

Unit 9 – Language Models

Convolutional networks provide a specialized architecture that is well suited to solving a variety of problems on input data that can be arranged as an array, and under the assumption that the spatial arrangement of the data is meaningful (though possibly at different scales). Image-related tasks fit this category well, and as a result convnets are extensively used on images.

We may wonder whether we can apply the same kind of neural architecture to the problem of understanding and processing text. For instance, this page of text can be thought of as a grid in 2D inside of which we find an arrangement of characters that for an observer capable of recognizing them and organizing them into words, contains ***meaningful information***.

Why not treat a page of text such as this one as just another 2D array of data corresponding to rows and columns of characters, and use a suitably built convolutional network to solve interesting problems similar to those we know we can solve on regular images? Indeed, it is possible to use a convolutional network to carry out tasks such as ***text classification***, and ***sentiment analysis*** (Conneau et al., 2016). The network's architecture is fairly similar to standard image processing convnets such as VGG and ResNet.

Even though a ConvNet performs well at very specific language tasks, common language processing problems such as ***text generation***, ***summarization***, ***query understanding***, and ***language translation*** have the property that ***global context*** matters. This means that the information provided by ***individual features*** even at ***multiple scales*** should be influenced and transformed by content present in the entire input. ConvNets rely on spatial locality and contiguity for feature detection and analysis, and as such, are not the best architecture to handle the kinds of problems described above.

To fully understand how current language models work, we first need to take a look at how ***text is represented and manipulated*** within language models, and at the kinds of problems that motivate the current architectures for dealing with language.

Text representation in LLMs

The unit of information in text is rarely a single character. A convnet working on input text would somehow have to ***learn kernels to identify each individual grouping of characters that has meaning***. While this may be feasible, research in natural language understanding already provided a tool for solving this particular problem.

Instead of working on individual characters, a first step of processing converts input text into a ***sequence of tokens*** – these consist of groups of characters that may be identifiable as individual words but often are not. The process of turning input text into a sequence of tokens is called ***tokenization***, and there are many variations on the details of how it is done, but you should think about it as breaking text into individual, unique parts that have more meaning than individual characters and are helpful in processing language.

GPT-4o & GPT-4o mini GPT-3.5 & GPT-4 **GPT-3 (Legacy)**

This is an example of how text is broken into tokens by a tokenizer, it identifies units of meaning that are relevant in the context of language processing. Many different ways exist to break text into tokens.

Clear Show example

Tokens	Characters
41	209

Example of how text is broken into **tokens** for input into a language model, each **token** is represented as a unique integer defined within a **vocabulary** (image generated from the Open AI online tokenizer)

From the above we have:

- Text is pre-processed to extract **a sequence of tokens**. The **order** of these tokens matters greatly and any model we build to process language must be able to use the information provided by the ordering of tokens in the input text.
- The **tokenizer** defines a **vocabulary** of unique tokens that can be used for language processing for current language models (2025) this is in the order of 100K – 200K unique tokens, and includes tokens to handle technical terms such as what you would find in Math, Science, and other specialized fields.
- Because words can be broken up into components, unusual or foreign words, things like names and even grammatical errors will be parsed into some sequence of known tokens.
- These tokens will then be processed by the language model whose job it is to make sense of the sequence in order to perform some specific task.

Context is everything

The key property of language that makes language processing challenging (with the types of architectures we know up to this point) is that the **information provided** by individual tokens is to a very large degree dependent on the context – such context is provided by other parts of the text, some of which can be at a significant distance from the place a token occurs.

Convolutional networks are great at capturing spatial relationships in its input within contiguous, reasonably sized neighbourhoods, but they struggle to capture the complex interactions between words that may be far apart but clearly affect each other's meaning.

Similarly, text passages can have any length – a language model may be required to work on a few words, a paragraph, a page or an entire article or book (think, for example, about automatic summarization). Even worse, the context itself may not be present *in the input text* but may be something the model has to learn from *training data*. Simply put, a good language model will have *learned* about the *usual meaning* of *tokens*, and use the specific input it is provided to *enrich, modify, and refine* what it has learned through training in order to arrive at the correct interpretation of every token in the input sequence it has been given.

Consider the following, short sentences:

The fish could not go into the basket because *it* was too big

The fish could not go into the basket because *it* was too small

Both sentences have the same structure – if you were applying a pattern-matching kernel to them (as you would if you built a convnet to process text) they would appear very similar. The same *bolded* token *‘it’*, however, refers to very different things in each case.

The *context* here is not obvious, and it is not provided by the input text itself. It has to be *inferred* from prior knowledge about the relative sizes of fishes, baskets, and a bit of logic: if the fish is too small, it definitely fits inside the basket – so the basket must be *‘it’* in the second sentence. A good language model must be able to arrive at the *most likely meaning* for an ambiguous token, and it must be able to use information available anywhere within the input text, as well as what it has *learned* during training.

In what follows, the word *meaning* is used to describe the *changeable, context-dependent* information that an individual *token* may represent. It is *not fixed*, or *grounded*, rather, it is *arrived at* through careful processing of the context the token appears in. To illustrate this idea, consider the following example:

"At three o'clock precisely I was at Baker Street, but Holmes had not yet returned." (from The Adventures of Sherlock Holmes by Sir Arthur Conan Doyle)

Consider the word *‘street’* in the sentence above. We know from any dictionary that this particular word has a specific *ground-truth meaning*:

street (noun) - A public road in a city or town that has houses and buildings on one side or both sides.
(from the Oxford learner's dictionary)

However, placed within the context provided by the text around it, we could determine that besides describing a public road, the word refers to a *specific street* (Baker Street) on which there is a *particular house* (221B) wherein lives a *famous detective* (Sherlock Holmes) along with his friend and sidekick (Watson). This kind of *enriched meaning* is what we are after in order to build language models.

Mapping tokens to context-dependent meaning

If we want to have **tokens** that are able to capture rich context-dependent information, we first have to carefully consider the problem of **how to represent these tokens** in the first place. A very simple but (as it turns out) not very helpful solution would be to simply replace each token in the input text with its unique ID. So, for instance, the sentences above may look like so:

785	213	1591	17	89	341	22	1921	1521	56	777	4516	99
785	213	1591	17	89	341	22	1921	1521	56	777	4516	97

The token IDs are made up, but you can see what is happening – we converted a sequence of text into a sequence of integer IDs that correspond to the unique tokens in our vocabulary that represent the words in the input.

The similarity between both sentences is even more striking when we think about them as a sequence of token IDs. It shouldn't be surprising that a convolutional network will have a hard time figuring out that the token '777' in the two sentences above refers to different things.

We can not represent **tokens** in our language model with simple, unique IDs because we need to be able to distinguish between different possible meanings for the same token; and we need to be able to update a token's meaning to account for added context. Even within the same block of text a particular **token** may mean different things, and there is nowhere within a sequence of token IDs where we can **store** or **encode** such information. Therefore, we need a more flexible and powerful representation for individual tokens.

Representing tokens as vectors

In order for our model to be able to **refine** the meaning of specific tokens in a sequence, we must provide it with a representation that can be **transformed** in such a way that the transformed version is **richer in meaning** than the initial one. We can achieve this if we encode **tokens** as **vectors**. We specifically want to

- Have each **token** in our **vocabulary** be represented by a **unique vector** in a space of appropriate dimensionality (more on this shortly)
- Have the **direction of the vector** encode **meaning** in such a way that **proximity** with regard to a reasonable vector distance measure is related to similarity in meaning

A first attempt at encoding **tokens** as feature vectors would involve a simple and commonly used **one-hot encoding**: The length of the vectors is the same as the size of the **vocabulary**, and each token is represented by a vector that is **all zeroes except for the entry corresponding to the tokenID** which would be set to **1**.

This is an easy thing to do, but it doesn't really achieve what we want. A one-hot encoding for tokens as vectors has a few undesirable properties:

- Vectors are too long (recall we said current language models have vocabularies with over 100K tokens)
- The original vector's direction has nothing to do with meaning, it is simply an accident of what particular ID was assigned to each token. Proximity in vector space tells us nothing about similarity of meaning

Word embeddings

The current approach for handling **tokens** in language models is to learn **a word embedding** – this is a transformation from the space of **one-hot token-ID vectors** to a space with a **much lower dimensionality**. The transformation encodes **semantic information** as **vector direction** in the lower dimensionality space. It reduces the size of the vectors to something that is manageable in practice, while allowing the language model to transform these vectors to refine and enrich the meaning of each token as per our original intent.

If you have worked with dimensionality reduction before (for example, PCA in the context of classification) then you already have a good idea of how this process may work. A similar process to PCA can be used to create word embeddings for tokens. Global Vectors (GloVe, Pennington et al., 2014) compute the embedding from a **word-co-occurrence matrix** using a dimensionality reduction process similar in principle to PCA.

Current LLMs **learn the embedding** during the training process together with the parameters for the rest of the model. In this way the embedding itself is optimized for whatever task the language model is expected to solve. However the embedding was created, the form of the end result is the same:

- A word embedding matrix, with **one column** per **token**, the corresponding column represents the **vector** that encodes the **initial meaning of a token** within the context of the language model, training data-set, and task being solved.
- The dimensionality of the embedding is a parameter of the language model, and is much smaller than the size of the vocabulary.
- Vector direction in the space spanned by the embedding is related to **semantic content**, so **angle similarity or proximity** in this space imply **similarity of meaning**

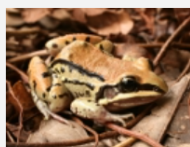
1. Nearest neighbors

The Euclidean distance (or cosine similarity) between two word vectors provides an effective method for measuring the linguistic or semantic similarity of the corresponding words. Sometimes, the nearest neighbors according to this metric reveal rare but relevant words that lie outside an average human's vocabulary. For example, here are the closest words to the target word *frog*:

0. *frog*
1. *frogs*
2. *toad*
3. *litoria*
4. *leptodactylidae*
5. *rana*
6. *lizard*
7. *eleutherodactylus*



3. *litoria*



4. *leptodactylidae*



5. *rana*



7. *eleutherodactylus*

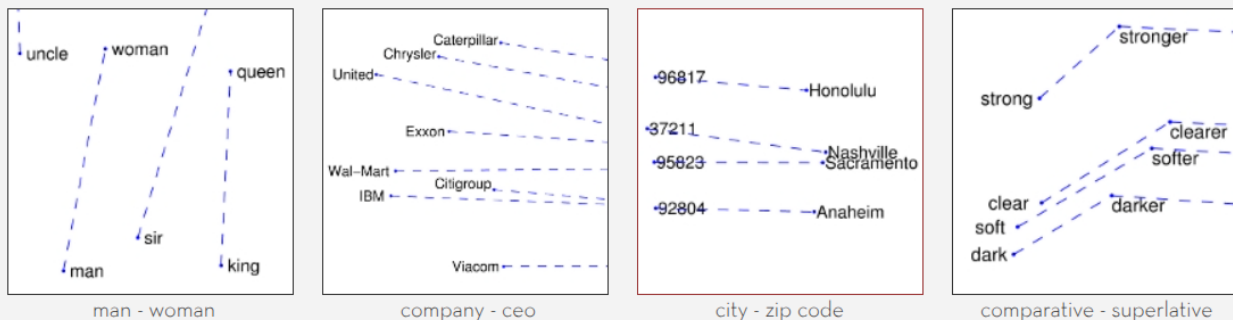
Within the embedding's subspace, proximity encodes **similarity of meaning**
(image from GloVe, by Pennington et al.)

- An emergent property of the above is that moving along a particular direction in the embedding space **modifies a token's meaning** in a predictable and stable way

2. Linear substructures

The similarity metrics used for nearest neighbor evaluations produce a single scalar that quantifies the relatedness of two words. This simplicity can be problematic since two given words almost always exhibit more intricate relationships than can be captured by a single number. For example, *man* may be regarded as similar to *woman* in that both words describe human beings; on the other hand, the two words are often considered opposites since they highlight a primary axis along which humans differ from one another.

In order to capture in a quantitative way the nuance necessary to distinguish *man* from *woman*, it is necessary for a model to associate more than a single number to the word pair. A natural and simple candidate for an enlarged set of discriminative numbers is the vector difference between the two word vectors. GloVe is designed in order that such vector differences capture as much as possible the meaning specified by the juxtaposition of two words.



The underlying concept that distinguishes *man* from *woman*, i.e. sex or gender, may be equivalently specified by various other word pairs, such as *king* and *queen* or *brother* and *sister*. To state this observation mathematically, we might expect that the vector differences *man* - *woman*, *king* - *queen*, and *brother* - *sister* might all be roughly equal. This property and other interesting patterns can be observed in the above set of visualizations.

(image from GloVe, by Pennington et al.)

Summary - what we have so far

- We will convert input text into a **sequence of tokens**, the ordering is critical for determining the meaning of individual tokens.
- **Tokens** will be represented by **unique vectors** in a **word embedding** that has been learned on the specific problem the language model is designed to address.
- Meaning (in the sense of context-dependent, refined semantic information) is encoded by **direction** in the space of the embedding.
- This allows a model to **enrich**, **refine**, and **infer meaning** by transforming the input vectors based on learned information, and the sequence of input tokens.

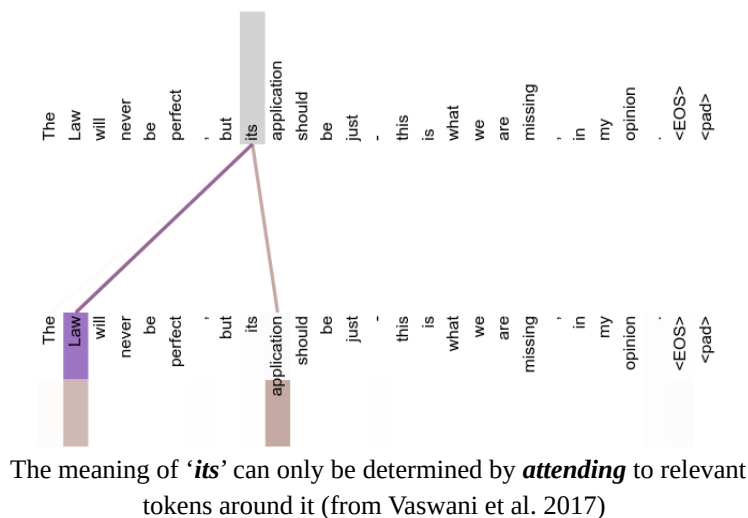
Large Language Models

With the above, we are ready to take a close look at the building blocks of current **Large Language Models (LLMs)**. Existing models are incredibly large (billions of learned parameters, hundreds of layers of processing) and perform amazing feats in terms of **understanding queries, text generation, summarization, automated translation**, and so on.

Consider the following example (from Vaswani et al. 2017, “Attention is all you need”):

“The Law will never be perfect, but **its** application should be just.”

We know each **token** in the sentence above will be represented by a vector in a suitably high-dimensional space in which direction represents **meaning** – so each token’s **initial vector** already encodes useful information – however, a word such as ‘**its**’ may mean different things, so whatever its initial direction in the word embedding, it will not be sufficient to tell a model how to interpret it in the specific sentence we are looking at.



What we need is a way to **transform** a **token’s initial vector** so that its meaning is **enriched and refined** based on the tokens around it. We know this is possible because of the structure of the embedding. The mechanism by which we will achieve this is called **attention**.

The process of **attention** involves:

- Given a particular **token** in a sequence (e.g. ‘**its**’ in the sentence above), figure out which other tokens are **relevant** (which tokens a language model should pay **attention** to) in order to **refine** its meaning.
- Determine how to use the **embeddings of the attended tokens** (e.g. ‘**Law**’ and ‘**application**’ in the example above) to **transform the embedding** of the original token to enrich its meaning.

- Produce a new **feature vector** that corresponds to the token's **refined semantic content** that includes information encoded by the '**Law**', and '**application**' embeddings. Those embeddings themselves could have undergone their own process of context-dependent refinement.

A **transformer** is a specific neural-net architecture designed to carry out the attention process as described above. There are three fundamental components to the transformer architecture whose names come from the field of information retrieval: **Queries (Q)**, **Keys (K)**, and **Values (V)**. These terms are borrowed from the field of **semantic search**, so let us consider a simple example of how they are defined in the context of a search problem, and then we shall see how they apply to the process happening inside a **transformer** network.

Suppose we are building an image search engine – it will allow a user to find images of interest. As a part of this search engine, a convolutional network will process images and generate **feature vectors** that somehow encode the image content.

A user would submit a request for images that contain certain objects or have particular properties. This request could be converted in a **vector of features** that describes what the user wants. This is the **query (Q)**. The search engine would then compare or match this query against the feature vector of each of the images in the database – each specific image's vector is a **key (K)** that can be used to determine how well that particular image matches the user's **query (Q)**. The search engine can then figure out how well individual images match the query, and return a sorted list with the relevant **values (V)** (the images themselves) to the user.

In the context of LLMs, and specifically for **transformers**:

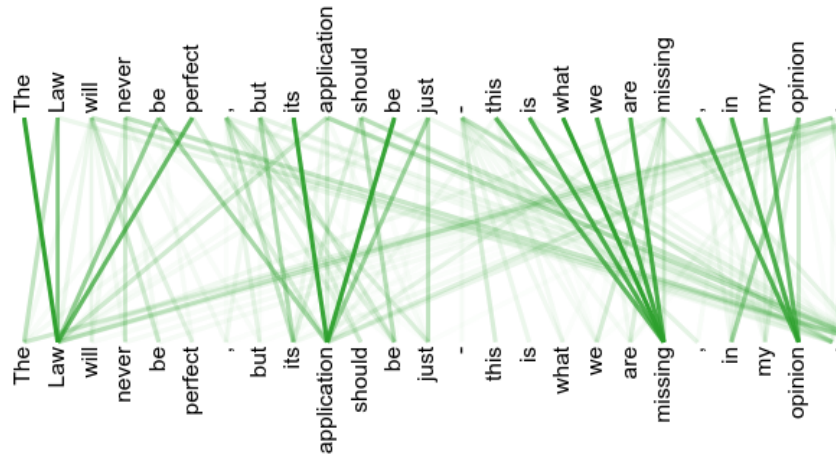
- The **queries (Q)** are linear combinations of the features in a token's feature vector. The transformer learns useful queries during training. They will correspond to patterns that, if found in the input, would meaningfully change the meaning of a token.
- The **keys (K)** are also linear combinations of the features in a token's feature vector. They correspond to patterns in a token's features that make them **relevant** to particular queries, and are also learned during training.
- The **values (V)** are (as you may expect) linear combinations of a token's features, where the weights determine **how much** a token's features will contribute toward updating the **semantic information** of other tokens in the input as a result of how well their **keys** and **queries** match.

Here's how everything is implemented:

First, the input tokens corresponding to the text we are providing the language model are organized into a matrix. If the input contains ***N* tokens** and each token is represented by an embedding vector with ***D* features**, the input matrix will be ***N* by *D* - *N* rows and *D* columns**.

The **output** of the transformer will be a new matrix of size ***N* by *D*** with feature vectors that correspond to the tokens after their meaning has been **transformed** as a result of the operations carried out inside the transformer. The **meaning** of each transformed vector depends on **all the tokens in the input** but the amount by which each input token affects the transformed output depends on how

relevant each input token is toward the meaning of the output one. The goal of the **attention mechanism** is to determine which tokens are relevant to which outputs, and by how much.



Example of **self attention** – the strength of the colour indicates how relevant each word is to the meaning of every other word in the sentence (from Vaswani et al. 2017)

Specifically, we want

$$\vec{y}_n = \sum_{i=1}^N a_{i,n} \vec{x}_i$$

where the **attention weights** are all positive, add up to 1, and depend on the relevance of each input vector \mathbf{x} to output vector \mathbf{y} . How these weights are determined is the key to how the transformer operates. The transformer's operation is fairly straightforward. It relies on learning three **weight matrices** used to compute the **keys**, **queries**, and **values**:

$$\mathbf{Q} = \mathbf{XW}_q$$

$$\mathbf{K} = \mathbf{XW}_k$$

$$\mathbf{V} = \mathbf{XW}_v$$

In the above, \mathbf{X} is the $N \times D$ matrix with one row per input token, each \mathbf{W} matrix is of size $D \times D$ and the resulting matrices \mathbf{Q} , \mathbf{K} , and \mathbf{V} are of size $N \times D$ and consist of linear combinations of the features in each of the input token vectors.

The **self attention weight** that determines how **relevant** input token i is to output token j is given by the dot product of $\mathbf{Q}(j,:)$ and $\mathbf{K}(i,:)$ - the corresponding rows in the query and key matrices for tokens i and j . The full set of **self attention weights** is given by

$$\mathbf{A} = \mathbf{QK}^T$$

which is an $N \times N$ matrix. Recall that the **self attention weights** for output j have to be all positive and add up to 1.0, this is achieved by processing the rows of matrix \mathbf{A} by passing each row through a **softmax** function.

$$\mathbf{A}' = \text{softmax}\left(\frac{1}{\sqrt{D}} \mathbf{A}\right)$$

$$\mathbf{A}'(i, j) = \frac{e^{\mathbf{A}(i, j)}}{\sqrt{D} \sum_{k=1}^D e^{\mathbf{A}(j, k)}}$$

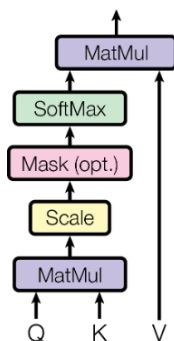
The scale factor with the square root of D ensures the output vectors have unit variance. Given the **scaled dot product self attention** weights in matrix \mathbf{A}' , the **refined** output vectors are obtained as

$$\mathbf{Y} = \mathbf{A}'\mathbf{V}$$

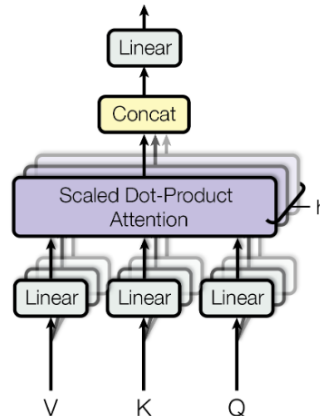
A module implementing the operations described above is called an **attention head**.

NOTE: We talk of the matrices \mathbf{Q} , \mathbf{K} , and \mathbf{V} as corresponding to **queries**, **keys**, and **values** in the context of information retrieval but it's important to remember that the learning process is not actually trying to learn matrices that can cleanly be taken to represent either of these things – the network will simply learn values for these matrices such that the **attention mechanism** is able to help the model carry out its task. Thinking of **queries**, **keys**, and **values** is just a convenient abstraction.

Scaled Dot-Product Attention



Multi-Head Attention



Single head, and multi-head attention modules (from Vaswani et al. 2017)
the masking operation allows specific tokens to be ignored by the attention mechanism and may be required for specific applications.

Multi-head attention

Similar to having a convolutional network learn a large number of different **kernels**, transformers benefit from learning different sets of **queries**, **keys**, and **values**. This means that instead of training a single attention head, we will train and run **multiple layers of attention** in parallel. Each **attention head** computes output vectors that are transformations of the input tokens relevant to the specific **queries** the head cares about. The different outputs have to be combined to produce the final output vectors.

This is done by first concatenating the outputs \mathbf{Y}_i of each attention head into a single, large matrix of size $N \times \mathbf{KD}$ where \mathbf{K} is the number of attention heads:

$$\mathbf{H} = [\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_k]$$

$$\mathbf{Y}_{mh} = \mathbf{H}\mathbf{W}_o$$

where \mathbf{Y}_{mh} contains the final output vectors from the **multi-head attention** block, and \mathbf{W}_o is a matrix that linearly combines the outputs of the different heads to produce the final output vectors. The output matrix also has to be learned during training. It is important to keep in mind that there is now a fairly large number of parameters that have to be learned:

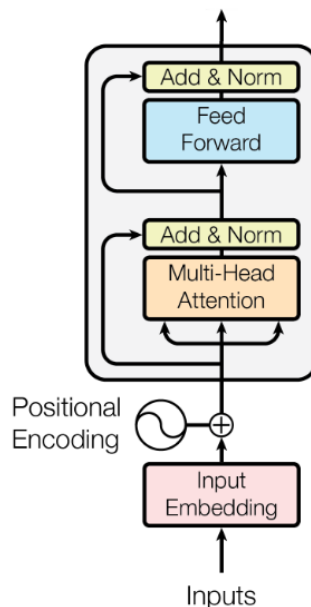
- We have $2 \times K$ matrices of size $D \times D$ for the **queries** and **keys** of the K attention heads
- We have K matrices of size $N \times D$ for the **values** of the K attention heads
- And we have 1 matrix of size $N \times KD$ for the **output** matrix of the multi-head attention module

This adds up very quickly, so for example, the GPT-3 model which by now is not considered particularly large clocks in at over 175 billion parameters. A good portion of these are in the many, many weight matrices that have to be learned for the multiple layers of multi-head attention modules that extract, refine, and transform the meaning of input tokens from their context.

The Transformer

The **transformer** is the fundamental block of processing in modern LLMs. It consists of a **multi-head attention** module, a **skip connection**, **batch normalization**, and a **fully-connected** neural net (usually two layer) possibly also with **skip connections**.

The input to the transformer block is a sequence of **tokens** that the LLM is processing. The **output** is a transformed set of **tokens** that forms the input to another transformer block – LLMs consist of stacks of transformers progressively refining and enriching the meaning of the input tokens until the final block outputs a sequence that contains the result LLM was trained to produce.



A *transformer* module consists of a multi-head attention block, followed by a fully-connected, feed-forward net, notice the skip connections, and the batch normalization layers (from Vaswani et al. 2017)

We have already seen what the **multi-head attention** block is doing, so let us consider the role of the **fully-connected** network that forms the second part of a transformer block. Multi-head attention ensures that the meaning of each **token** can be suitably **refined** by all the other **tokens** in the input – it produces a **non-linear** mixture of the features in the input’s token sequence that results from learned **attention patterns** useful to the model.

The **feed-forward** network then **processes each of the vectors output by the multi-head attention** block independently. The network typically has two layers. The first layer **projects the vectors to a higher dimensional space**, the second layer mixes and combines features in this high-dimensional representation to produce the final output vectors of the same dimensionality as the input to the transformer.

The **feed-forward network** takes an input embedding vector and updates each feature to account for interesting/useful patterns that span the entire vector. This non-linear transformation of features plays a significant role in the refinement process. There is some evidence to suggest that the **feed-forward** net can store factual information from the training data, and that such information can be encoded into the embeddings as they pass through the fully-connected block.

Transformers can be stacked together to build language models. The **encoder-only model** which was used in the original GPT works in this way:

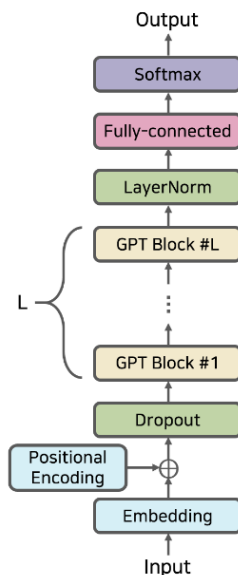


Diagram of the GPT model, each GPT block is a **transformer**.
Image: Minhyeok Lee, CC-BY-4.0

Building a working LLM

Now that we understand the processing taking place inside a **transformer** block, we have to look into a couple of important technical details that are required to make a language model work. In particular, we will focus on models whose task is to **produce text that responds to a user’s prompt** by **generating a sequence of output tokens** that correspond to the text that has the highest probability of being the correct completion for the corresponding prompt.

Consider the following block of text (which may be part of the training data for an LLM):

‘It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way—in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.’

(A Tale of Two Cities, by Charles Dickens)

Can we build a language model such that, given as input some length of the text above is able to predict **what comes next**? For instance:

Given as input

‘it was the season of Darkness, it was the’
 ‘we were all going direct to’
 ‘It was the best of times, it was the worst’

The model predicts

spring
Heaven
of

The task of the model is to choose **the highest likelihood word** that extends the prompt by one word. We can then use the newly predicted word to **expand** the prompt by one word and run the model again to predict the most likely **next word**.

Given as input

‘It was the best of’
 ‘It was the best of **times**’
 ‘It was the best of **times**,’
 ‘It was the best of **times**, **it**’

The model predicts

times
,
it
was

We can keep doing this as long as we want, and the model will keep generating new text. While it is fun to have a model re-create text it has already seen, the actually useful thing is that **once it has been trained**, the model can produce **useful text to extend prompts it has not seen before**. Which means such a model can produce **meaningful** and often **useful** answers for all kinds of user prompts.

It is **incredibly important** to remember that the text produced **does not have to be logically consistent or factually correct** (though it often is both). It just has to have **a high probability of plausibly extending the current sequence of tokens** being presented to the language model. Keep the examples below in mind whenever you are using an LLM to generate text!

Can you give me a list of fruits whose name ends in 'um'?

Sure! Here's a list of fruits that end with "um"—though it's a pretty rare ending, so there aren't many:

1. **Plum**
2. **Crabapple** (scientific name: *Malus pumila*, though this one is a bit of a stretch)

Unfortunately, "um" is an uncommon ending for fruit names, and I couldn't find many examples. If you have specific fruits in mind or need help with something else, feel free to ask!



fruits that end in um

AI Mode All Images Shopping Videos Short videos Forums More Tools

AI Overview

The common fruit **plum** ends in "-um". Other less common examples include **quince** and **persimmon**. Some joke or fabricated fruit names also end in "-um," like "applum" or "bananum".

- **Plum**: A very common stone fruit.
- **Quince**: A hard, fragrant fruit that is often cooked.
- **Persimmon**: A sweet, reddish-orange fruit.
- **Umeboshi**: A Japanese pickled plum that is often translated as "salted Japanese plums".

The most common fruit whose name ends in "um" is:

- **Plum**

Other possibilities are sometimes mentioned, depending on how strictly you define "fruit" or if you consider less common terms:

- **Applum** (A hybrid of apricot and plum, often sold in the foodservice industry).
- **Capsicum** (Botanically a fruit, though commonly used as a vegetable, e.g., bell peppers).

I noticed some search results also mention humorous or non-standard creations like "Bananum" or "Strawberrum," but those are not recognized names for actual fruits.

Is there anything else I can help you find out about fruits or food names?

The output above should not surprise you – and it doesn't mean these models are *dumb* or *not useful*.

The models learn to figure out *meaning* and we are asking for something that has little or nothing to do with a word's meaning: how a noun sounds, and what it means, are not closely related. Bottom line – the output of a text-generating LLM **can not be taken to be logically or factually correct**. It has to be carefully checked before being put to any serious use

As noted earlier, there are two technical details we have to look at before we can understand how a language model may be able to accurately produce a prediction for the **next word that extends the input**.

Firstly – the processing that happens inside the **transformer** depends exclusively on the **values of the features within the embedding vectors** provided as input to the transformer. All the matrices involved, the **keys**, **queries**, and **values** as well as the processing that happens within the **feed-forward** fully connected network can only use the content of the **embedding vectors**. For this reason, the processing within a transformer **does not use the information regarding the positions of the tokens in the sequence**.

The token's position within a sequence is critical for determining its **meaning**. It is a problem that, as described above, the **transformer architecture** does not have access to, or a way to use this information.

Positional encoding

The fundamental problem we have to solve is that the **transformer** can only access the values stored in the **features of each embedding vector**. So if we want to provide information regarding the relative position of **tokens** in the input sequence, such information **has to be encoded into the feature vectors themselves**.

At the same time, we designed the **embedding** so that transformations to the feature vectors change the **meaning** of the **tokens** they represent. So we can not just go carelessly transforming the encoding

vectors and expect the resulting directions to work. To make positional encoding work, we will take advantage of the high dimensionality of the token embeddings. In such high dimensional spaces, choosing a pair of random directions is likely to result in vectors that are nearly orthogonal – which indicates the space of possible directions is huge, and we can find a set of directions to use for the positional encoding that are **un-correlated** with whatever directions the embedding uses to represent **meaning**.

There are several approaches currently in use for encoding position within the feature vectors, and the details of which particular one is used by which model are not so interesting as the concept of how position gets encoded in embedding vectors. Here we will consider the original encoding from Vaswani et al. which modifies the **token embedding vectors** \mathbf{x} by adding a positional encoding component \mathbf{r} such that

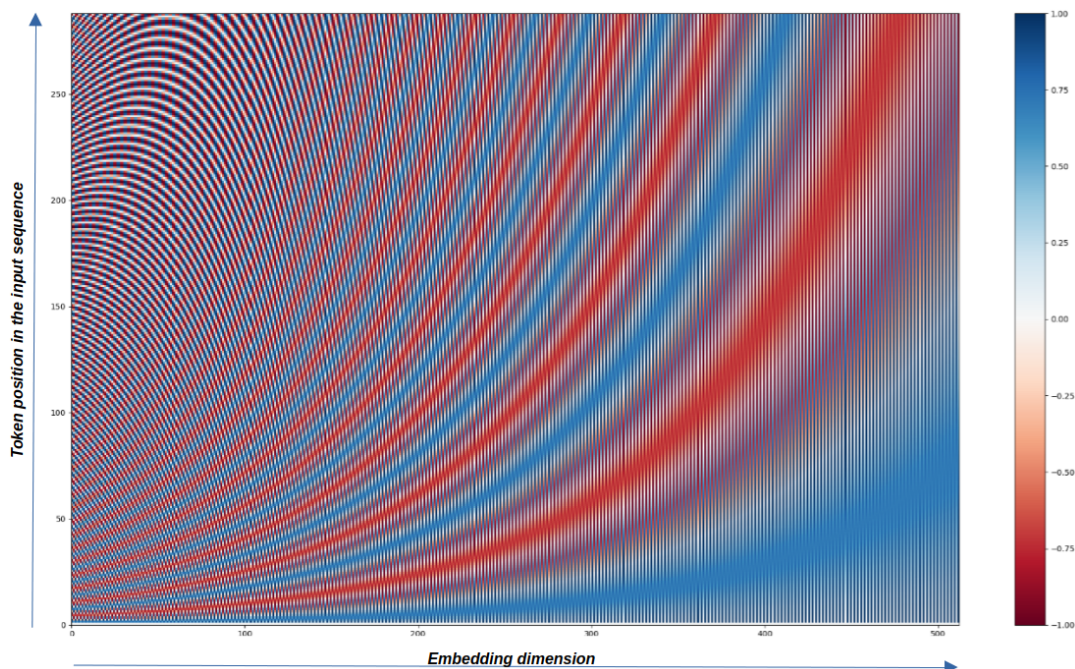
$$\vec{x}'_i = \vec{x}_i + \vec{r}_i$$

The entries in vector \mathbf{r}_i for a **token at position i in the sequence** are taken from a family of either **sine** or **cosine** waves in which the **frequency** is different for each **feature j in the embedding vector**:

$$\vec{r}_{i,j} = \begin{cases} \sin(10000^{-[j/D]i}), & i \text{ is even} \\ \cos(10000^{-[(j-1)/D]i}), & i \text{ is odd} \end{cases}$$

(Vaswani et al., 2017)

This is not particularly intuitive, so let's visualize what it happening:



Visualization of the vector \mathbf{r}_i added by the sinusoidal positional encoding described above. For a token whose position in the sequence is i , the positional encoding vector \mathbf{r}_i is given by the i^{th} row in the matrix above. Plot by Yang et al., CC-BY-NC-ND-4.0

The positional encoding vector is not random, if you consider the positional encoding of a few consecutive tokens, you can verify that their positional encoding provide a unique fingerprint for each

token that predictably depends on their position within the input sequence. This additional information allows the language model to learn **queries**, **keys**, and **values** that take into account the positions of related tokens within the input sequence as part of the attention process.

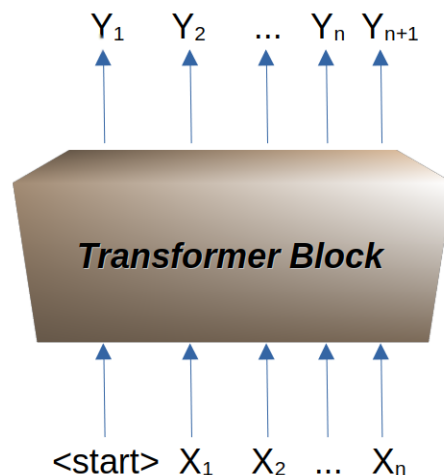
Up to this point we have assumed that the **word embedding** used to represent **tokens** was created separately from the LLM, but in practice, the best performance is achieved by **learning the embedding together with all the other parameters in the model**. The embedding vectors are initialized randomly and the same process of gradient descent used to learn **queries**, **keys**, **values**, and **weights for the feed-forward blocks** is also used to learn the **embedding vectors**.

With this in mind, it should be easier to understand that the resulting embedding will work with the positional encoding vectors – positional encoding will have been used throughout the training process that resulted in the embedding vectors themselves.

Predicting the next word

The final piece of the problem is to consider how to actually obtain a prediction from the LLM. So far, a **transformer** or a **stack of transformers** can take an input sequence of **tokens** and produce transformed output vectors for each that have been enriched with meaning derived from the context provided by the input tokens themselves as well as the knowledge the model has accumulated during training. However, the output is just a set of **transformed tokens** corresponding one-to-one to the input. So, how is it possible for the model to generate a **prediction** for the next token that best continues a specific input?

First step – shift the output with regard to the input. Since we want to predict **the next word** in a sequence we will **train the model so the transformed tokens in the output are shifted by one** with regard to the input as shown below:



The input is padded with a special **start** symbol, and the **LLM** learns during training to predict each **output token** from the **preceding ones** in the input sequence. The last vector output by the **LLM** is the **embedding corresponding to the predicted next token** that we want.

It is important to understand that this last **embedding vector** is not itself a prediction – it just points towards a location in the **embedding space** that corresponds to the **meaning** expected from the **next token** the **LLM** needs to produce. We still have to **translate this embedding vector** into a prediction for a specific **token** in the **vocabulary**. Therefore, we add one final **fully-connected** layer (see the GPT diagram above) to the **LLM** whose job is to take **the embedding vector** y_{n+1} and translate it into a **logits vector** with the same size as the vocabulary where the value of each entry gives **the probability that the corresponding token should be next**. The **logits vector** can then be passed through a softmax function to turn it into a probability distribution over all tokens in the vocabulary.

Given the **logits vector**, predicting the next token can be done in one of two ways:

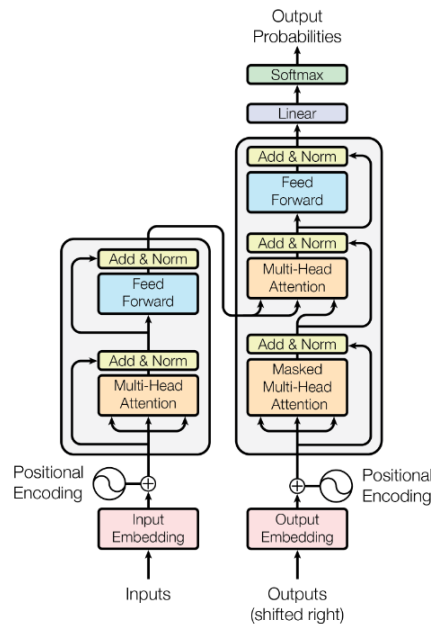
- Choose the token whose probability is highest
- Sample from the **logits vector** such that a **token** is selected with probability given by its corresponding logit

In either case the **LLM** has now produced a new **token**, the expanded text can be fed back through the **LLM** to continue text generation. The process continues until the **LLM** decides it has come to the end of a plausible sequence of text that answers a user prompt. One way in which this is achieved is by having a special **token** ‘<eos>’ to indicate the end of a sequence. This token is added to every document the **LLM** is trained on, so the model learns where a **natural ending point** may occur for a given text passage. During generation, if the next **token** in the sequence is the ‘<eos>’ token, the generation process stops.

Cross Attention

The same architecture described above for generating text can be put to use for a variety of important applications. For instance, **summarization** and **language translation**. Both of these applications rely on an extension to the **attention mechanism** that allows for multiple sources of information to be used by a **transformer network** in order to determine **tokens** to generate.

The following diagram shows **cross attention** at work in a typical **transformer** networks

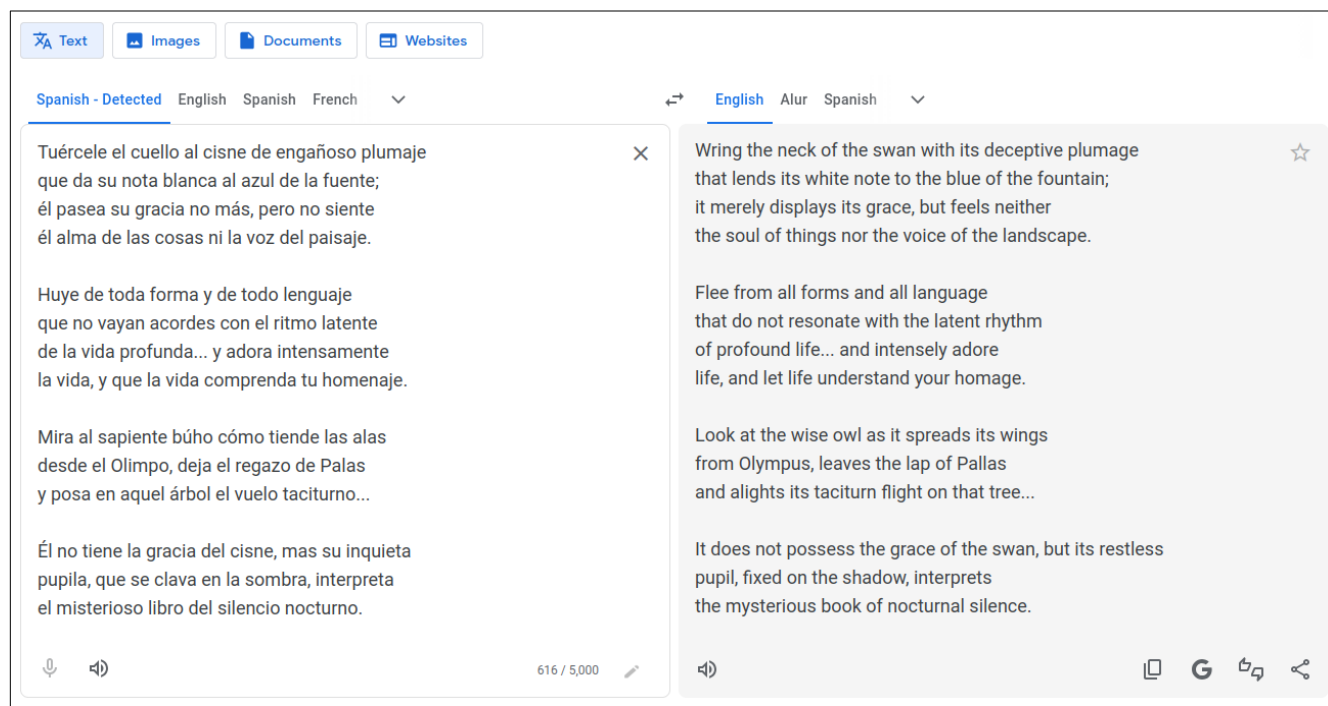


Cross-attention mechanism (Vaswani et al. 2017)

Suppose we want to do automatic text translation – for instance, we want to translate ‘*Buenos días, quiero dos tacos con salsa por favor*’ into English. The left-hand side of the network above is an **encoder** – it takes the input text in Spanish and **produces embedding vectors with refined meaning** as we would expect from a regular **transformer** network – the right side of the network is a **decoder** network, it looks at the **generated output** in English, and produces the translated text.

It works like a standard **transformer** network, except for a **cross-attention block** that receives the **encoder’s output** and incorporates it into the attention mechanism of the **decoder** network – this is what allows the **meaning** of the **tokens** derived from the text in Spanish to inform and guide the generation of the equivalent text in **English**. Inside the **cross-attention block** the **keys** and **values** are generated from the **encoder’s output** (the original Spanish text), while the **queries** are generated from the **decoder’s output** (the generated translation).

The entire model is trained on a large corpus of text with corresponding translations. Automatic translation has improved dramatically with the development of **transformer networks**, as the sample below demonstrates from a poem by *Enrique Gonzalez Martinez*.



This is a very good translation despite the fact that poetry is typically difficult, and illustrates what is possible with **transformer networks**. The material in these notes covers the classic **transformer language models**. Current **LLMs** are incredibly sophisticated and can handle text, mathematical notation, formulas (for example Physics or Chemistry), allow for multi-language text, and support multi-modal queries (e.g. images or sound recordings).

*Acknowledgements: A big **thank you** to Allan Jepson for carefully reading through these notes and providing me with detailed and very precise feedback on how to make them better!*