).



A basic illustration of the difference between a recurrent neural network and a traditional feed-forward neural network (source: Uditvani.com)

The following video (**22 min**) does a great job of introducing the concept of recurrent neural networks in a very intuitive way and I recommend watching it before continuing:

A friendly introduction to Recurrent Neural Networks

RNNs by design take two inputs, the current example they see, and a representation of the previous input (hence why they're called "recurrent"). This is the reason they perform well when dealing with sequence-related tasks: the sequential information is preserved (in a so-called hidden state, or *context*) and used in the next instance. RNNs are often applied in natural language processing.

A commonly used variant of RNNs is the so-called long short-term memory (LSTM), which has higher memory power.

Seq2Seq Models and Attention Mechanism

Sequence-to-sequence (seq2seq) models are deep learning models that take a sequence of items as an input (such as a sequence of words, or *tokens*) and output another sequence. They have been very successful in tasks such as image captioning, text summarization, and translation (Google Translate started using such a model in production in late 2016).

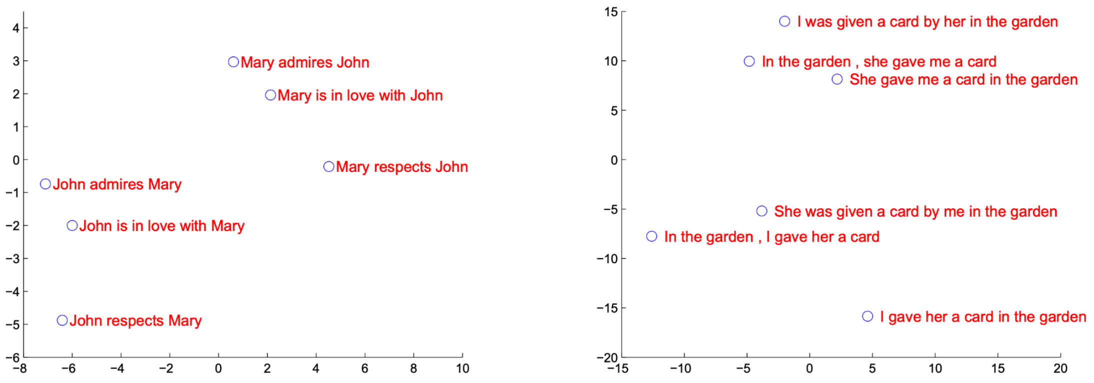
Attention mechanism

To explain the attention mechanism, let us consider the Neural Machine Translation seq2seq model. This model consists of two components: an encoder, whose job is to encode the input sequence into continuous vector representations, and a decoder, whose job is to generate the output sequence. Both encoder and decoder tend to be recurrent neural networks (RNNs).

Let us first discuss how these models work *without* the attention mechanism.

The encoder processes each item in the input sequence, compiling them into a vector (called the context). The size of the context vector can be set ahead of model training. Essentially, the context vector summarizes the entire input sequence into a fixed-length vector.

The concept of a context vector is somewhat similar to that of word embedding but applied to an entire sequence (of words). The following image, from the paper *Sequence-to-Sequence Learning with Neural Networks*, illustrates the learned context vector representations for different input sentences:



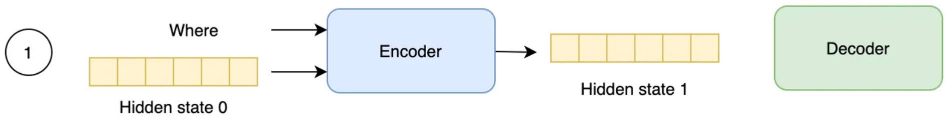
Learned context vector representations in a 2-dimensional projection (source: [Sequence-to-Sequence Learning with Neural Network](#))

You can see from the preceding image that these representations are sensitive to word order, which is indeed what we want when it comes to natural language processing. For example, “Mary admires John” is relatively far in meaning to “John admires Mary” (as they express entirely different ideas despite the two sentences having the exact same words), and “Mary admires John” is relatively close to “Mary is in love with John”, since “is in love with” and “admires” aren’t too far apart, semantically speaking.

After processing the entire input sequence and sending the context vector over to the decoder, the decoder then begins producing the output sequence. A helpful animation that illustrates this process on a high level is available in the blog article Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention) (direct link to animation here).

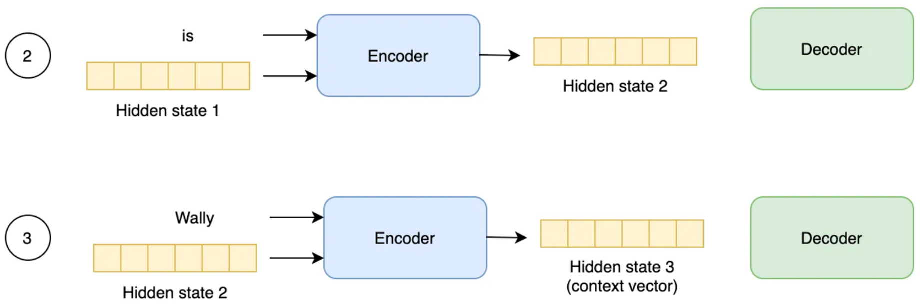
Since RNNs work by processing sequences item by item, we can think of them as processing a series of steps sequentially, where each time step represents one item in the sequence. Therefore, to understand how the encoder and decoder RNNs work in more detail, let's see what happens at each time step as it applies to the example of translating "Where is Wally" to Italian:

1. The input word from the sequence is converted into a vector using a word embedding algorithm (not shown in the image below). The encoder RNN takes two inputs: the embedded word vector and the current hidden state from the previous step. In the first step, this will be a special initial hidden state.



Source: Two minutes NLP — Visualizing Seq2seq Models with Attention, Medium article

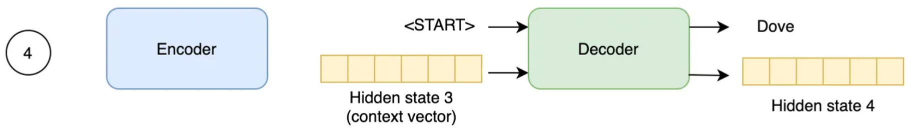
2. The encoder RNN processes the input and updates and outputs the new hidden state (hidden state 1 in the preceding image).
3. The process repeats until all input words are processed by the encoder. The final hidden state (called the **context vector**) is handed over to the decoder



Source: Two minutes NLP — Visualizing Seq2seq Models with Attention, Medium article

Then, the decoder RNN starts doing its thing. At each time step:

1. The decoder RNN takes as input the hidden state vector plus the previous output token (in the first time step, the hidden state vector will be the encoder's context vector, and the token will be the special <START> token).



Source: Two minutes NLP — Visualizing Seq2seq Models with Attention, Medium article

2. The decoder RNN outputs a sequence item ("Dove" in the preceding image) and updates its hidden state vector.
3. Repeat.

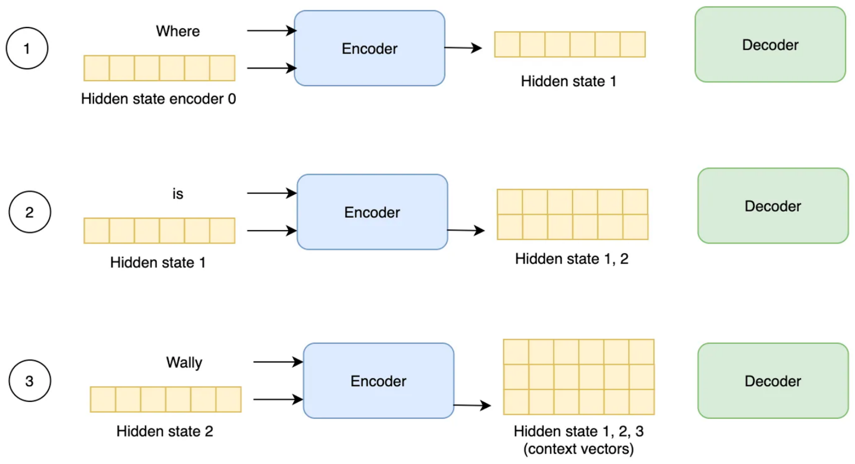
The drawback of having a single context vector as an output for the entire input sequence is that, in the case of long sequences, there is a high probability that the initial context has been lost by the end of the sequence. Remember that the context vector is a fixed-length vector, so the longer the input, the more information is "lost". But even for short sequences, this approach still has the drawback of not capturing the fact that different input words may be relevant at different steps.

Enters **Attention**.

Since the problem was that a single context vector is probably not enough to capture the context of a long sequence, then how about having one vector per input sequence item (i.e., one vector per token)? That's exactly what attention is: the encoder sends in as many hidden state vectors as there are tokens in the input sequence. In addition, the decoder assigns different attention weights to different hidden states at each time step, allowing the model to "focus" on the relevant parts of the input sequence as needed. In other words, if a word used way back at the beginning of a sentence is considered relevant (semantically and contextually speaking) for the current word being processed by the decoder (we'll talk more about how relevance is measured later), this word will be more prominent for the decoder and therefore will affect the output more strongly than for example the word that came just before.

So, if we apply the attention mechanism, the new flow is as follows:

1. The input word from the sequence is converted into a vector representation (same as before)
2. The encoder RNN then takes two inputs: the embedded word vector and the hidden state vectors from the previous step.
3. The encoder RNN outputs a new hidden state vector for the current token (appending to, not replacing, the previous ones).

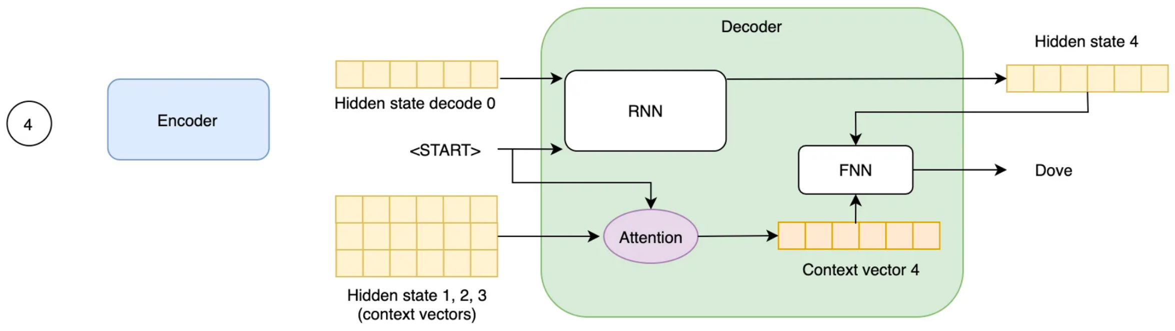


Source for image: *[Two minutes NLP — Visualizing Seq2seq Models with Attention, Medium article]*

Once the entire input sequence is processed, the decoder RNN will then:

1. Take the previously generated token and the decoder's own hidden state vector which we will refer to here as the *target hidden state* vector
2. [**Attention step**] Take all the context vectors produced by the encoder RNN (each vector is most associated with a certain word in the input sequence) and use the current target hidden state to derive attention weights for each input vector. In other words, the decoder determines the relevance of each input word at the current time step by assigning a higher attention weight (i.e., a higher *score*) to the most relevant words.
3. Multiply each encoder context vector by their respective score (this is done via a matrix multiplication), and then by a softmax function to bring these values down to a scale from 0 to 1 and in a way that the weights sum up to 1. This effectively creates a probability distribution over the encoder's hidden states.
4. Sum up all the weighted vectors (where the weights are the attention weights) and concatenate the resulting vector with the decoder's current target hidden state to yield the final attention vector. This vector represents the relevant information from the input sequence for the current decoding step. The attention weights determine the contribution of each encoder hidden state (effectively the contribution of each input word, since each hidden state is most associated with a word) during the decoding step.
5. The resulting attention vector is then passed through a fully connected feed-forward neural network (FNN) layer followed by a softmax activation function, which outputs a probability distribution over the target vocabulary. The token with the highest probability is selected as the output token. The output token plus the updated hidden state is passed as inputs to the next time step.

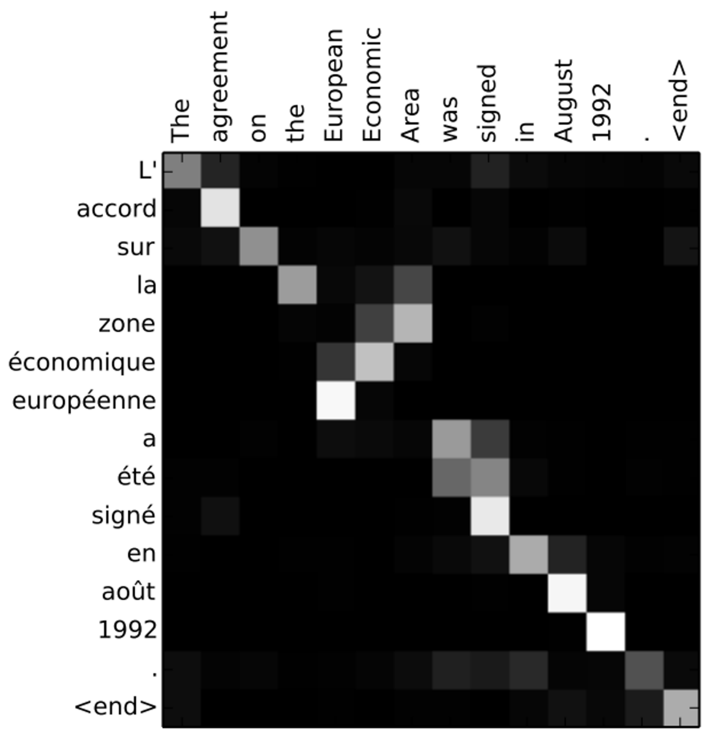
The full decoding step with the attention mechanism is illustrated in the following image:



Source for image: Two minutes NLP — Visualizing Seq2seq Models with Attention, Medium article

To more concretely illustrate the attention mechanism, consider the following sentence: "The agreement on the European Economic Area was signed in August 1992." as an input to a translation model which will translate it to French.

The following matrix shows, for each generated French word, what was the attention score assigned to each of the English words in the input sentence (the lighter the color, the higher the score):



Source: [*Neural Machine Translation by Jointly Learning to Align and Translate. Paper*]()

For example, after the first 4 tokens (*L'accord sur la…*), the attention layer will look at the calculated attention scores (see the next row in the matrix, the one indexed by "zone") and find that the most relevant words are "Area" followed by "Economic". That helps the decoder predict "zone" (French translation of "area"), thus effectively capturing the right word order in the target language. In statistical machine translation, this "what is translated to what at a given position" or the different word orders between languages are referred to as "alignment" (hence the title of the paper "*Neural Machine Translation by Jointly Learning to Align and Translate*"). The attention mechanism helps establish the word alignment between languages during translation.

And how are the attention scores calculated? The attention weights are updated during the training process as the model learns to align the input and output sequences. This is achieved through back-propagation and optimization techniques, such as stochastic gradient descent or *Adam*. In other words, **the model is learning by itself which words to "pay attention" to, or how relevant other words are for a given word in a sentence**.

Transformers

With an intuitive understanding of embeddings, RNNs, sequence-to-sequence (seq2seq) models, and the attention mechanism, we now have a good foundation for understanding Transformers.

The Transformer was proposed in the seminal paper Attention is All You Need in 2017. Currently, they are the standard model used not only for seq2seq tasks but also for language modeling.

An Intuition for How Transformers Work and How They're Different

Transformer models operate solely on attention mechanisms, i.e. no recurrence is used (therefore no RNNs). The main motivation for moving away from RNNs is that they're not very efficient when processing longer sequences, even with the attention mechanism. RNNs (and even their more powerful variants such as LSTMs) still have a limited reference window, i.e., they can only look back at so many words before running into issues (such as *vanishing gradients*). In addition, the encoder needs to process the input sequentially, left to right, and is therefore not capable of considering words that appear to the right when encoding a particular word. To illustrate this limitation, take the following two sentences:

1. On the river bank…
2. On the bank of the river…

The first sentence can probably be encoded properly by RNN encoders, since by the time the encoder is processing the word *bank*, it has already "seen" the word *river* and should be able to disambiguate it from the word *bank* as a financial institution. However, the second sentence presents a problem: the encoder will see the word *bank* first, before seeing the word *river*. Therefore, when the encoder is processing the word *river*, *bank* has already been encoded (most likely as a financial institution as that is how the word is most often used).

Another consequence of this sequential processing of the input is that parallelization can't be used during training.

In contrast, a Transformer's encoder will "attend' to tokens all at once by applying the attention mechanism already at the encoding steps - a mechanism that is referred to as **self-attention**, and proceed to encode all tokens in parallel thanks to **positional encoding**. Positional encoding allows Transformers to still take word order into consideration (crucial for language processing) but without the need to process words sequentially.

Relying entirely on self-attention and positional encoding for tasks such as translation and language modeling are the two main innovations introduced by the Transformer, and we'll explore these concepts in more detail.

Self-Attention

If you recall from the previous section, the attention mechanism in a seq2seq encoder-decoder model is used during the decoding step, where the decoder "attends to" all encoder states and updates its current decoder state. The encoding of each token was done sequentially and independently, i.e., the encoder didn't attend to all the tokens in the input sequence to better set each token's individual representation. It was the decoder's job to look at the wider picture.

In contrast, self-attention (also referred to as intra-attention) happens during both encoding and decoding. With self-attention, the encoder will "look at" all other states of the input sequence while generating the representation for any given token. In other words, the encoder itself is already attending to all tokens (left and right) when encoding each token, so the resulting representations will already capture the context-dependent meanings of words. The decoder will also apply self-attention as it generates tokens by looking at and attending to the previously generated tokens. At the decoder, the self-attention component happens to be called *masked* self-attention because a mask is applied to prevent the decoder from looking "ahead", i.e. looking at words that haven't been generated yet. And how exactly would that be possible in the first place? It isn't during inference (when you're using the model to actually generate words), but it is during training, in which case - because supervised training is used - there are reference responses available to the decoder. So, peeking ahead is indeed possible during training, hence the importance of the mask.

In addition to the self-attention part, the decoder also includes a decoder-encoder attention layer similar to that of seq2seq models.

In summary, the encoder attends to every token in the input sequence, gathering context, before generating representations for each token. And, on the decoder side, the same happens as tokens are generated, with "every token" referring to the previously generated tokens.

Let's now look at how self-attention is formally implemented. The original paper describes the attention function as follows:

The attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Let's break this down. The concepts of query and key-value pairs are analogous to those of retrieval systems and can be intuitively understood as follows:

- **Query**: information to be searched in the system. For example, when you search for "cats" on a database (say, representing Youtube videos), "cats" is your query.
- **Key**: the set of keys representing entries in the database (for example, on Youtube, these could be things like the video title, description, tags, etc).
- **Value**: the information that is retrieved from the keys that match the query.

Intuitively, the attention function is comparing the current token being processed (the query token) to the other tokens (the keys). It then gets weights (scores) for the values based on how good of a match we get between query and keys (i.e., how relevant are those other tokens to the current token). Then we apply those weights to the values and sum them up to generate the output. We repeat that for every token and concatenate all results into one final output vector. This vector will encapsulate a representation of the input tokens along with a measure of relevance between each pair of tokens.

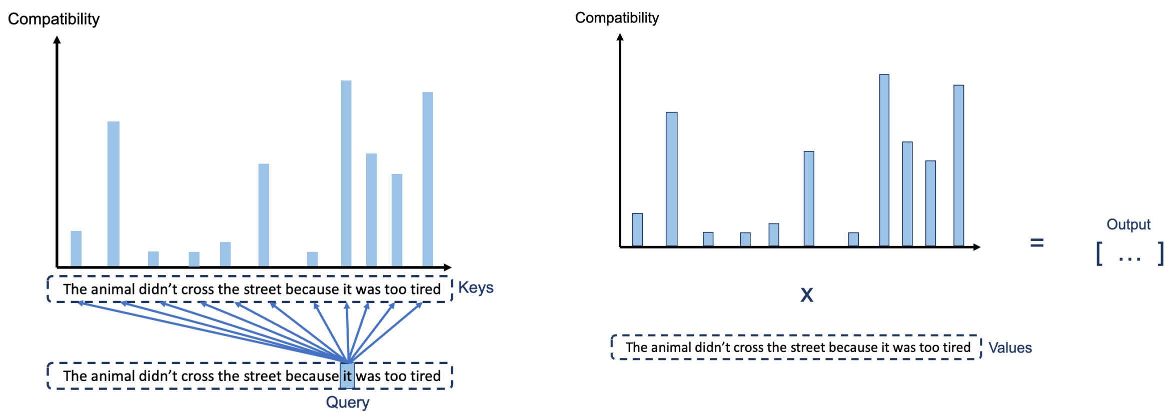

Compatibility
The animal didn't cross the street because it was too tired
Keys
The animal didn't cross the street because it was too tired
Query
Compatibility
X
The animal didn't cross the street because it was too tired
Values
=
Output
[…]

Illustration of Query, Key, and Value concepts

Another way to develop an intuition around this, especially if you're a programmer, is to compare the attention function to a dictionary lookup. Imagine a dictionary that maps tokens to some value:

```
dict = {                  lookup(dict, bank) =
    on:     v0,               v0 * 0.04 +
    the:    v1,               v1 * 0.04 +
    river:  v2,               v2 * 0.30 +
    bank:   v3,               v3 * 0.60 +
    they:   v4,               v4 * 0.02 +
    [MASK]: v5,}              v5 * 0.0
```

Comparing attention function to a soft dictionary lookup (source: [Transfer learning and Transformer models (ML Tech Talks)](#))

Now let's say we're looking up the word bank in the dictionary (i.e., the word bank becomes a *query*), whose value is *v3* in the preceding image. But, we would like to retrieve not only the value associated with *bank* (since we don't just want to look at each word in isolation) but instead we want to get a weighted sum of all the values of all tokens, where the weights reflect how similar each token is to *bank*. Obviously, the weight should be the largest for the word *bank* itself (and that's 0.60 in the preceding image), and somewhat large for *river* as it influences the meaning of bank a lot (0.30 in the preceding image). Other tokens will also get some non-zero score (except for the special *[MASK]* token which marks the end). The lookup function will thus return something like what is shown on the right.

And how are the values for query, key, and value vectors obtained? By multiplying the encoded representation of the token (X) by Q, K, and V matrices respectively. And how are these matrices constructed and their values obtained? They are learned by the model during training. In other words, they are model *parameters*. The attention scores (weights) are obtained through a dot product between the keys and the query vectors, which just happens to be a good way of measuring the similarity between vectors.

Once we have Q, K, V, and attention scores, the output of the self-attention layer for each query is the sum of values weighted by their attention score. The following image illustrates the usage of these parameters:

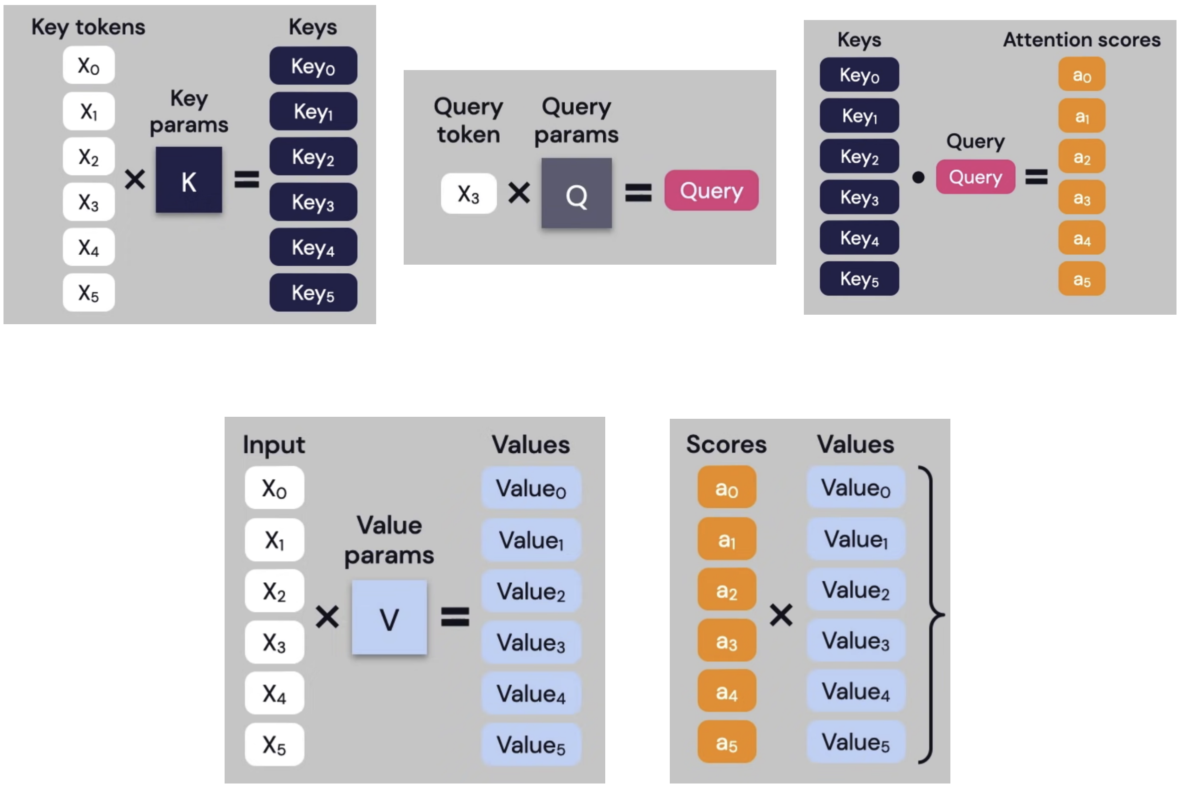


Illustration of Q, K, and V parameters (source: [Transfer learning and Transformer models (ML Tech Talks)]())

Note that X here, the encoded representation of the token, is a *non-contextual embedding* representation. Effectively, these matrices *transform* these "raw" token representations into the Query, Key, and Value vectors. And the motivation behind doing so is that they help obtain a better representation for computing the compatibility between two vectors. Intuitively, this helps tokens attend to other tokens, some being considered more relevant than others. This is crucial when dealing for instance with pronouns, as you want the pronouns to attend to the noun/subject it is referring to more so than to itself (since pronouns by themselves carry no meaning or context) or other non-relevant words.

In summary, the Q, K, and V calculations for a given token go like this:

1. Transfer the token vector from the input sequence (X) to a **query** in vector space: $Query = X \cdot Q$.
2. For every token in the sentence (Xi), transfer them to the vector space as **key vectors**: $Keys = Xi \cdot K$.
3. For each pair (**query**, **key**), obtain their compatibility strength by using the dot product `A=Q(dot)K`

4. Self-attention then generates `values`$=X\cdot$`V` and the embedding vectors for each token (X) by summing the values `Vi` multiplied by their respective score `Ai`. Each token contributes proportionally according to its compatibility strength to **Q**.

In reality, these calculations are done by not one but multiple attention "heads". Let's see why that is.

Multi-Headed Attention

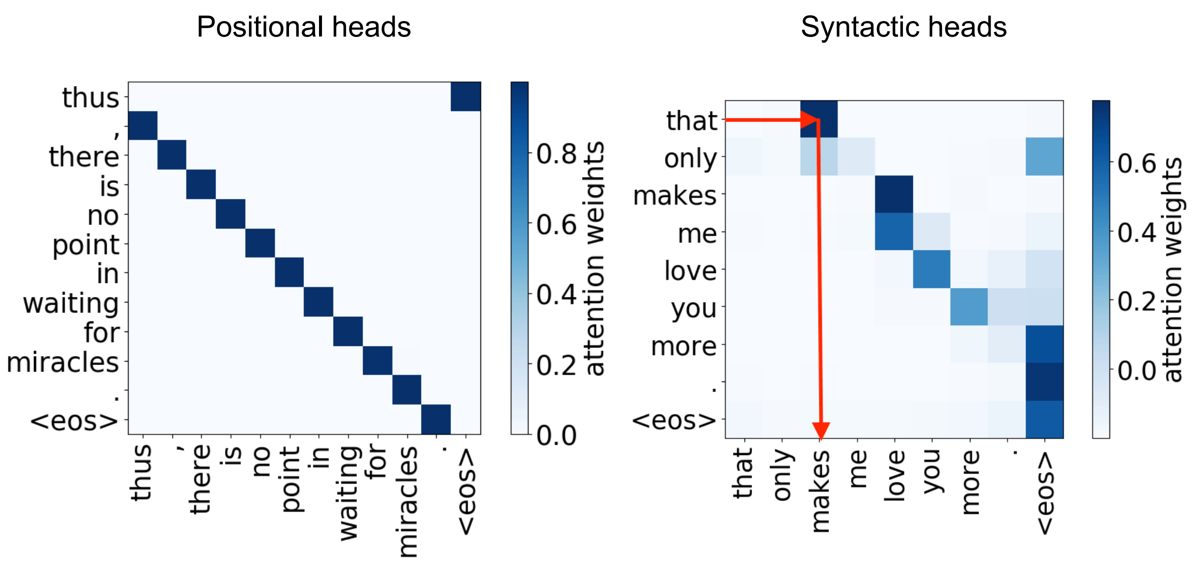
If you think about the way we use language, there are many perspectives one can take when trying to understand the meaning and role of a word in a sentence because there are several types of relationships we can draw between words. The form of the word also matters a lot, especially when generating them during decoding. For example, in some languages, the subject will define the verb inflection or the gender of the adjectives.

As an example, take the following sentence: "The animal didn't cross the street because it was __". What sort of words could come next here? As you think of words, try to think of the underlying relationships you're establishing between that word and the other words in the sentence. For instance, you may have thought of "scared", or "tired". Why? First, there's the structure of the sentence which hints at the fact that an adjective probably comes next. But not just any adjective. It has to be one that makes sense to associate with animals, because "it" is referring to the animal, and one that motivates the action (of *not* crossing the street). Or perhaps you thought "it" referred to "street", and maybe you thought of other adjectives such as "wide" or "large". Also, if this was written in another language, such as French or Portuguese, the gender of that adjective would need to agree with the gender of the word "animal" (or "street"). You get the idea: there are several things to think of at the same time here, and that is the motivation behind the concept of multi-headed attention.

Multi-headed attention allows the model to map words not into a single context vector space, but into multiple representations across different attention "sub-spaces". What these sub-spaces are in terms of what aspects of a relationship between words they represent we don't precisely know as these are learned by the model. We just know that there probably are several (the original paper uses 8) and we let the model figure them out during training. This is a common technique in machine learning, with a similar concept applied to clustering algorithms and convolutional networks for images (multiple channels).

That being said, in a 2019 paper titled Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned, the authors have identified some interpretable roles for these heads, which are: positional (focusing on a token's immediate neighbors), syntactic (tracking major syntactic relations in the sentence such as subject-verb, verb-object) and rare tokens (attending to least frequent tokens).

The following image illustrates what sort of "attending" the positional and syntactic heads are doing:



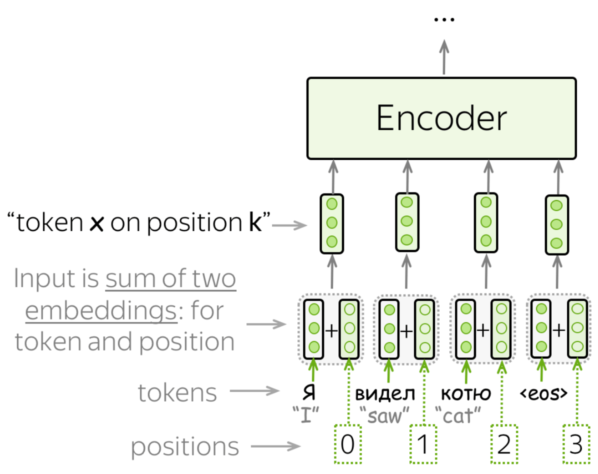
Source: Sequence to Sequence (seq2seq) and Attention, NLP course

The “rare tokens” head is a little more counter-intuitive and, in fact, it is believed to be a sign of overfitting (the model is trying to hang on to rare “clues”).

Positional Encoding

Since the Transformer model architecture doesn’t contain any recurrence, it does not know the order of input tokens. But the order clearly matters for language processing. For that reason, the input representation of a token is actually the sum of two embeddings: the word embedding (which is learned together with the training of the overall network) and the **positional embedding**. In the Transformer paper, the authors suggested a fixed positional encoding using a formula based on sines and cosine functions.

The following image illustrates the idea of adding positional encoding to the word embedding:

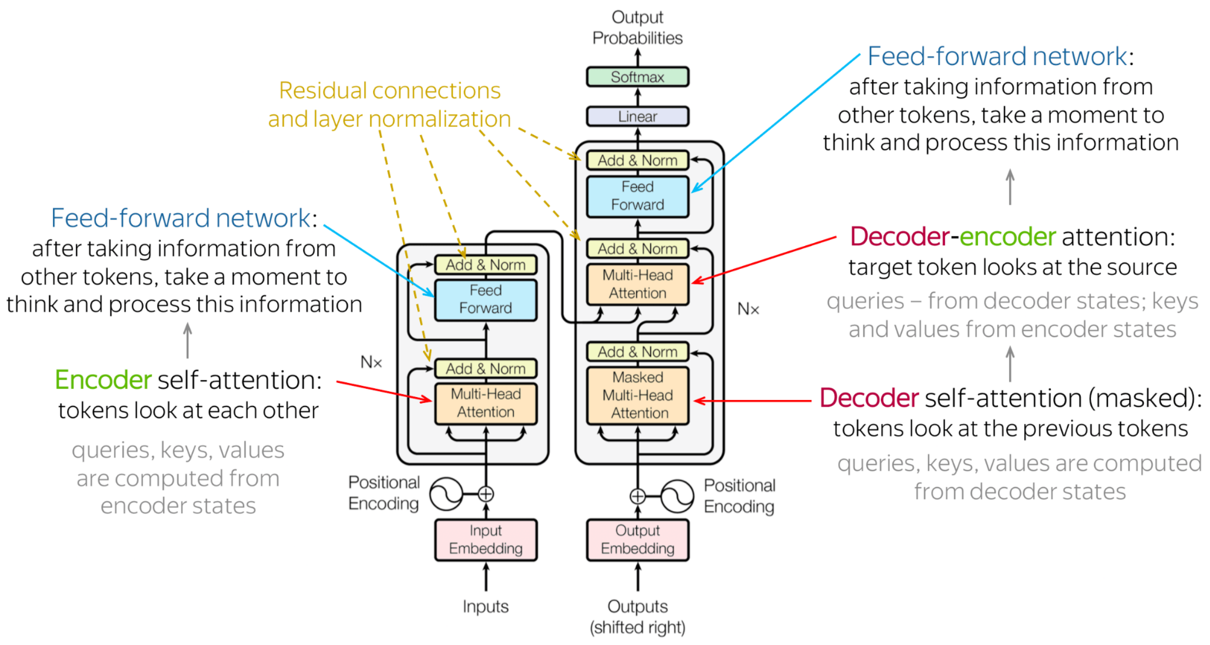


An illustration of positional encoding (source: Sequence to Sequence (seq2seq) and Attention, NLP course)

The intuition is that by adding these values to the token embeddings, we obtain more meaningful distances between the embedding vectors once they're projected into Q, K, and V vector spaces. This makes sense because where the words are relative to each other in a sentence does impact their meaning a lot and therefore should contribute to the embedded representations of the words.

Bringing it Together: The Transformer Model Architecture

With an understanding of the building blocks of the Transformer model, we can now look at the model architecture:


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Output Embedding
Outputs (shifted right)
Positional Encoding
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Input Embedding
Inputs
Residual connections and layer normalization
Feed-forward network: after taking information from other tokens, take a moment to think and process this information
Encoder self-attention: tokens look at each other
queries, keys, values are computed from encoder states
Decoder-encoder attention: target token looks at the source
queries – from decoder states; keys and values from encoder states
Decoder self-attention (masked): tokens look at the previous tokens
queries, keys, values are computed from decoder states

Transformer model architecture, annotated (source: [Sequence to Sequence (seq2seq) and Attention, NLP course](#))

The encoding and decoding components are actually stacks of six encoders and six decoders, respectively. Each layer in the stack is structurally similar to all other layers, but their internal weights are different.

The Encoder Side
The first encoder layer will take the word embedding (which includes the positional encoding) as input. Each subsequent encoder will take the previous encoder's output as its input. Note that this is not recurrence: each encoder takes the output of the encoder immediately below, not from itself.

Each encoder consists of two layers: a self-attention layer, which works by calculating Q, K, and V vectors as discussed in a previous section, and a feed-forward neural network layer. The latter applies to each position separately and identically (though the parameters are different from layer to layer). And here there's one key property of Transformer models worth highlighting: each input token flows independently through its own path in the feed-forward neural network and therefore they can be processed in parallel. This allows for a substantial improvement in training efficiency when compared to recurrent or convolutional neural networks.

The purpose of the feed-forward neural network layer in the encoder is to further process the output from the self-attention mechanism. Specifically, it applies a position-wise, non-linear transformation to the attention output. This layer consists of two linear transformations (dense layers) with a non-linear activation function, usually ReLU (Rectified Linear Unit), in between. The first linear transformation projects the input to a higher-dimensional space, while the second one projects it back to the original dimension. Put it simply, this layer adds an additional level of complexity and expressiveness to the encoding process, enabling the model to capture more complex patterns and relationships in the input data.

One final step in the encoder is the layer normalization step ("Add & Norm"). This is a technique that has been found (in a 2016 paper titled Layer Normalization) to reduce the training time in feed-forward neural networks by normalizing the summed input given to "neurons" (the nodes in the network) on each training case.

One final detail about the encoder is that it applies a dropout regularization step. This is one of several regularization techniques that can be used to prevent overfitting, which is when the model "memorizes" the training data and doesn't generalize well.

The Decoder Side
Each decoder also consists of self-attention and feed-forward neural network layers, plus an additional in-between layer which is the encoder-decoder attention. The self-attention layer attends to previously generated tokens (masking future positions during training) in the decoder itself, while the encoder-decoder attention attends to the input tokens (by using the encoder's K and V vectors).

The decoder's feed-forward neural network is ultimately generating a vector that represents (in some low dimensional space) the target tokens with their relative weights. A final layer, consisting of a fully connected neural network (linear layer) plus a softmax function, then maps that representation to a higher dimensional one which consists of the probabilistic distribution over all the target tokens of the target vocabulary (e.g. the English language). It, therefore, outputs a vector with all tokens in the target vocabulary followed by a number between 0 and 1, such that they all add up to 1 (they represent probabilities). The token with the highest probability (or sometimes another token within the top-K probabilities, as we'll see later when talking about large language models) is then chosen as the output.

Building a Transformer Model

If you want to get hands-on in building a transformer model and understand how each code piece relates to each of the Transformer building blocks, check out this Jupyter Notebook: The Annotated Transformer.

Large Language Models (LLMs)

A language model can be very simply defined as follows: given any sequence of words, a language model assigns a probability to each word in the sequence. Language models produce probability distributions over sequences of words.

If you think about it, that's all that is necessary to model human language. If you're translating text, you need a model that scores the candidate translations properly. In speech recognition, you need a model that scores the candidate utterances pointing out which ones are more likely to have been spoken. In natural language generation, you need to keep assigning probable word candidates in a way that the generated text maintains coherence. Now, obviously, assigning the right probabilities is the difficult part. Up until Transformer models were invented (in 2017), recurrent neural networks and sequence-to-sequence models were the best we could do. Their drawbacks were addressed by the Transformer models.

Transformers trained on large corpora of data, with billions of trainable parameters, turned out to be better than anyone expected at language modeling. These models, which we refer to as **large language models** (**LLMs**), have demonstrated impressive results on a wide variety of natural language processing tasks, including emergent abilities. LLMs such as LaMDA and GPT are based on a modified, decoder-only version of a Transformer.

Let's dive deeper into how they came to be and how they work.

Training Transformers using Self-Supervised Learning

Self-supervised learning (SSL) is a paradigm of machine learning that requires no human-annotated labels for model training, which means we can use a dataset consisting entirely of unlabelled data samples. However, this is not quite unsupervised learning. An SSL pipeline will generate labels automatically during the first stage, and then use the labels for supervised learning in the second and later stages. At the end of the day, SSL belongs to the category of supervised learning. But instead of human labeling, the model itself implicitly extracts supervisory signals such as correlations, metadata embedded in the data, or domain knowledge present in the input to generate labels.

Although there's no formal definition of LLMs, we can say that they consist of large (typically billions of weights or more) neural networks (typically transformers) trained on large quantities of unlabelled text using self-supervised learning. In most cases, a transformer is trained to maximize the probability assigned to the next word seen in the training data, given the previous context. An LLM may also use a bidirectional transformer (as in the example of BERT) to assign a probability distribution over words given access to both preceding and following context. The act of training a language model on text corpora is commonly referred to as *pre-training*. This is to differentiate it from the actual training of the model on specific tasks or for specific purposes. Pre-training is only concerned with getting the model to "understand" language. However, these two terms are often used interchangeably.

Though LLMs are trained on "simple tasks" such as predicting the next word in a sentence, with sufficient training data and model size (in terms of parameters) these models seem to capture much of the syntax and semantics of the human language and “memorize” many facts as a side effect.

Emergent Abilities

One impressive characteristic of LLMs is that these models develop emergent abilities through scaling. Emergent abilities are defined as capabilities that are not present in simpler (smaller) models, and which were not explicitly designed into the model, but are present in larger models. Example emergent abilities include multi-step arithmetic, taking exams, chain-of-thought prompting (more on that later), unscrambling a word’s letters, etc.

Transfer Learning and Foundation Models

Many natural language processing (NLP) tasks require common knowledge about language. For example, knowledge of grammar and the difference between verbs and nouns or adjectives, as well as their relationships. No matter what NLP task we’re trying to produce, whether it’s sentiment analysis, question answering, or translation, knowing how to model and express language is a base knowledge that gets applied in the same way across all different tasks.

Transfer learning is a machine learning concept and method where we reuse a pre-trained model, with the base knowledge that we need, and build on it to specialize it to a new task. The most common paradigm for applying transfer learning is pre-training followed by fine-tuning for any target task. This is why language models that are not specialized to any NLP tasks but which have simply been trained (in a self-supervised fashion) on large corpora of text are referred to as pre-trained models. Fine-tuning is then the process of training these models further (but not from scratch) to perform a target task well. The following image illustrates a practical example of this:

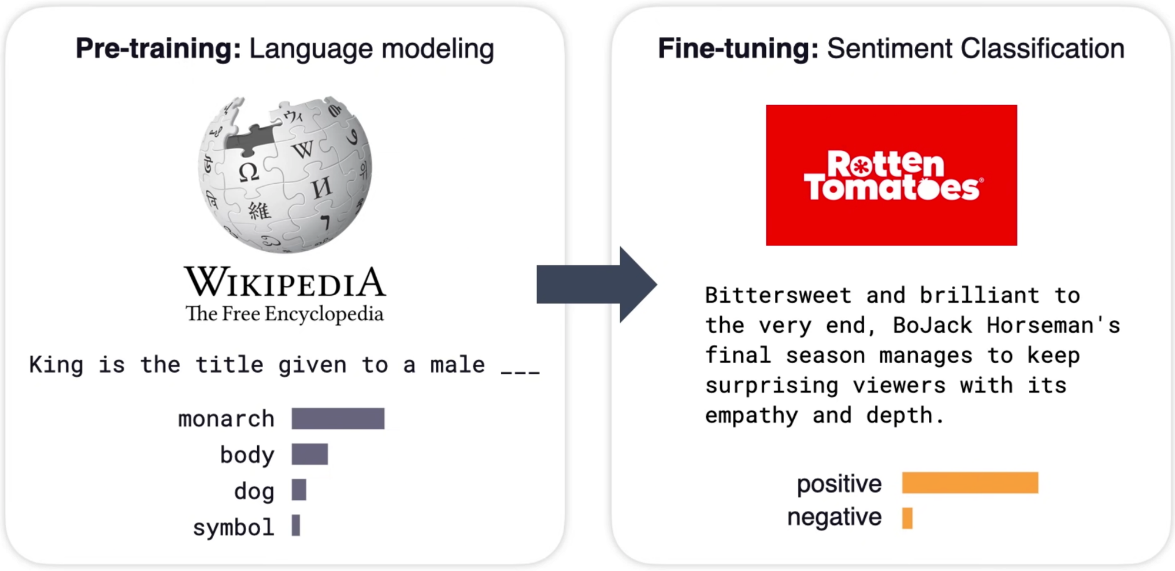

Pre-training: Language modeling
Wikipedia
The Free Encyclopedia
King is the title given to a male ___
monarch
body
dog
symbol
Fine-tuning: Sentiment Classification
Rotten Tomatoes
Bittersweet and brilliant to the very end, BoJack Horseman's final season manages to keep surprising viewers with its empathy and depth.
positive
negative

Illustration of transfer learning (source: [Transfer learning and Transformer models (ML Tech Talks)](#))

The Stanford Institute for Human-Centered Artificial Intelligence's (HAI) Center for Research on Foundation Models (CRFM) coined the term foundation model to refer to "any model that is trained on broad data (generally using self-supervision at scale) that can be adapted (e.g., fine-tuned) to a wide range of downstream tasks". Therefore, these pre-trained models can also be referred to as foundation models.

A Brief History of LLMs

In 2018, BERT was developed by Google by training a transformer model on a combination of BookCorpus (a dataset consisting of over ten thousand unpublished books scraped from the Internet) and English Wikipedia, totaling around 3.3 billion words. In the same year, OpenAI's GPT (generative pre-trained transformer) was trained on BookCorpus, which amounts to 985 million words.

In February 2019, OpenAI launched GPT-2 as a "direct scale-up" of GPT, with a ten-fold increase in both its parameter count (to 1.5 billion parameters) and the size of its training dataset (to 8 million webpages). Also in 2019, OpenAI transitioned from a nonprofit to a "capped" for-profit company.

In June 2020, OpenAI released GPT-3, which used a decoder-only transformer architecture with a 2048-token-long context window and a size of 175 billion parameters. Sixty percent of the weighted pre-training dataset for GPT-3 comes from a filtered version of Common Crawl consisting of 410 billion byte-pair-encoded tokens. Other sources are 19 billion tokens from WebText2 representing 22% of the weighted total, 12 billion tokens from Books1 representing 8%, 55 billion tokens from Books2 representing 8%, and 3 billion tokens from Wikipedia representing 3%. GPT-3 was trained on hundreds of billions of words and is also capable of coding in CSS, JSX, and Python, among others.

In January 2022, OpenAI started using reinforcement learning from human feedback (RLHF) to provide demonstrations of desired model behavior and rank several outputs from the model. This data was used to fine-tune GPT-3 and create InstructGPT, a model that was much better at following instructions than GPT-3, while also making up facts less often and showing small decreases in toxic output generation.

Also in January 2022, Google published LaMDA (Language Models for Dialog Applications), a family of Transformer-based neural language models specialized for dialog, pre-trained on 1.56T words of public dialog data and web text.

In March 2022, OpenAI released GPT-3.5 and Codex, an API for generating code.

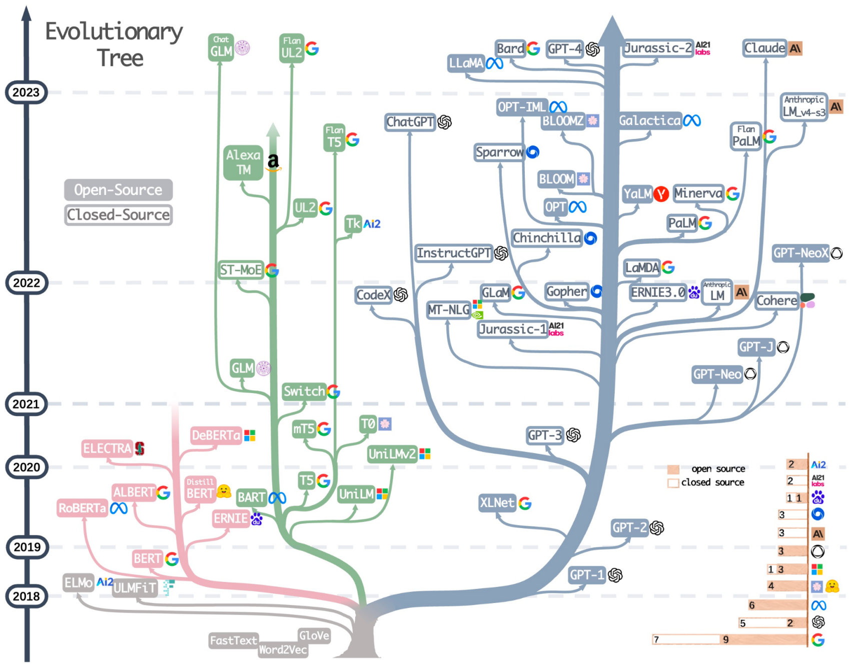
In April 2022, Google trained PaLM (Pathways Language Model), a 540-billion parameter, densely activated, Transformer language model. This model was trained on a new ML system, called the Pathways system, which enables highly efficient training on specialized hardware. PaLM demonstrated strong capabilities in multilingual tasks and source code generation.

In November 2022, OpenAI released ChatGPT, which is a fine-tuned model in the GPT-3.5 series. ChatGPT set a record for the fastest web application to reach 100 million users.

In March 2023, OpenAI released GPT-4, which introduced multimodality (text + image), greater alignment (less likely to produce harmful content), and a better understanding of nuanced instructions. OpenAI has not released details about the model, including (but not limited to) model size, architecture, hardware, dataset construction, training method, or scale. OpenAI did mention scaling optimizations to make training more efficient.

Also in early 2023, Google, Deepmind, Meta, Stanford, and others have released other large language models such as LLaMA, Alpaca, and Med-PaLM 2, and continue to drive innovation in large language models and generative AI.

This is an evolving landscape, however, and there have been several more models, model variants, and companies creating these models than the ones listed. The following image displays an evolutionary tree of models created since 2018:



Evolutionary tree of LLMs (source: LLMs Practical Guide, Git repository)

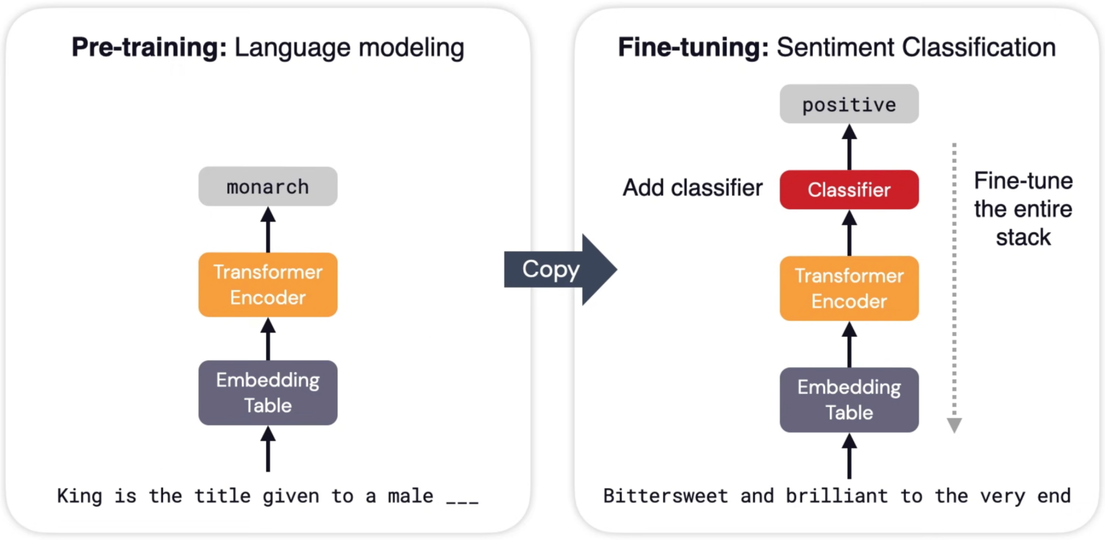
Tuning, Prompt tuning, Fine-tuning, Parameter-Efficient Tuning (PET)

Since 2020, LLMs have become capable enough that re-training a model with additional task-specific training was no longer necessary. These models became “customizable” through prompt design or by adding a small learnable layer, either of which doesn’t involve retraining the model. This has created some confusion around what it means to “tune” a model, so let’s differentiate these terms.

Fine-tuning

Fine-tuning is the practice of modifying an existing pre-trained language model by training it (in a supervised fashion) on a specific task or domain-specific dataset. This generally involves updating the base model’s parameters and is expected to improve performance on the target task.

For example, we can fine-tune a pre-trained language model to perform sentiment classification on movie reviews. We can achieve this by adding a classifier layer on top of the base model (which can also be referred to as the foundation model). This classifier layer may be a simple linear transformation layer, which will add new trainable parameters to the stack. Then, we train this combined model on labeled data which will fine-tune the entire stack, adjusting the parameters of the base model. The following image illustrates this scenario:


Pre-training: Language modeling
monarch
Transformer Encoder
Embedding Table
King is the title given to a male ___
Copy
Fine-tuning: Sentiment Classification
positive
Add classifier
Classifier
Transformer Encoder
Embedding Table
Fine-tune the entire stack
Bittersweet and brilliant to the very end

Example of pre-training followed by fine-tuning to perform sentiment classification (source: [Transfer learning and Transformer models (ML Tech Talks)](#))

It's worth noting that the fine-tuning process adjusts the original parameters of the pre-trained models. However, it's not as expensive (computationally speaking) as the actual pre-training of the model as the weights in this case are not randomly initialized but are already starting from a "good" place thanks to pre-training. That's why it's called "fine"-tuning, as these are minor adjustments only.

For hands-on experience fine-tuning a BERT-based language model, check out [this Colab notebook](#).

Prompt tuning (vs Prompt Design)

One way to improve the model's performance on certain tasks without updating the model's parameters is through prompt design. With prompt design, you "prime" the model with one or more examples of problem-solution pairs before giving it the "real" prompt. "One-shot prompting", "few-shot prompting", or "X-shot prompting" all refer to the number of examples that are provided within each prompt.

An example of one-shot prompting to adapt the model to analyze the sentiment of movie reviews would be the following:

```
Review: This movie stinks.
Sentiment: negative
```

```
Review: This movie is fantastic!
Sentiment:
```

Few-shot performance has been shown to achieve competitive results on NLP tasks.

In zero-shot prompting, no examples are provided. Instead, you simply try to steer the model to give the right answer by engineering the prompt in the right way. For example: "*The sentiment associated with the movie review 'This movie is fantastic!' is*".

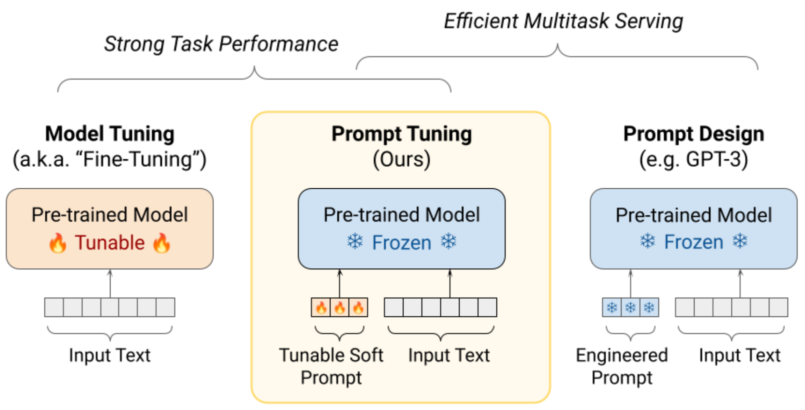
Prompt design doesn’t involve any model training or tuning and is therefore the cheapest and easiest way of influencing model responses.

What about prompt tuning?

Prompt tuning is one type of parameter-efficient tuning (PET). PET refers to tuning techniques that don’t involve updating the base model’s parameters but instead introduces a new set of parameters connecting the final layer of the base language model to the output of the downstream task (i.e., the creation of an “adapter layer”, separate from the base model). This is different from fine-tuning in that the base model remains “frozen”, i.e., its parameters are not trainable. Only the adapter layer, which is a lot smaller, is trainable.

The specific prompt tuning technique pioneered by Google Research introduces tunable “soft” prompts. Just like engineered text prompts, these soft prompts are concatenated to the input, but the difference is that the tokens that comprise the (soft) prompt are learnable. So, instead of manually trying to design the right prompt to obtain the best results, you have machine learning do the job for you and optimize the prompt over a training dataset. In addition, while discrete text prompts are typically limited to under 50 examples due to constraints on model input length, tunable soft prompts have no such limitation and can learn from datasets containing thousands or millions of examples. You can think of this technique as adding one large (and learnable) prompt prefix to each prompt.

The following image illustrates the differences between fine-tuning, prompt tuning, and prompt design:



Fine-tuning vs prompt tuning vs prompt design (source: [Guiding Frozen Language Models with Learned Soft Prompts, Blog article]())

As model size increases, prompt tuning catches up to the performance level of model fine-tuning. Intuitively, the larger the pre-trained model, the less of a “push” it needs to perform a specific task, hence the more capable it is of being adapted in a parameter-efficient way.

Other related techniques have also emerged, being often referred to as parameter-efficient fine-tuning (PEFT) techniques, creating some confusion around the terminology. What they all have in common is that they all include a tunable component and they all keep the base language model frozen (not trainable). Examples of such techniques include LoRA (Low-Rank Adaptation of large language models) and P-Tuning v2 (also a prompt tuning technique).

Chain-of-Thought Prompting

Chain-of-thought (CoT) prompting is a prompt engineering technique proposed by Google researchers in 2022. It improves the reasoning ability of LLMs in multi-step problems by prompting them to generate the various intermediate steps that lead to their answers.

For example, given the question:

“The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?”

A CoT prompt that induces the LLM to answer with the reasoning steps (the chain of thought), such as “*A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.*”, is likely to improve the responses of the model on similar questions.

When applied to PaLM, CoT prompting allowed it to perform comparably with task-specific fine-tuned models on several tasks, even setting a new state of the art at the time on the GSM8K mathematical reasoning benchmark.

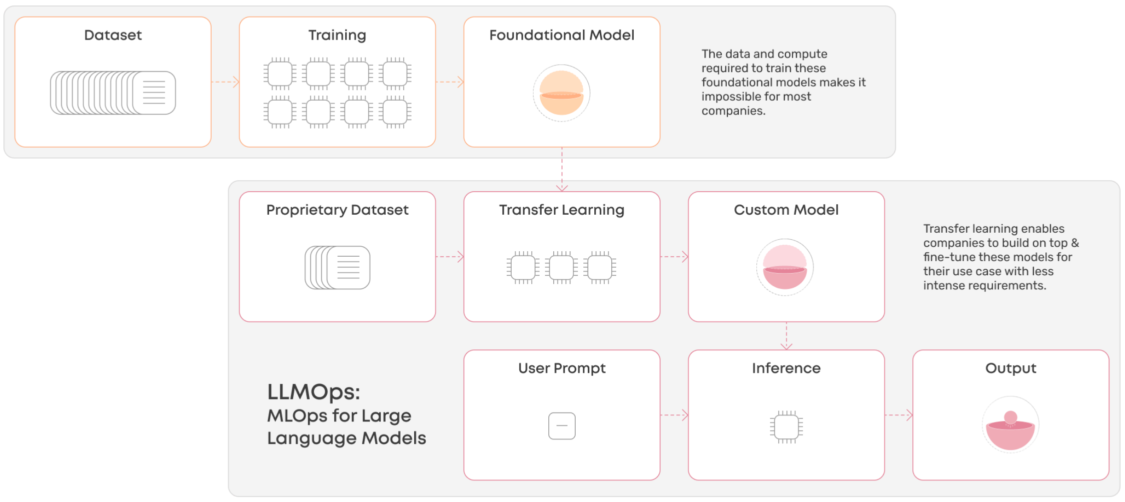
CoT prompting works better with larger and more powerful language models and can also help improve interpretability of responses.

LLMOps

MLOps (Machine Learning Operations) comprises techniques to streamline the process of taking machine learning models to production, monitoring, and maintaining them. There are, however, some peculiarities about LLMs that must be taken into account when it comes to operationalizing these models. For example, pre-training a large language model requires a large number of resources and can take a long time. Therefore, retraining these models on a regular basis is not realistic. In addition, with the parameter-efficient tuning techniques we previously discussed, that is not necessary (in most cases) in the first place.

The concept of LLMOps as a narrower definition of MLOps has thus emerged to differentiate the way LLMs are operationalized from the way traditional ML models are. While there’s no formal definition for it, currently LLMOps typically refers to the application of transfer learning to existing, pre-trained foundation models, with the creation of adapter layers trained on proprietary datasets, possibly followed by templated prompts or prompt prefixes.

The following image illustrates this idea:


Dataset
Training
Foundational Model
The data and compute required to train these foundational models makes it impossible for most companies.
Proprietary Dataset
Transfer Learning
Custom Model
Transfer learning enables companies to build on top & fine-tune these models for their use case with less intense requirements.
LLMOps:
MLOps for Large Language Models
User Prompt
Inference
Output

An illustration of LLMOps (source: LLMOps: MLOps for Large Language Models)

The LLMOps landscape includes foundation model providers (such as OpenAI, Google Cloud, Meta, Hugging Face, and others), ML- and LLM-optimized platforms (such as Google Cloud's Vertex AI with its Cloud TPUs), but also tools and frameworks such as MakerSuite, Gen App Builder, and Langchain that help users build applications powered by LLMs, sometimes chaining LLMs together.

References

Word Embedding: Basics, Medium article

Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)

GitHub - tensorflow/nmt: TensorFlow Neural Machine Translation Tutorial

The Illustrated Transformer

Seq2seq and Attention

Sequence to Sequence Learning with Neural Networks

Neural Machine Translation by Jointly Learning to Align and Translate

Attention is all you need

GPT-4 Technical Report

LaMDA: Language Models for Dialog Applications

The Power of Scale for Parameter-Efficient Prompt Tuning

LoRA: Low-Rank Adaptation of Large Language Models

P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks

Additional Resources

The Annotated Transformer

LLM Papers by LifeArchitect

The ChatGPT Prompt Book

LLMs Practical Guide