Introduction to AI Transformers

Module 1.1: Advanced

Objectives:

- Identify the differences between Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL).
- Describe transformer architecture: Attention mechanism, Encoder-Decoder structure, Multi-head attention, and Position-wise Feed-Forward Networks.
- Gain a brief understanding about key transformer models: BERT, GPT, T5, etc.
- Define real-world applications of transformers.

Introduction

This lesson is an introduction to transformer architectures, which is the technology that drives large language models like **GPT**, which stands for **G**enerative **P**re-trained **T**ransformer. Here we will cover the differences between artificial intelligence, machine learning, and deep learning. We will go into aspects that comprise transformers such as the attention mechanism, encoder-decoder structure, multi-head attention, and position-wise feed forward networks. The last part of this lesson will expand on key transformer models like GPT and BERT, and then define some real-world applications of transformers.

1. Differences between Artificial Intelligence, Machine Learning, and Deep learning

Artificial intelligence (AI), machine learning (ML), and deep learning (DL) are all interconnected fields. Each has a purpose, and they can be overlapping throughout the fields. AI is an umbrella term that covers a wide range of technology that involves machines performing tasks that would have typically required human intelligence. Generally, these are tasks involving decision-making, visual perception, or processing language. ML is a subset of AI where the focus is on enabling machines to learn from data, without being explicitly programmed. In comparison to traditional programming where the rules/instructions are explicitly programmed, in ML the rules/instructions are learned from the algorithm that is used. DL is then a specialized subset of ML, where neural networks are used to learn and model complex patterns from large datasets. DL algorithms using multi-layered neural networks have revolutionized the field of AI with their ability to be accurate in tasks such as image recognition, language translation, and processing speech.

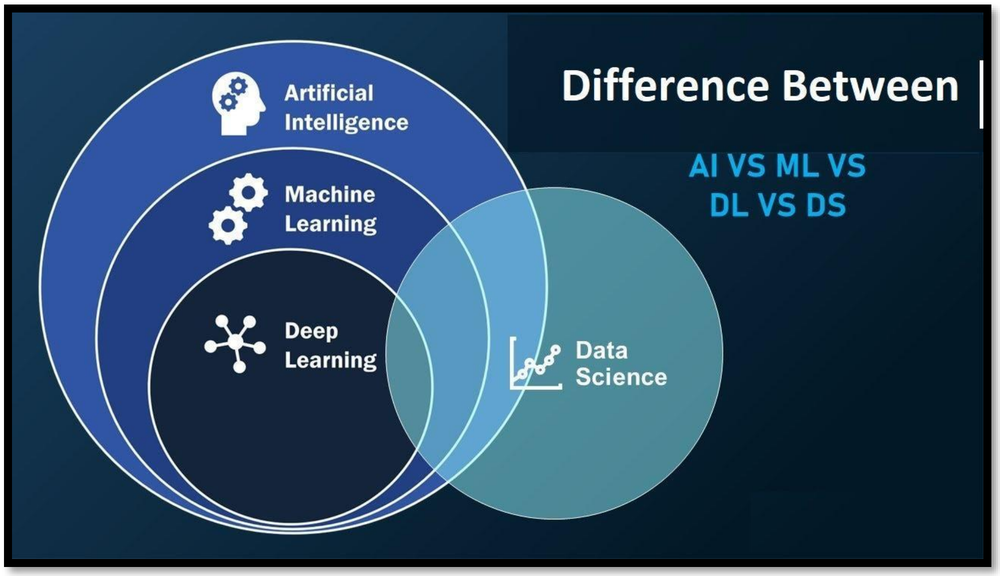


Figure 1: Diagram illustrating the relation between AI, ML, and DL. From https://medium.com/@shahinaperveen9444/differences-between-ai-ml-dl-ds-cc899d6921bf

To summarize, AI is the entire domain of creating machines that learn, and then ML and DL give us specific methods to complete these tasks using learning that is driven by large datasets.

While traditional ML/DL/NLP models are very useful (refer to the asynchronous machine learning materials for more info on specific models), they have some weaknesses that limit how useful they can be.

One weakness of traditional deep learning models such as recurrent neural networks (RNNs), is that they process data sequentially. This sequential processing means that the data cannot be processed in parallel, the words must be processed one at a time. This drastically increases the computational time.

Another weakness of the traditional models is something called the vanishing gradient problem. Essentially this is a problem during backpropagation in deep learning models, where a sequence of gradients of loss functions are multiplied by derivatives of activation layers, for each layer (for a refresher on deep learning models i.e., neural networks, and their components, see the machine learning asynchronous material). These involve small numbers so as subsequent layers are multiplied, these values go towards zero. This affects traditional models such as RNNs because if they are analyzing a long sentence, they have a "short memory" i.e. by the end of the sentence, the start of the sentence is pretty much forgotten.

Combining these problems together, traditional models are difficult to scale up to larger sequences, such as an entire book. They become too computationally expensive and would take too long to run.

Transformers were introduced to fix these problems. They solve the parallelization problem by introducing the self-attention mechanism which allows for computing the relationships between words in a sequence in parallel, making the calculations much faster and more scalable than sequential models like RNNs. The self-attention mechanism also allows for the memory of long-range dependencies as there is no more vanishing gradient problem, because instead of sequential multiplication of gradients in the layers of a neural network, they use positional encoding and attention weights. These details of transformers are what we will go over throughout this lesson!

2. Transformer Architecture

In AI, a transformer is a type of architecture in deep learning that was developed at Google and introduced in a 2017 paper called **Attention Is All You Need**. Transformers essentially convert words into mathematical representations, and the way these representations are related relates to how words are related. This is how tools such as ChatGPT can understand sentences.

2.1 Overview of transformers

GPT in ChatGPT stands for Generative Pre-trained Transformer, with generative meaning this is a model that generates new text. Pre-trained refers to how the model has been already trained on large amounts of data, and it is possible to fine-tune it to more specific tasks with additional training. A transformer is a specific type of neural network, and it is the key invention that has led to the recent massive innovations in AI and is a major part of what has brought AI to the eye of the general public, similar to what Google did with computers and the internet in the 1990s.

We will go through the process of how a transformer works. In the most general sense, they transform one type of data into another type of data. For example, AI image generators transform text to images. You could also transform audio to text to get an automatic transcription of a video. When you ask ChatGPT a question it is transforming text to text. The transformer in the previously mentioned 2017 paper **Attention is All You Need** was created to translate text from one language to another language.

The big picture idea of the transformer is that it takes an input sentence, or sequence of text, and makes a prediction for what the next word in the sentence or text in the sequence will be. More specifically, this prediction is a probability distribution of words, where the higher probability words are more likely to be the "correct" next word in the sequence.

Note:

A probability distribution is a way of encoding how often certain values will occur in some dataset. For example, heights of people follow a normal distribution.

One task you could imagine is running this method recursively. Let's say you begin with a starting sequence of words; you then generate the probability distribution of possible next words. You then take a random sample from this probability distribution and append it to the end of the sequence. Then, you generate the probability distribution of the **next** word after that, and repeat.

For example:

- "Alex Lifeson is the guitar player for"
- "Alex Lifeson is the guitar player for the"
- "Alex Lifeson is the guitar player for the Canadian"
- "Alex Lifeson is the guitar player for the Canadian rock"
- "Alex Lifeson is the guitar player for the Canadian rock band"
- "Alex Lifeson is the guitar player for the Canadian rock band Rush"
- "Alex Lifeson is the guitar player for the Canadian rock band Rush."

Here the text in red is the word generated in that step. In the third step for example, the probability distribution might include Canada, USA, and France, with Canada having the highest probability. The probability distribution for the fourth step might be rock, jazz, and EDM. You could see from this how it **would** be possible to randomly sample USA, or jazz, even though that is not correct here because Rush is not an American jazz band!

From this you can see how it is possible for tools like GPT to "hallucinate" facts, something to always be careful of.

Note:
In a 2023 personal injury suit against a Colombian airline, the defendant's lawyer found themself potentially facing legal sanctions after using ChatGPT to do research for the case. The case research included citations of cases that were completely non-existent, they were **hallucinated** by the large language model. This is a key lesson in knowing not just the strengths but also the weaknesses of large language models. Any important fact that you get from an LLM should be double checked with a classic internet search to verify that it is real.

Article: Lawyer Used ChatGPT In Court—And Cited Fake Cases. A Judge Is Considering Sanctions

Article URL: https://www.forbes.com/sites/mollybohannon/2023/06/08/lawyer-used-chatgpt-in-court-and-cited-fake-cases-a-judge-is-considering-sanctions/

Transformers are mainly composed of tokens, embedding, positional encoding, and the multi-head self-attention mechanism. These involve an encoder-decoder structure. Essentially, the encoder is what processes the context of the input sequence, and the decoder is what generates the output sequence (for example translated or summarized text).

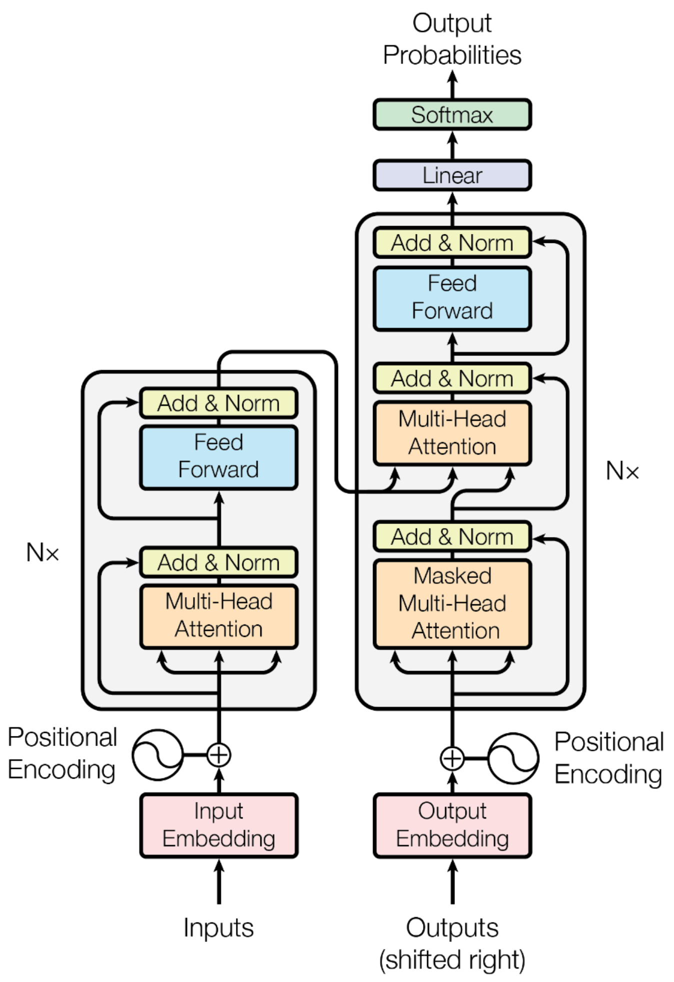


Figure 2: A diagram of the encoder-decoder structure from Attention is All You Need.

The overall steps that the transformer goes through to generate the probability distribution of the next word are:

1. Tokens and tokenization
2. Word embedding
3. Positional encoding
4. Attention mechanism

To briefly define each one, before we go into more detail, tokens are the units of text used for NLP which can be words, characters, or groups of words. The size of the tokens is called the "granularity" and depends on the tokenization method used.

Word embeddings are the tokens converted to vector representations, where the semantics and meanings of words are encoding in a large dimensional vector space.

Positional encoding gives information on the order of the tokens in the sequence, which is important because transformers process data in parallel (i.e. "all at once"), not sequentially like more traditional models.

The attention mechanism is how the meanings of entire sequences are calculated by analyzing relevant parts of the input sequence.

Let's now walk through these steps and how they work!

2.2 Tokens and Tokenization

In language models, tokens are the fundamental units of input data which are processed. Words must be first converted to tokens before the "word embedding" step can be carried out. This allows for complex data like sentences and paragraphs to be broken down into smaller pieces. In natural language programming, known as NLP, tokens can represent not just words but also linguistic elements such as syllables, characters, and punctuation marks. For example, the sentence "She went to the store." could be split into tokens that as a Python list would look like:

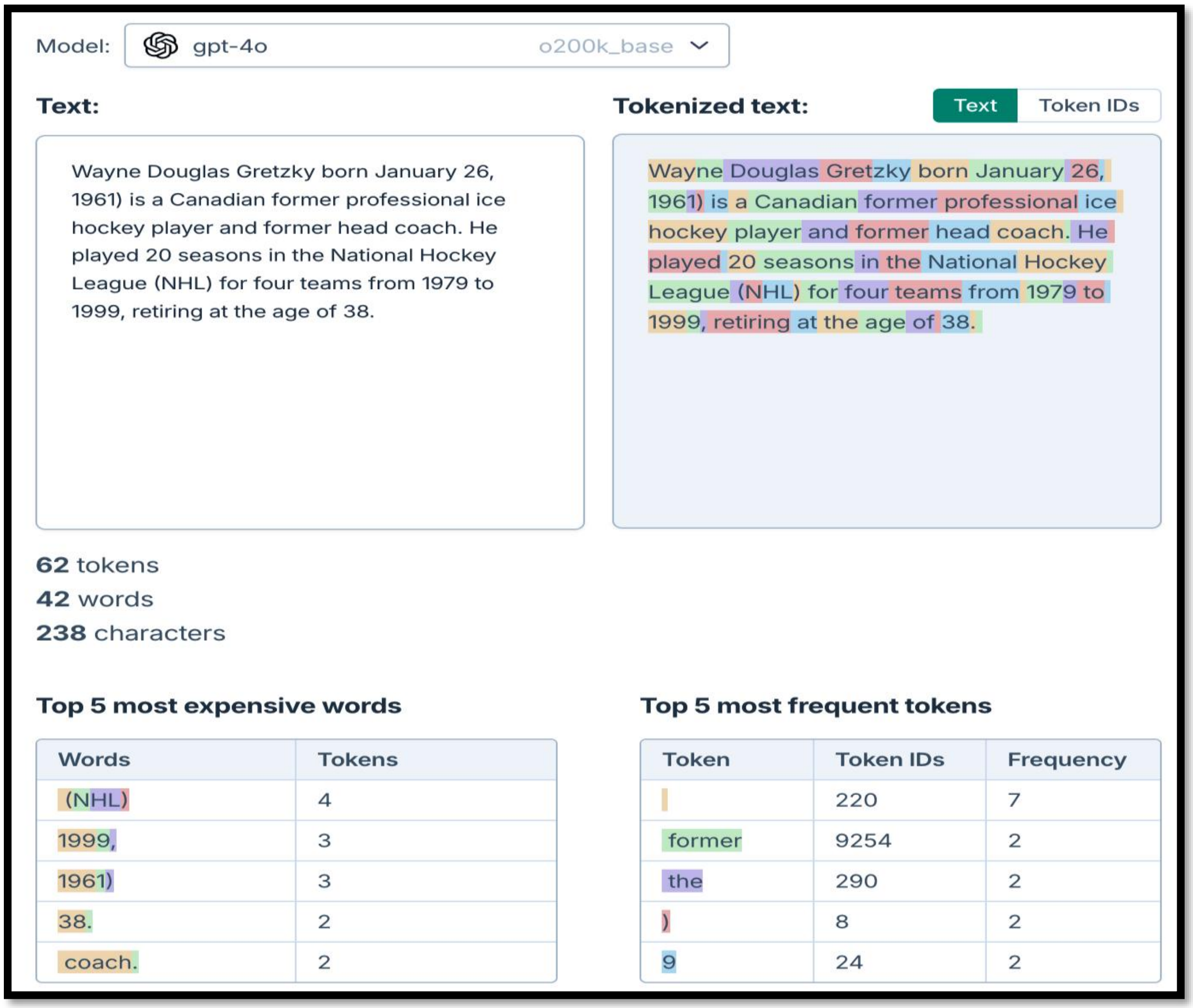
["She", "went", "to", "the”,"store", "."]

Here is a fun tool for seeing how the tokens look for different sequences:

GPT Tokenizer Playground

GPT Tokenizer Playground URL: https://gptforwork.com/tools/tokenizer

An example:


Model: gpt-4o o200k_base

Text:

Wayne Douglas Gretzky born January 26, 1961) is a Canadian former professional ice hockey player and former head coach. He played 20 seasons in the National Hockey League (NHL) for four teams from 1979 to 1999, retiring at the age of 38.

Tokenized text: Text Token IDs

Wayne Douglas Gretzky born January 26, 1961) is a Canadian former professional ice hockey player and former head coach. He played 20 seasons in the National Hockey League (NHL) for four teams from 1979 to 1999, retiring at the age of 38.

62 tokens
42 words
238 characters

Top 5 most expensive words

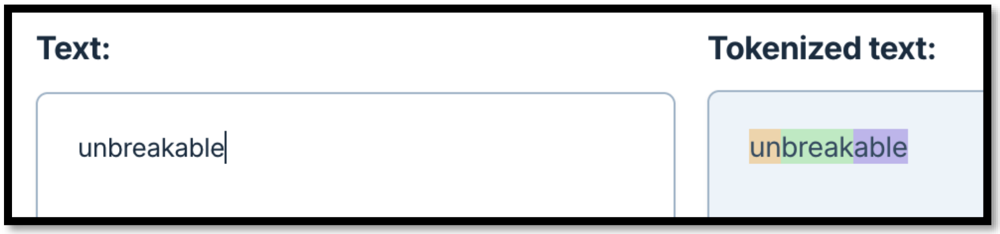
Words	Tokens
(NHL)	4
1999,	3
1961)	3
38.	2
coach.	2

Top 5 most frequent tokens

Token	Token IDs	Frequency
	220	7
former	9254	2
the	290	2
)	8	2
9	24	2

Figure 3: An example of a sequence being broken down into its tokens, using the tool available at https://gptforwork.com/tools/tokenizer.

Tokenization is the process of splitting sequences into tokens. Language models often use subword tokenization, see, for example, how the word "unbreakable" gets tokenized:


Text:

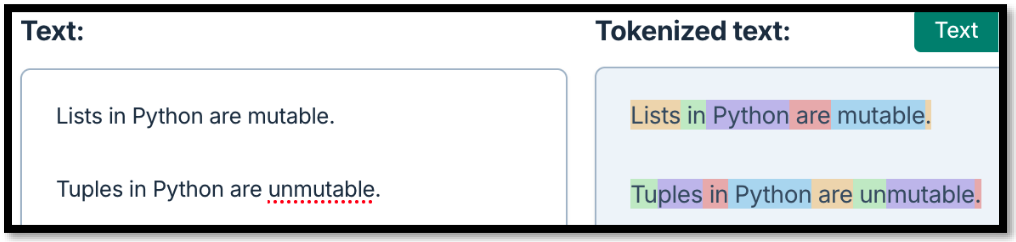
unbreakable

Tokenized text:

unbreakable

Figure 4: The tokenization of "unbreakable".

You can imagine then how the word "breakable" would be tokenized. These two words are similar, but the prefix "un" flips the meaning. Here is one more example, that talks about the difference between lists and tuples in Python:



Lists in Python are mutable.

Tuples in Python are unmutable.

Figure 5: Two sentences describing lists and tuples in Python. Look at how the prefix "un" flips the meaning of the sequence.

Tokenization allows the models to handle a large vocabulary and understand different forms of words, even if the model has not seen that specific word before. Tokens allow language models to analyze patterns and context within text. This allows the models to understand and generate coherent sequences and language.

This is how tokenization words for words, the concept can also be expanded to images and audio. Here are examples using PyTorch, of audio classification and image classification.

2.2.1 Word Embedding

Once we have the tokens, each token is associated with a vector representation. These vectors could be visualized as being coordinates in a high dimensional vector space, where words with similar meanings are "close together" in the vector space.

To visualize this concept, let's look at a 2-dimensional example of word embeddings (in this simplified example the tokens are entire words):

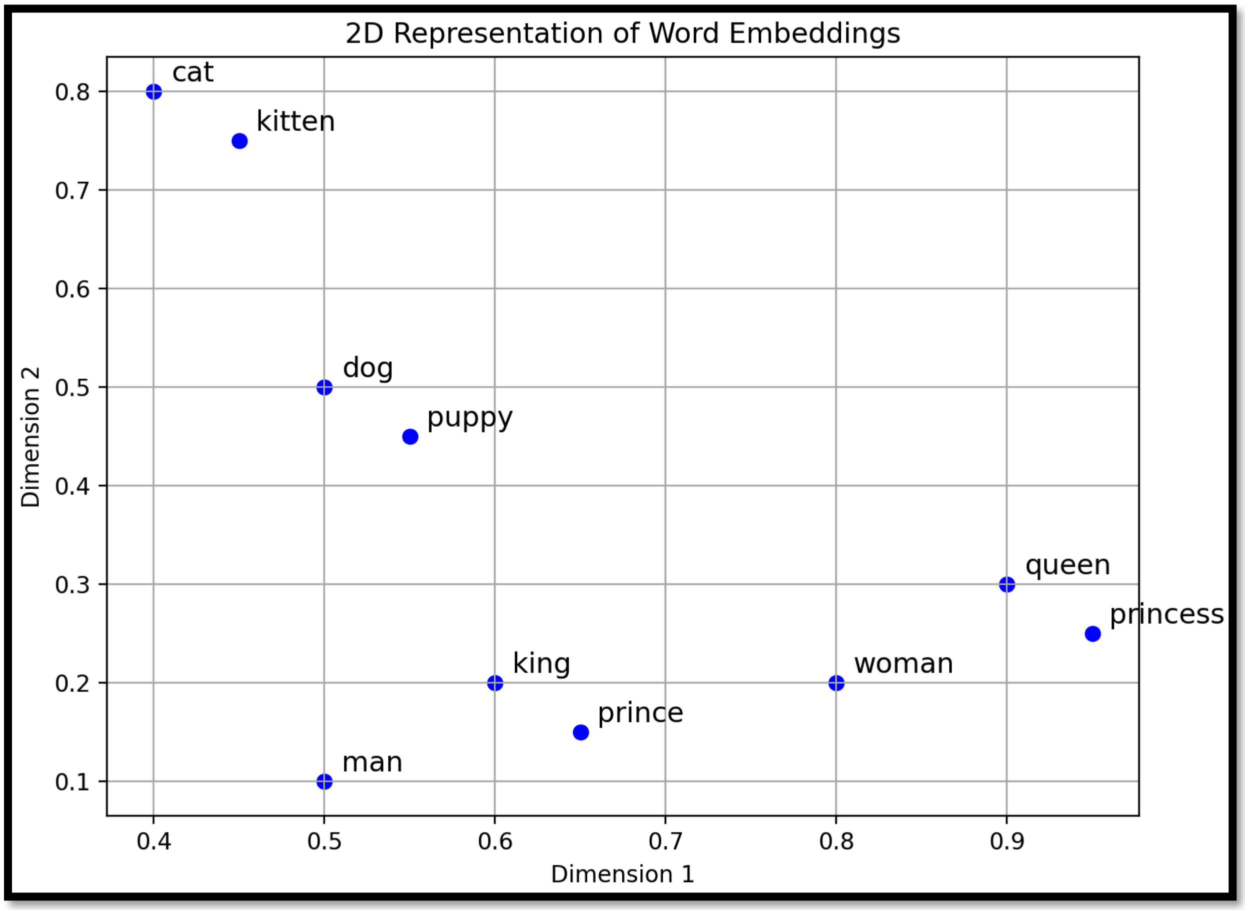


Figure 6: Embeddings

We can see how the direction between "man" and "king" is the same as the direction between "woman" and "queen", indicating that this direction in the vector space encodes the concept "royaltyness". Similarly, "king" to "prince", "queen" to "princess", "cat" to "kitten", and "dog" to "puppy" are all in the same direction, indicating that this direction in the vector space encodes the concept "offspringness". In two dimensions this would be very limited as there are only two possible orthogonal directions, if we wanted to encode a third concept here it would be a combination of "royaltyness" and "offspringness", which is why in practice these embeddings are in much higher dimensional vector spaces.

Note:

In GPT-3 the embedding vector space has 12,288 dimensions!

2.2.2 The Word Embedding Matrix

The embedding matrix is how we link tokens to vector representations, so that we can encode concepts like in the example above. The embedding matrix has one column for each token, so the number of columns can be thought of as the size of the vocabulary. The number of rows can somewhat be thought of as the ability of the model to understand context, semantics, and relationships between words. It is essentially the information in the language. It is important to also consider the computational resources available, for example if you had a vocabulary of 50,000 tokens and a 300-dimensional embedding vector space, this would result in 300 x 50000 = 15 million parameters just for the word embedding matrix.

This action of converting words into vectors that reside together in a vector space is an integral part of AI and language models. To clarify the earlier example with the 2D plot of word embeddings, the directions encoding meanings and concepts are an emergent property of the training. In other words, the meanings and concepts that are characterized by directions in the vector space are what is learned by training the model, they are not predetermined or set by the user.

Note on the ethical considerations of the word embedding matrix:

In general, when you train an AI model on some dataset, the biases in the training dataset will be reflected in the final results of the model. In the same way, for a language model, the biases in the training text will propagate through to the word embedding matrix and therefore the generated text. This phenomenon is outlined in the paper **Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings.**

There is a publicly available group of related models for embedding called "word2vec" that is trained on text from Google News that is written by professional journalists. Even the word embedding matrix here contains disproportionate word associations that reflect gender and racial biases.

For example, you might be able to see from our simplified 2D example above how the word embedding matrix could easily result in an oversimplified view of gender and societal roles.

2.3 Positional Encoding

Unlike traditional models such as RNN, transformers do not inherently know the token order, due to their parallel computations. This is very important for understanding sequences for tasks such as natural language processing. Positional encoding is the aspect of transformers that allows them to take into account the order of tokens or words.

For example, without positional encoding, the two sequences "The cat ate a fish." and "The fish ate a cat." would be identical, which clearly loses key information. Positional encoding attaches a vector to each token's embedding which represents its position in the sequence.

2.4 Attention Mechanism

The attention mechanism is a key part of the transformer architecture. It was introduced in the 2014 paper *Neural Machine Translation by Jointly Learning to Align and Translate* (https://arxiv.org/abs/1409.0473) by Bahdanau, Cho, and Bengio. Its core task is to determine which words in the sequence are important when processing each specific word.

Note:
In AI, attention is the method of determining the importance of each component of a sequence relative to all the other components of the sequence.

In other words, the attention mechanism is the part of the transformer which processes and understands the input sequences. It determines the relevancy between tokens, building contextualized representations between words.

First, the attention mechanism takes in the embedding vectors corresponding to the tokens from the input sequence. These vectors are in a high-dimensional space. For example:

2.4.1 Input is Embedded Tokens

Input sequence: "The cat sat on the mat."

Tokenized sequence: ["The", "cat", "sat", "on", "the", "mat","."]

Note that here we take the entire words as tokens but in practice the words could be broken down more.

2.4.2 Heads: Query, Key, and Value Vectors

The attention mechanism then transforms each embedding vector into the following vectors (with simplified explanations):

- Query (Q): What is this token looking for?
- Key (K): How relevant is this token?
- Values (V): Information about this token

Each set of query, key, and value vectors comprise one "head".

Note that these definitions are somewhat simplified but are a good overall view of what is going on here.

The query vector, representing what each token is looking for in each other token, will act as a search query, like an SQL lookup. For example, in the sentence "The cat eats fish.", the query could represent how the subject "cat" is performing the action "eat" on the object "fish".

The key vector is then representing the relevance of tokens to other tokens. So in the example "The cat eats fish.", "cat" and "fish" would have a strong relevance to each other.

The value vector then represents the information related to the token, which relates to the word embedding discussed earlier.

Mathematically, these vectors are calculated by multiplying the embedding matrix (each column is the embedding vector for each token) by the query weight matrix, key weight matrix, and value weight matrix. These matrices are learned through training. So, each token has a corresponding query vector, key vector, and value vector.

2.4.3 Attention scores

Next, is to calculate the attention scores. This tells us how important a token is towards generating the output. This is calculated by taking the dot product between each of the query and key vectors, and then scaling the value vectors by the results of the dot product.

Reminder:
Calculating the dot product between two vectors returns a single number, i.e. a scalar. If we have two vectors (a,b,c) and (x,y,z), the dot product would be calculated as:

a*x + b*y + c*z

This can also be thought of as a weighted sum. Geometrically, this is a projection.

2.4.5 Softmax function

These resulting vectors (query dot key, multiplied by value) are then concatenated into a matrix called the attention scores.

The next step is to convert the attention scores so that they can be interpreted as a probability. This is done by ensuring that the scores are between 0 and 1, and they all sum to 1. This is done by applying the softmax function, which will assert these conditions. For a series of numbers denoted by x_i, the softmax function would be calculated as following:

$$\boxed{\mathrm{softmax}(x_i) = \frac{e^{x_i}}{\sum(e^{x_i})}}$$

In words, you take the exponential of each value and divide that by the exponential of every value summed up. After applying the softmax function, the attention scores have been transformed to attention weights. These weights tell the language model the importance of each token.

Next, the attention weights are used to calculate a new representation of each token. For this, a weighted sum of the value vectors is calculated, using the attention weights to weigh the sum. We now have vectors that tell us how much attention each token should give to each other token; this is called the attention output.

2.4.6 Example of attention mechanism

Let's go over the example that we introduced earlier. For simplicity we will shorten it.

Since there is just one set of query, key and value vectors, this is one head of attention.

Input sequence: "The cat sat"

Tokenized sequence: ["The", "cat", "sat"]

After tokenization/embedding we will have the embeddings for "The", "cat", and "sat".

We now calculate the query, key, and value for each token. The query vector for "The" would be represented by the variable q_1, "cat" is q_2, and "sat" is q_3.

The same is done for the keys and values, represented by k_i and v_i.

The attention mechanism will calculate the dot product between q_1 and k_2 to find how much attention "The" will give to "cat". This is done for all pairs of tokens.

The softmax function is then applied and we now have the attention weights. An example is "The" could have a higher attention to "cat" compared to "sat".

The value vectors are then multiplied by the attention weights, and we now have a new representation for each token.

2.4.7 Multi-head attention

Multi-head attention extends the single-head attention mechanism. This allows for different relationships and contexts between words to be captured. One of the major breakthroughs in language models is that multi-head attention can be computed in parallel, greatly increasing efficiency.

2.4.8 Motivation

With single-head attention, we can have one copy of the attention between each token and each other token. You can see how this could limit how deep of an understanding we can have of the context of the sequence. For example, in the sentence "The cat chases the mouse", the word "cat" could relate to the word "chases" as the subject of the action but can also relate to "mouse" as the noun being chased.

With multi-head attention we can capture the different types of relationships and dependencies between the words by looking at different aspects of the relationship between words. Each attention head will learn a different representation of the input sequence, resulting in the model seeing multiple patterns and different contexts. Since a single head can just look at one aspect of the sequence, adding extra heads enables the model to learn different perspectives of the input which results in a better performance for tasks such as text translation or summarization.

Adding multiple heads is done by having multiple sets of query, key, and value vectors, where each set analyzes a different relationship within the input sequence. For example, one head could represent the grammatical structure of the sequence, and another head could represent meanings of words within the sequence.

2.4.9 How it works

Multi-head attention works by using multiple sets of the query, key, and value vectors. If we have N attention heads, we will have N versions of each vector being calculated in parallel.

For each token in the input sequence, the language model will have different weight matrices, to capture these multiple relationships between words. The attention is calculated multiple times, with each calculation starting with different query, key, and value vectors/matrices. This results in the calculated query, key, and value spaces to be varying, capturing different aspects of the input sequence.

The attention from each head is then concatenated together. A linear transformation is then applied to combine the information of the heads. This result has the combined context and relationships from the different attention heads.

Examples of these different relationships could be that one of the heads could focus on the grammar such as subject-verb or subject-object, another head could focus on semantics, and another could focus on the position of words and sentence structure.

2.5 Position-wise feed-forward networks

A key part of the transformer architecture is the position-wise feed-forward network (also called the FFN). After the multi-head attention weights are calculated, the new token representations are passed through the FFN. This refines the token representation to add non-linearity which can better capture complex patterns.

Motivation

If there was no FFN in the transformer, the results in the language model could be overly linear. The FFN introduces non-linearity. This allows the model to capture more complex relationships and contexts within the sequences. Non-linearity could result in reduced flexibility in token representations, and the meanings being shallower.

How it Works

The FFN's input is the token representations that are output from the multi-head attention step. This step refines the token representations. The FFN works on each token representation independently and is applied on each token in the sequence.

The FFN introduces non-linearity through non-linear activation functions, like in deep learning, these allow for more complex patterns to be learned by the model. This is a similarity between FFN and DL, which is different from traditional more linear methods. The FFN also introduces token-wise transformation (i.e. multiplication by weights and addition by biases).

Note:

One key difference between the attention mechanism and the position-wise feed-forward network is that the former involves the tokens interacting with all other tokens, and the latter acts on each token fully separately. This is why it is called "position-wise".

The FFN takes two linear transformations (fully connected layers in deep learning) which have a non-linear activation function in between them.

Each token embedding vector is multiplied by the weight matrix of the first linear layer and added to the first bias, this goes through the ReLU non-linear activation function (ReLU stands for Rectified Linear Unit), and then this result is multiplied by the second layer which is made of the second weight matrix and bias.

2.6 Output layer

The encoded output of the FFN is then sent to the decoder. The decoder contains attention and its own feed forward network.

The decoder first receives the encoded sequence. This is sent through a masked multi-head attention mechanism. The key to this being masked is that this is how it "guesses" which tokens/words will come next. This then goes through the same position-wise feed-forward, and finally a softmax function to calculate the probability distribution of the next token.

3. Key Transformer Models

Two of the most prominent transformer models in natural language processing are BERT and GPT. These were developed by Google and OpenAI, respectively.

3.1 BERT

BERT, standing for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers, is an encoder-only transformer model. Its key feature is the bidirectional context, meaning when processing sequences it takes into account the context from both left to right and right to left. This allows for a deeper understanding of meaning and context within text. The pre-training task involves masked language modeling, where certain words are masked in the input and the transformer model then tries to predict them based on the context. Another pre-training task is next sentence prediction, where the model looks to understand the relationship between sentences. Combining these two pre-training tasks is what allows BERT to be strong at understanding context, and be effective for applications such as text classification, named entity recognition, and question answering. A strength of BERT is how it adapts through fine-tuning, allowing it to be customized to natural language tasks.

4.1 GPT

In contrast, **GPT** (**G**enerative **P**re-trained **T**ransformer as stated earlier) is a decoder-only transformer model. GPT uses unidirectional context, where sequences are processed from left to right only. The most important pre-training task is next-token prediction, where GPT predicts the next token of a sequence. This allows for the generation of coherent and contextually relevant text, where the following words are anticipated based on the input. OpenAI has continued to release new versions of GPT where the model size keeps increasing. GPT is especially strong at text generation, conversation, and code generation.

4. Real World Applications of Transformers

4.1 General applications of transformer models

Transformers are a versatile tool with a wide range of applications. They are particularly useful in natural language processing (NLP). In NLP, transformers complete tasks such as text generation, where they can create text that is coherent. Another strength of transformers is language translation, due to the transformer's ability to understand context within sentences. Also, sentiment analysis, where the sentiments within text are identified to quantitatively understand a large amount of social media posts or product reviews. Another application of transformers is question answering systems, where precise answers can be provided to questions by determining context.

Expanding from NLP, transformers are also useful in computer vision. One example is visual question answering, where transformers can analyze both visual and text inputs to provide answers to questions about images or video. An example of this could be medical imaging diagnostics, such as analyzing X-rays or MRIs to detect abnormal features like a tumour or a fracture. Computer vision can also relate to object detection and recognition, where systems can identify and classify objects within images and videos.

Transformers can be used in healthcare for analyzing medical records. This can be in extracting and processing information from patient medical records to assist doctors with decision making. Another related use is drug discovery, where transformers can analyze a large number of chemical properties and interactions, to increase efficiency in identifying potential drug candidates. A third medical example is using predictive diagnostics to forecast patient conditions using historical data, which can help with detecting symptoms earlier.

Another domain for applying transformers is in finance. One example is algorithmic trading, where market trends can be analyzed to determine complex trends and make trading decisions. Fraud detection is another use, where anomalies in user activity can be determined to flag potential fraud in financial transactions. In risk management, transformers could be used to assess financial risks.

Entertainment platforms such as Netflix or Spotify can use transformers for content recommendation. These can analyze your past behaviour such as the shows you have watched or songs you have listened to, and further shows/songs can be recommended based on other user activity.

A domain where transformers are making a big impact is robotics and automation. This could be in the form of human-robot interaction, where robots could understand and respond to human speech by considering the context of the input sequence. Also, in autonomous systems such as self-driving cars where transformers are used to analyze the surroundings and make real-time decisions using sensor data.

4.2 Applications of BERT

BERT is great at handling text classification such as sentiment analysis, spam detection, and topic categorization. This is due to the ability to understand deeper contexts from its bidirectional nature. This allows the model to distinguish texts into different categories, such as classifying movie reviews into positive, neutral, or negative.

Another application of BERT is named entity recognition (also called NER). This is the classification of entities such as people or locations from text. BERT is strong at this, again, because of its bidirectionality which helps it to understand the context surrounding specific tokens. Examples of this would be labeling named entities in customer support tickets, medical records, financial documents, or legal documents.

BERT is also strong at question answering tasks. This would include being given a question and a passage and finding the text in the passage that answers the question. The bidirectional architecture will consider both the question and passage together, making it strong at finding answers even in a large body of text.

Language translation is another strength, again due to the bidirectionally helping to understand deeper context within sequences.

4.3 Applications of GPT

One of the key applications of GPT is text generation. This arises from the unidirectional architecture which specializes in predicting the next word of a sequence. This makes GPT strong at generating creative texts like stories, or informative texts like technical manuals (keep in mind all text should always be double checked for accuracy).

GPT is also useful for conversational AI and chatbots. This is because it generates natural sounding responses which take context into account. The long-range dependency when considering tokens in the input sequences is what allows for this.

Another very useful application is generating code, along with troubleshooting. GPT-3 has been fine-tuned to both understand and generate programming languages like Python, making it an excellent tool for helping with code. This works best if you describe in detail a specific function to write. This does not mean that you do not need to know how to code, you still need foundational knowledge to use GPT for coding.

5. Conclusion and Reminder

Transformers are a recently developed AI architecture that allows for efficient computations to understand the meanings of text sequences, from single sentences to entire books. These take in an input sequence, do some task such as translating to another language or summarizing, and return an output sequence. This process involves steps such as tokenization, word embedding, positional encoding, and multi-head self-attention.

When working with Large Language Models, always keep hallucination in mind. Remember that at their core, LLMs generate text that is based on the training data, depending on the model and whether retrieval augmented generation (a.k.a. RAGs, which are covered in a subsequent module), LLMs are not necessarily truth based, they are solely based on the patterns in the training data. For important information such as medical, legal, or engineering information, always double check any output from a LLM with a real source.