Transformer Based Learning for BERT and GPT Language Models

Avinash Kak

Purdue University

Lecture Notes on Deep Learning by Avi Kak and Charles Bouman

Sunday 27th April, 2025 11:44

Preamble

Large Language Modeling (LLM) made its debut in the year 2019. OpenAI officially released the GPT-2 model in February 2019 and Google officially released its BERT model in October 2019.

What is most distinctive about LLMs is that they can ingest terabytes of publicly available textual datasets, learn from that data without any supervision, and become experts in word-to-word, clause-to-clause, sentence-to-sentence, and paragraph-to-paragraph continuity properties of narratives.

My goal in this lecture is to highlight some of the important aspects of LLMs, the architectures of their neural networks that are based on Transformers, how they carry out unsupervised learning, etc.

I'll start by illustrating several of the LLM concepts through my explanations of BERT. The main reason for that is that my acquaintanceship with BERT dates back almost to the year it was born. The GPT models have entered my consciousness rather recently.

BERT started out as an acronym for *Bidirectional Encoder Representations from Transformers*.

Preamble (contd.)

However, from the way the acronym BERT is now used in publications and in general communications, you could say that BERT has become a noun unto itself. BERT was first presented in the 2019 paper by Devlin et el.: "*BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding,*":

`https://arxiv.org/pdf/1810.04805.pdf`

In retrospect, the word "Bidirectional" in what BERT stands for is an accident of history and now is probably a source of confusion for most new users of BERT. As you know, there's nothing fundamentally unidirectional about a Transformer.

[**As explained in my Week 13 lecture, a Transformer learns an attention map for the input. Ignoring the batch axis, if you feed an $[N_w, M]$ tensor at its input, where N_w is the number of elements (tokens, patches, etc.) in the input sequence and M the size of the embedding vector representation of each element, the output of the Transformer will also be shaped $[N_w, M]$, but with a difference. When you multiply the learned Q and K^T tensors for the final output, you will get a $N_w \times N_w$ array that is the Attention Map over the N_w elements of the input sequence. The element (i, j) of the of the Attention Map array indicates as to what extent the i^{th} element in the input attends to the j^{th} element in the same input.**]

The word "Bidirectional" in BERT came about because it was Google's advance over the OpenAI's first version of GPT that was programmed to calculate self-attention in an autoregressive manner — by scanning a sentence left to right. In an Autoregressive Model, the dot-products that go into the calculations of the attention for each word only depend on the previous words in the input sentence.

Preamble (contd.)

Large Language Models are meant to help us cope with the following dilemma: Solving a practical problem with complex neural-network architectures based on Transformers requires a large amount of labeled training data. But creating labeled data can be expensive.

You run into this dilemma particularly when you initialize the learnable parameters with random weights, a common practice for the simpler neural networks.

Over the years, researchers have posited that it should be possible to reduce the burden of supervised training for solving a specific task if **we first initialized the learnable parameters with inexpensive unsupervised training**. The researchers established the veracity of such claims with small-scale experiments.

But, now, through LLMs, we know that the above is true in general. We can significantly reduce the extent of supervised learning needed if we first initialize the learnable weights with inexpensive unsupervised learning on freely available public datasets.

Preamble (contd.)

I have already mentioned the main publication for BERT. For GPT-2, the official publication is the 2019 paper "*Language Models are Unsupervised Multitask Learners,*" by Radford et al.:

https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

and for GPT-3, the publication is the 2022 paper "*Language Model are Few-Shot Learners,*" by Brown et al.:

https://arxiv.org/pdf/2005.14165.pdf

Here is a brief chronology of the more famous LLMs (and one not so famous) that have been released to date:

```
          GPT-2       officially released in Feb 2019
            |
            v
          BERT        officially released in Oct 2019
            |
            v
          GPT-3       officially released in Nov 2022
            |
            v
          GPT-4       officially released in March 2023
            |
            v
    Try not to laugh
            |
        babyGPT       officially released in April 2025    https://engineering.purdue.edu/kak/distBabyGPT/
```

Preamble (contd.)

Here is a comparison of BERT with GPT-3 by Celeste Mottesi dated February 9, 2023:

https://blog.invgate.com/gpt-3-vs-bert

According to the author, whereas GPT-3 performs better at tasks such as summarization and translation, BERT can be expected to perform better at tasks such as sentiment analysis and natural language understanding.

Since GPT-3 was training on 45 TB of data, whereas BERT was trained on mere 3 TB, one would think GPT-3 would beat BERT hands down. Perhaps, the deep-learning mantra "*the more data you have the better off you are*" does not apply to every aspect of natural language processing.

Preamble – How to Learn from These Slides

At your first reading of these slides, just focus on thoroughly understanding the following four concepts related to LLMs:

- At the most fundamental level, you need to understand what's meant by *generative training* as opposed to *discriminative training*. This material is covered in Slides 10 through 13.
- How exactly unsupervised learning is carried out in BERT (Slides 30 through 35) and in GPT (Slides 64 through 68).
- You also need to understand as to why tokenizers play a critical role in generative training. What would happen if we used no tokenizers at all, just used the words directly as the tokens? This material is covered in Slides 16 through 29 and Slides 37 through 42.
- The concept of Learning in Context (LIC) that is presented on Slides 83 through 88.

That makes for a total of under 40 slides for your first reading. Please be aware, though, that you may first need to review the material in my Week 13 lecture to understand how the Transformers are used for LLM.

Outline

Outline

Generative vs. Discriminative Training

- For most people, the word "generative" is evocative of creating "something from nothing" — for example generating images from noise using either diffusion or adversarial learning. That is NOT the meaning of "generative" in "Generative vs. Discriminative".
- To convey the ideas of what's meant by "generative vs. discriminative" in training a neural network, let's say that X represents the input to the network and Y its output. Typically, you train a classification neural network discriminatively because you trust the training data and you want the network to make a correct prediction Y for the given input X.
- Therefore, the main focus in discriminative training is to get the network to correctly estimate the conditional probability $p(Y|X)$ — because the accuracy of X is not open to question.

Generative vs. Discriminative Training (contd.)

- In "generative" training, on the other hand, while you may still need to make discriminations based on the output Y, you also want to make the network learn the inter-relationships between the different elements of the input X.
- That way, the network would also become smarter about the probabilistic variability in the different X that may correspond to the same Y. In this manner, you could say that you want to put both the input X and the output Y on an equal footing with regard to where you want to posit your trust.
- That begs the question: How to learn X?
- Modern thinking is that if X has any structure at all — all languages are highly rich in structure — and if we could prescribe learning objectives for the different facets of that structure, all we would need to do would be to let the neural network loose in the internet so that it can ingest as many different examples of X as it can find and thus learn the structure of X.

Generative vs. Discriminative Training (contd.)

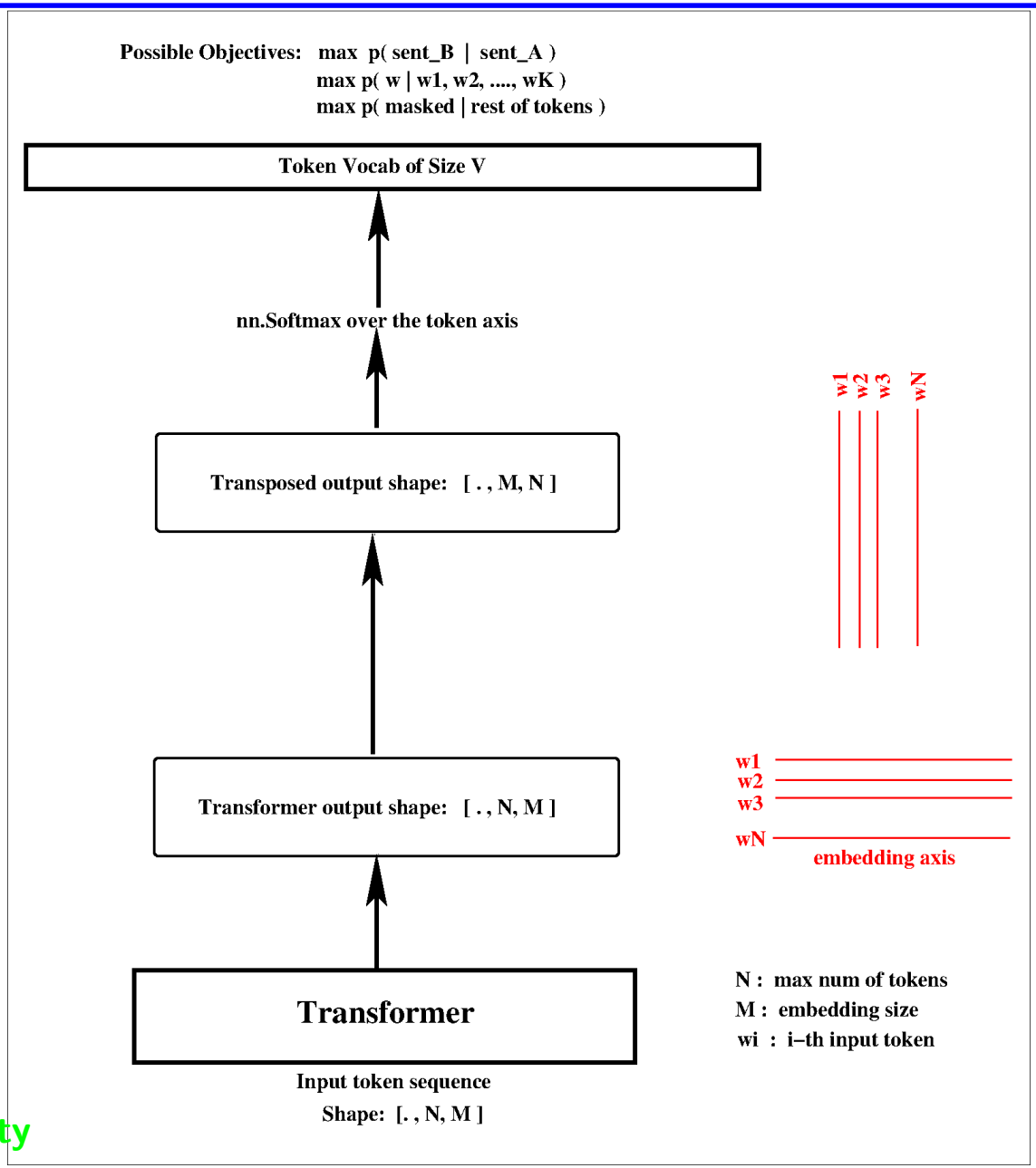
- Relative to the supervised training of neural networks, the approach outlined at the bottom of the previous slide should cost nothing.
- It is this idea that is foundational to LLMs. **They all use unsupervised learning to learn X.**
- The figure on Slide 14 is meant to capture the unsupervised generative learning at a very general level. As shown, you feed a sequence of tokens at the input to a Transformer based network.
- The precise format of the input sequence would depend on what sort of object function you are trying to maximize (or what sort of a loss you are trying to minimize).
- For example, if the goal is to train the network at NSP (Next Sentence Prediction), the token sequence at the input will have two separate parts, one for a given sentence ($sentence_A$) and the other for the sentence that follows ($sentence_B$). Now the objective would be to maximize the conditional probability $p(sentence_B|sentence_A)$.

Generative vs. Discriminative Training (contd.)

- On the other hand, in Masked Language Modeling, the goal would be to mask out one of the tokens in the input token sequence and to maximize the probability of the correct token being predicted at that position given all the context tokens. So if the input token sequence is expressed as $(w_1, w_2, \ldots, w_N)$ and if we consider the w_m token to be masked, at the output of the neural network we would want to maximize $p(w_m | w_1, \ldots, w_{m-1}, w_{m+1}, \ldots, w_N)$.

- For yet another possibility, if your language model is autoregressive — meaning that the production of each token would depend only on the previously produced tokens — you would want to maximize $p(w_i | w_{i-1}, w_{i-2}, \ldots, w_1)$.

Generative Learning



Outline

Tokenization: Old vs. New

- In old-style NLP (Natural Language Processing) as represented by the Python NLTK library, tokenization merely meant to separate a sentence into individual words, numbers, and punctuation marks.

  [In old-style NLP, the steps that come after tokenization are referred to as stemming and lemmatization. The goal of stemming is to reduce the different variants of a word (also knows as "inflections") to their common root. For example, stemming reduces the words "program", "programs", "programmed", and "programming" to the root word "program". Since stemming would sometimes lead to meaningless root words (example: "nothing"to "noth"), a more complex algorithm called lemmatization was invented for the same purpose that works with the constraint that the root word has to be a valid dictionary word. The main motivation for stemming and lemmatization was to reduce the size of the word vocabulary in order to reduce the memory and the processing costs. Since much of the early NLP work was focused on tasks such as text classification (for, say, sentiment analysis), it was believed that changing the word forms through stemming and lemmatization would not affect the overall classification. That would obviously not be the case for tasks that require finer-grained understanding of the text, as for automatic translation.]

- Tokenization as used in modern deep-learning based LLM bears zero resemblance to what it has meant in the past. Unfortunately, the old use has led to several misconceptions regarding its current usage, including the one that says that you need tokenization *primarily* to break up longer words into smaller words. In modern tokenization, whether or not a long word would get broken up into smaller subwords would depend on the frequency of occurrence of the long word.

Why Tokenization is a Must for LLM

- **The main purpose of tokenization in the context of LLM is to create a fixed-sized vocabulary for the corpus you are working with — regardless of the size of the corpus itself.**

  [**Unless your text corpus is based on a set of documents frozen in time, ordinarily, as the size of a text corpus goes up, the size of the vocabulary will also go up — despite the illusion to the contrary created by the fixed sizes of the language dictionaries you have seen all your life. How we express ourselves is a living thing. We are constantly inventing new words and new expressions; these form important components of what's referred to as the zeitgeist.**]

- Having a fixed-sized vocab is important because the loss functions used in the deep-learning networks used for language processing are based on maximum-likelihood calculations for the next token given the tokens seen previously. That requires estimating the probabilities associated with all possible tokens at the next position. As you can imagine, it would be impossible to engage in such probabilistic reasoning if you did not know in advance the size of the vocabulary or what precisely the vocabulary was.

- So how do the tokenizers ensure a fixed-sized vocabulary?

Tokenization for a Fixed Target Vocab Size

- The tokenization steps presented in this section are at the heart of several modern tokenizers that allow you to specify the size of the vocabulary for representing a text corpus. The starting point for these steps is what is known as the *base vocabulary*, which is the vocabulary of the basic alphabetic characters that go into the words, numbers, punctuations, etc.

- In the steps outlined on the next slide, the base vocab will consist of the chars corresponding to the *first* 256 Unicode numbers (or code points) that are assigned to every character in all the languages in the world.

  [**Note that there are two distinct concepts involved when using Unicode: the number that the Unicode assigns to each character and the bit pattern to be used for that number. The Unicode itself is about the former, whereas utf-8 is about the latter. The current largest Unicode number that utf-8 can represent is $0x10FFFF$ in hex. The Unicode numbers are represented by *U+hhhh* notation in narrative and by *\Uhhhhhhhh* and *\uhhhh* in code, with the former requiring exactly 8 hex digits and the latter exactly four and where 'h' is a hex digit. The utf-8 bit patterns are displayed using the *Oxhhhh* notation.**]

- Of the first 256 Unicode numbers, the numbers in the range 0-127 corresponding exactly to ASCII encodings. [**When you use `chr()` and `ord()` in this range, the behavior of these two functions is indistinguishable from the old days of ASCII dominance.**]

Tokenization for a Fixed Target Vocab Size (contd.)

- For Unicode numbers greater than 127, the utf-8 encoding follows the Unicode multi-byte representation rules as represented as shown below. Note that 'x' is for the free bits to be used for the bit pattern for the Unicode number in question.

	1st Byte	2nd Byte	3rd Byte	4th Byte	Number of Free Bits	MAXIMUM EXPRESSIBLE Unicode VALUE
Rule 1:	0xxxxxxx				7	007F hex (127)
Rule 2:	110xxxxx	10xxxxxx			(5+6)=11	07FF hex (2047)
Rule 3:	1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16	FFFF hex (65535)
Rule 4:	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21	10FFFF hex (1,114,111)

- Python's two important I/O functions for text files — `chr()` and `ord()` — are based on Unicode by default. The former returns the char for a Unicode number and the latter returns the Unicode number for a given char. Under the hood, the bit patterns used are based on utf-8 encoding according to the four rules presented above.

[For example, `ord('©')` returns the Unicode number 169, which according to the rules shown above would require a 2-byte utf-8 representation. The bit pattern for 169 is '10101001'. So its utf-8 representation would be

```
11000010 10101001
```

where the ^ symbol shows how the free bits in the 2-byte representation were used for the bits of the Unicode number 169.]

Tokenization for a Fixed Target Vocab Size (contd.)

What follows are the steps for what would be considered to be a fairly generic tokenizer today. Later I'll show my Python code for the steps.

1. Apply "pre-tokenization" to the training corpus. Pre-tokenization would ordinarily consist of separating the individual words on the basis of white space. [Assume that any marks (such as the punctuation marks, quotation marks, etc.) that are stuck to a word, either at the beginning of the word or at its end, as defining a unique word into itself. That is "book," is a word that is different from "book".]
2. Pre-tokenization may also include inserting between successive words a special symbol, such as the underscore character '_', that can be used by the decoder to facilinate merging the tokens back into whole words.
3. Represent every word generated by the previous step by a sequence of the characters in the base alphabet.
4. Measure the frequency of occurrence of the different pairs of consecutively occuring characters in the corpus. Frequency counts are measured on the basis of the number of words in which a pair of characters appears together.
5. Merge the most frequently occurring pair of characters into a subword and record the operation as a distinct merge rule.
6. Repeat the steps outlined above at the level of subwords in order to form longer subwords, until the tokenizer vocabulary has reached the user-specified size.
7. Repeat the steps outlined above at the level of subwords in order to form longer subwords, until the tokenizer vocabulary has reached the user-specified size.

The BPE Tokenizer in the babyGPT Module

- The babyGPT module includes a class called `TrainTokenizer`, presented on Slides 24 and 25, that is my implementation of the BPE logic.
- If you are experimenting with the babyGPT module with a trove of documents on a specialized domain (such as movies, music, academy awards, sports, etc.), you would want to first train a tokenizer that is meant specifically for that domain. With babyGPT, all you have to do is to use the `train_tokenizer.py` script in the Examples diredctory of the babyGPT module after you have set a couple of its variables:

```
from babyGPT import *

articles_dir = './saved_Adrien_News_Articles_56M'

tokenizer_trainer = babyGPT.TrainTokenizer(
                        corpus_directory = articles_dir,
                        target_vocab_size=50000,
                    )
tokenizer_trainer.train_tokenizer()
```

where `saved_Adrien_News_Articles_56M` is an "athelete news" corpus that was created by Adrien Dubois in Robot Vision Lab.

The BPE Tokenizer in babyGPT (contd.)

- The call shown on the previous slide will deposit the tokenizer vocab and the merge rules into a JASON file named `104_babygpt_tokenizer_49270.json` where the prefix '104' refers to the version '1.0.4' of babyGPT that was used for generating the tokenizer and the number 49270 the actual size of the tokenizer vocab.
- The two main components of the json file produced by tokenizer training are: a dictionary that has the mappings from the integer index values to the subwords, and an array of pairwise subword merges chosen during training.
- To apply the tokenizer produced to arbitrary text, you would need to make the following sort of a call:

```
python3  apply_tokenizer.py  text_sample_for_testing.txt  104_babygpt_tokenizer_49270.json
```

- You will find the script `apply_tokenizer.py` in the same Examples directory as mentioned previously.

The BPE Tokenizer in babyGPT (contd.)

- Shown below is the output produced by the call to `apply_tokenizer.py` on the previous slide:

```
THE ORIGINAL TEXT: :  One day after Serena Williams made headlines for wearing a "catsuit" on the court at the French Open, the 23-time Grand Slam champion has launched a fashion line of
 her own. Called "Serena" and sold exclusively online, pieces range from $35 to $250, according to Women's Wear Daily, and include athleisure wear, dresses, outerwear and more. Williams told
 the fashion publication that she wanted to be sure to create a clothing that felt "practical," but also one that focused on the concept of an s-word, including her name. "Everyone can have
 an 's' word. Mine is 'sure.' My mom's is 'steadfast.' A really good friend of mine who has been through a lot, her 's' is 'survivor,'" Williams explained to WWD. "This is a huge thing that
 I'm doing right now and a huge undertaking, and I need to be sure. And sometimes even when I walk out onto the court, I'm not sure, I'm not sure I'm going to win. I need to be more sure of
 myself and more confident in myself. And that's coming from me. And I feel a lot of people can relate to that." Williams, 36, has worked in fashion before, having developed lines with HSN
 and Nike. However, this line is the first one she's done on her own, and she made sure not to rush the process, especially given her evolving personal life. Last September, Williams and her
 now-husband, Reddit co-founder Alexis Ohanian, welcomed their first child, daughter Alexis Olympia Ohanian Jr.


ENCODED INTO INTEGERS:  [34043, 10274, 10673, 14482, 16726, 37175, 16592, 115, 5507, 10405, 9623, 97, 259, 13035, 49674, 256, 6058, 5135, 30564, 12367, 5135, 20718, 21643, 44, 5135, 17161,
 45, 44673, 16773, 6989, 8387, 7020, 48423, 8200, 3385, 3378, 100, 97, 22647, 8070, 7782, 5034, 2994, 46, 4194, 100, 259, 14482, 256, 1453, 100, 1201, 100, 1057, 2962, 1356, 22616, 121,
 8941, 44, 1083, 12201, 41019, 19243, 3105, 14863, 36, 16701, 5554, 36, 21410, 44, 37117, 5554, 18250, 258, 115, 3344, 114, 1333, 21691, 121, 44, 1453, 100, 21026, 7980, 105, 1560, 32361,
 44, 36823, 44, 3062, 9507, 32361, 1453, 100, 49357, 46, 16726, 5554, 108, 100, 5135, 22647, 4642, 4132, 48370, 14576, 17369, 5554, 2657, 1560, 5554, 27721, 4560, 97, 6993, 3058, 4132, 14914,
 259, 4954, 18653, 44, 256, 8161, 3438, 1200, 10584, 4132, 28344, 6058, 5135, 1566, 7782, 1453, 115, 45, 43549, 44, 46299, 5034, 2652, 1026, 46, 259, 5671, 10584, 10516, 33487, 1453, 260, 115,
 258, 43549, 46, 9556, 101, 10340, 260, 1560, 46, 258, 11223, 22762, 258, 115, 10340, 260, 4268, 8638, 46, 258, 65, 34230, 7105, 11159, 7782, 14488, 10218, 48423, 2657, 1050, 1197, 97, 25247,
 44, 5034, 260, 115, 258, 10340, 260, 4602, 14929, 4229, 44, 258, 256, 16726, 31192, 5554, 10457, 46, 259, 84, 775, 10340, 97, 9995, 101, 3058, 4132, 73, 258, 109, 19073, 5607, 2819, 1453, 100,
 97, 9995, 101, 22675, 44, 1453, 100, 73, 3835, 5554, 2657, 1560, 46, 2598, 100, 22166, 19972, 43050, 73, 19903, 3062, 6058, 5554, 5135, 30564, 44, 73, 258, 109, 871, 1560, 44, 73, 258, 109,
 871, 1560, 73, 258, 109, 7103, 40172, 5554, 3921, 46, 73, 3835, 5554, 2657, 49357, 1560, 7782, 8257, 7094, 1453, 100, 49357, 30007, 19718, 1065, 8257, 7094, 46, 2598, 100, 4132, 258, 115,
 19548, 14863, 1026, 46, 2598, 100, 73, 20710, 97, 25247, 7782, 40593, 10516, 807, 12570, 5554, 4132, 46, 256, 16726, 44, 11935, 44, 48423, 49681, 1065, 22647, 4867, 44, 2041, 118, 40172, 100,
 1800, 1208, 111, 22787, 8070, 115, 25718, 72, 83, 78, 1453, 100, 2727, 19203, 46, 5426, 1802, 44, 8550, 8070, 10340, 5135, 1151, 10584, 48370, 258, 115, 19019, 6058, 5034, 2994, 44, 1453,
 100, 48370, 37175, 1560, 871, 5554, 3316, 731, 5135, 24593, 115, 115, 44, 20579, 8807, 31717, 5034, 43155, 19795, 22272, 8885, 46, 3235, 781, 14020, 44, 16726, 1453, 100, 5034, 2819, 45,
 42700, 44, 17795, 42956, 1364, 45, 13537, 110, 10555, 30417, 79, 7588, 105, 1453, 44, 19468, 33518, 5135, 3816, 1151, 20811, 44, 10273, 1276, 1739, 9507, 30417, 22108, 79, 7588, 105, 1453,
 10999, 46]


DECODED SYMBOLIC TOKENS FROM THE INTEGERS:  One day after Serena Williams made headline s for wea ring a " cat suit " on the court at the French Open, the 23 - time Grand Sl am champion has
 la un che d a fashion line of her own. Calle d " Serena " an d sol d ex clu si vel y online, pi ec es ran ge from $ 35 to $ 250, according to Women ' s Wea r Da il y, an d include athle i sure
 wear, dresses, out er wear an d more. Williams to l d the fashion publication that she want ed to be sure to crea te a clo thing that felt " practi cal, " but al so one that focused on the
concept of an s - word, including her na me. " Every one can have an ' s ' word. Min e is ' sure. ' My mom ' s is ' stead fast. ' A really good friend of mine who has be en through a lot, her
 ' s ' is ' sur viv or, ' " Williams explained to WWD. " T his is a hug e thing that I ' m doing right now an d a hug e undertaking, an d I need to be sure. An d sometimes even when I walk out
 on to the  court, I ' m not sure, I ' m not sure I ' m go ing to win. I need to be more sure of my self an d more confi dent in my self. An d that ' s coming from me. An d I feel a lot of
 people can re late to that. " Williams, 36, has worked in fashion before, ha v ing d ev el o ped line s with H S N an d Ni ke. How ever, this line is the first one she ' s done on her own,
 an d she made  sure not to ru sh the proce s s, especi ally given her evolving perso nal life. La st September, Williams an d her now - husband, Red dit co - fou n der Alexis O han i an,
welco med the ir first child, da ug ht er Alexis Olympia O han i an Jr.
```

[**NOTE: The tokens shown above are rejoined into whole words ultimately. In babyGPT I have used the simplest possible strategy for facilitating that — using the underscore symbol '_' to separate the words in the input text.**]

- Shown in the next three slides is the implementation code for the class `TrainTokenizer` in the babyGPT module.

TrainTokenizer Class in the babyGPT Module

```
class TrainTokenizer:
    """
    See the documentation page at https://engineering.purdue.edu/kak/distBabyGPT/
    """
    def __init__(self, corpus_directory, target_vocab_size=50000):
        import babyGPT
        version_str =  babyGPT.__version__
        version_str = version_str.replace(".", "")
        self.tokenizer_json_stem   =   version_str + "_babygpt_tokenizer_"
        self.corpus_dir = corpus_directory
        self.unk_token = "[UNK]"                    # token for undecipherable bytes
        self.spl_tokens = ["<UNK>", "<SEP>", "<MASK>", "<CLS>"]
        self.target_vocab_size = target_vocab_size
        ##  Since we are assuming utf-8 encoding of the text corpus, we already have
        ##  the mappings between the numbers 0 through 255 and their corresponding
        ##  tokens as would be yielded by calling the Python function chr() on the
        ##  integers between 0 and 255.  [For example, chr(255) returns the character
        ##  'ÿ'. What that means is that 255 is the Unicode code point for this symbol.]
        self.next_index_available = 256
        ##  I use "testing_iter" for producing the intermediate results during training:
        self.testing_iter = 0

    def train_tokenizer(self):
        """
        See the documentation page at https://engineering.purdue.edu/kak/distBabyGPT/
        """
        def word_as_num_seq(word):                                                                                  ## (A)
            for char in list(word):
                if char not in char_to_num_dict:
                    char_to_num_dict[char] = self.next_index_available
                    merge_rules_dict[ self.next_index_available ] = char
                    self.next_index_available += 1
            return [char_to_num_dict[char] for char in list(word) ]

        def get_str_token( num ):
            """
            Note that merge_rules_dict is what becomes the vocab eventually.  We make the
            conversion by reversing the <key,num> pairs in merge_rules_dict.
            """
            if num in num_to_char_dict:
                return num_to_char_dict[num]
            elif num in merge_rules_dict:
                return merge_rules_dict[num]
            else:
                sys.exit("\n\n[get_str_token]  merge_rules_dict has no merge rule for the int token %d\n\n" % num)

        def subword_for_num_seq( num_seq ):
            subword = ""
            for num in num_seq:
                if num in num_to_char_dict:
                    subword += chr(num)
                elif num in merge_rules_dict:
                    subword += merge_rules_dict[num]
                else:
                    sys.exit("\n\n[subword_for_num_seq] merge_rules_dict has no merge rule for the int token %d\n\n" % num)
            return subword

        def update_tokenizer_dict( tokenizer_dict, most_frequent_pair, new_token_as_num ):
            new_tokenizer_dict = {word : [] for word in tokenizer_dict}
            for word in tokenizer_dict:
                str_rep = ",".join(str(i) for i in tokenizer_dict[word])
                to_be_replaced_pair =  r"\b" +  ",".join(str(i) for i in most_frequent_pair) + r"\b"
                replacement = str(new_token_as_num)
                output_str= re.sub(to_be_replaced_pair, replacement, str_rep)
                new_tokenizer_dict[word]  =  [int(i) for i in output_str.split(",")]
            return new_tokenizer_dict
```

(Continued on the next slide .....)

TrainTokenizer Class in babyGPT (contd.)

(..... continued from the previous slide)

```
def find_best_ngram_and_update_word_tokens_dict(tokenizer_dict):                                                    ## (E)
    all_consec_pairs_dict = { word : list( zip( tokenizer_dict[word], tokenizer_dict[word][1:] ) ) for word in tokenizer_dict }    ## (F)
    all_consec_triples_dict =   { word : list( zip( tokenizer_dict[word], tokenizer_dict[word][1:],  tokenizer_dict[word][2:] ) )   ## (G)
                                                                                          for word in tokenizer_dict }
    all_consec_quads_dict   =   { word : list( zip( tokenizer_dict[word], tokenizer_dict[word][1:],  tokenizer_dict[word][2:],     ## (H)
                                                                    tokenizer_dict[word][3:] ) ) for word in tokenizer_dict }
    all_consec_all_ngrams_dict = {}
    for word in all_consec_pairs_dict:
        if word in all_consec_triples_dict and  word in all_consec_quads_dict:
            all_consec_all_ngrams_dict[word]  =  all_consec_pairs_dict[word] + all_consec_triples_dict[word] + all_consec_quads_dict[word]
        elif word in all_consec_triples_dict:
            all_consec_all_ngrams_dict[word]  =  all_consec_pairs_dict[word] + all_consec_triples_dict[word]
        else:
            all_consec_all_ngrams_dict[word]  =  all_consec_pairs_dict[word]
    all_consec_all_ngrams_dict  =   {word : all_consec_all_ngrams_dict[word] for word in all_consec_all_ngrams_dict
                                                                          if len(all_consec_all_ngrams_dict[word]) > 0}
    most_frequent_ngram = list(Counter( list( itertools.chain(*all_consec_all_ngrams_dict.values()) ) ).keys()) [0]
    string_for_merges_array = "%s %s" % (get_str_token(most_frequent_ngram[0]), get_str_token(most_frequent_ngram[1]))
    merges.append( string_for_merges_array )
    subword_for_most_frequent_ngram  =  subword_for_num_seq( most_frequent_ngram )
    if self.testing_iter % 100 == 0:
        print("\n\n[testing_iter: %d] Will merge the following subwords for the new most frequently occurring subword:" % self.testing_iter)
        if len(most_frequent_ngram) == 2:
            print("%s    %s" % (get_str_token(most_frequent_ngram[0]), get_str_token(most_frequent_ngram[1])))
        elif len(most_frequent_ngram) == 3:
            print("%s    %s    %s" % (get_str_token(most_frequent_ngram[0]), get_str_token(most_frequent_ngram[1]),
                                                                          get_str_token(most_frequent_ngram[2] )))
        else:
            print("%s    %s    %s    %s" % (get_str_token(most_frequent_ngram[0]), get_str_token(most_frequent_ngram[1]),
                                        get_str_token(most_frequent_ngram[2]), get_str_token(most_frequent_ngram[3]) ))
        print("\n\nAdding to tokenizer vocab: ",  subword_for_most_frequent_ngram)
    merge_rules_dict[self.next_index_available] = subword_for_most_frequent_ngram
    new_tokenizer_dict = update_tokenizer_dict( tokenizer_dict, most_frequent_ngram, self.next_index_available )
    if self.testing_iter % 100 == 0:
        print("\n\n[testing_iter: %d] UPDATED tokenizer dict:\n" % self.testing_iter)
        for word in new_tokenizer_dict:
            print("%s  =>  %s" % (word, str( [get_str_token(i) for i in new_tokenizer_dict[word]] )))
    self.next_index_available += 1
    return new_tokenizer_dict                ## End of find_best_ngram ....

seed_value = 0
random.seed(seed_value)
os.environ['PYTHONHASHSEED'] = str(seed_value)
dir_textfiles =  self.corpus_dir
##  The dict defined in the next statement stores the mappings from the symbolic tokens to integers that
##  represent them. For the number range 0 through 255, the mappings stored are those that are returned
##  by calling chr() on the Unicode numbers between 0 and 255. Subsequently, as larger tokens are
##  constructed by merging the "sub-word" tokens, we add those tokens and their associated numbers to
##  this dict:
char_to_num_dict = { chr(num) :  num for num in range(256) }
num_to_char_dict = { num : chr(num) for num in range(256) }
merge_rules_dict = { i : "" for i in range(256, self.target_vocab_size) }
##  I store all pairwise merges in the following array.  Each element of this array is a string
##  that looks like  "str1 str2" where str1 and str2 are the two subwords that are to be merged together.
merges = []
text = ""
if os.path.exists(dir_textfiles):
        textfiles = glob.glob(dir_textfiles + "/*")
        print("\n\nNumber of text files: ", len(textfiles))
        for filedoc in textfiles:
            if os.path.isfile(filedoc):
                with open( filedoc, encoding='utf8', errors='ignore' ) as f:
                    text += f.read()
print("\n\nlength of the text string: ", len(text))
##  We will store the merged char mappings for the new tokens in this dictionary
merged_symbols_dict = {num : None for num in range(256, self.target_vocab_size) }
```


(Continued on the next slide .....)

TrainTokenizer Class in babyGPT (contd.)

(..... continued from the previous slide)

```
all_words = text.split()
print("\n\nNumber of words in the list 'all_words': ", len(all_words))
print("\n\nfirst 100 entries in all_words: ", all_words[:100])
##  We need the word frequencies BECAUSE we need to find the most frequently occurring token pair in the corpus.
##  That is, for a given token pair, we need to know the number of words in which that pair occurs.
words_with_counts = Counter(all_words)
unique_words = list(set( all_words ))
print("\n\nnumber of UNIQUE words: ", len(unique_words))
print("\nfirst 100 UNIQUE words: ", unique_words[:100])
word_tokens_dict =  { word : word_as_num_seq(word) for word in unique_words }                    ##  Initialization of word_tokens_dict
print("\n\nIterative learning of the merge rules:\n\n")
for i in range(256):
    merge_rules_dict[i] = chr(i)           ## the char returned by the function chr(i) is the char under utf-8 encoding
while self.next_index_available <= self.target_vocab_size:
    self.testing_iter += 1
    new_word_tokens_dict = find_best_ngram_and_update_word_tokens_dict( word_tokens_dict )
    if self.testing_iter % 100 == 0:
        print("\n\n[testing_iter = %d] Size of the tokenizer vocab: " % self.testing_iter,  self.next_index_available-1)
    word_tokens_dict = new_word_tokens_dict
    if self.testing_iter % 5000 == 0:
        merge_rules_dict[self.target_vocab_size + 1] = "<UNK>"
        vocab = {val : key for (key,val) in merge_rules_dict.items()}
        print("\n\n[testing_iter: %d] vocab: " % self.testing_iter, vocab)
        print("\n\n[testing_iter: %d] merges array:" % self.testing_iter, merges)
        vocab_and_merges =  {"version" : "1.0",
                             "truncation" : None,
                             "padding" : None,
                             "added_tokens" : [
                                  {"id" : self.target_vocab_size+1,
                                   "content" : "<UNK>",
                                   "single_word": False,
                                   "lstrip": False,
                                   "rstrip": False,
                                   "normalized": False,
                                   "special": True,
                                  },
                             ],
                             "normalizer": None,
                             "pre_tokenizer": {
                                 "type": "Whitespace"
                             },
                             "model" :  {"type": "BPE", "dropout" : None, "vocab" :  vocab,  "merges" : merges } }
        with open(self.tokenizer_json_stem + str(self.testing_iter) + ".json", "w") as outfile:
            json.dump(vocab_and_merges, outfile, indent=4)
merge_rules_dict[self.target_vocab_size + 1] = "<UNK>"
vocab = {val : key for (key,val) in merge_rules_dict.items()}
print("\n\nvocab: ", vocab)
print("\n\nmerges array:", merges)
vocab_and_merges =  {"version" : "1.0",
                     "truncation" : None,
                     "padding" : None,
                     "added_tokens" : [
                          {"id" : self.target_vocab_size+1,
                           "content" : "<UNK>",
                           "single_word": False,
                           "lstrip": False,
                           "rstrip": False,
                           "normalized": False,
                           "special": True,
                          },
                     ],
                     "normalizer": None,
                     "pre_tokenizer": {
                         "type": "Whitespace"
                     },
                     "model" :  {"type": "BPE", "dropout" : None, "vocab" :  vocab,  "merges" : merges } }
with open(self.tokenizer_json_stem + str(self.testing_iter) + ".json", "w") as outfile:
    json.dump(vocab_and_merges, outfile, indent=4)
```

TrainTokenizer Class in babyGPT (contd.)

- The workhorse of the code shown above is the function `find_best_ngram_and_update_word_tokens_dict(tokenizer_dict)` defined in Line (E). Note the departure from the common practice here: In addition to measuring the word frequencies associated with pairs of subwords, it also measures the frequencies associated with the trigrams and quadra-grams of subwords as shown in Lines (F), (G), (H).
- The dictionary `merge_rules_dict` at the top-level [and the same in the form of `tokenizer_dict new_tokenizer_dict` in the support functions] plays a central role in the logic of the script. The dictionary has the mappings from new subwords created by merging and the integer indexes assigned to them.
- Before the BPE logic kicks in, the function `word_as_num_seq()` defined in Line (A) is used to represent each word in the training file by a sequence of Unicode values. These numbers are always be in the range 0-255.

Well-Known Tokenizers of the Day for LLMs

- The following three tokenizers are commonly used for text processing in the modern context:
  - Byte-Pair Encoding (BPE),
  - WordPiece, and
  - SentencePiece
- Of these three, the core logic in the first two is based on discovering the merge rules for combining pairs of base symbols of the tokenizer into subwords and subsequently concatenating the subwords into longer subwords until the size of the tokenizer vocabulary has reached its user-specified value. This is the same logic as presented in babyGPT's TrainTokenizer class shown in this section.
- Common to both of the first two algorithms is pre-tokenization of the training corpus using the white-space as the separator between adjacent words.

Well-Known Tokenizers (contd.)

- However, there exist several languages in the world that do not lend themselves to such whitespace -based word separation. Tokenization for such languages works better with the third third alternative mentioned on the previous slide.
- The third alternative, SentencePiece, takes one step further the logic of a tokenizer like BPE and gets rid of the need for whitespace-based pre-tokenization of the text. Whitespace is treated as just another Unicode character and, for processing during tokenization, is escaped with the underscore symbol '_' (Unicode: U+2581).

Outline

BERT Architecture

- The basic architecture of BERT is exactly the same as that of the Master Encoder side of the figure shown on Slide 41 of my Week 13 lecture. That slide presented both the Master Encoder and the Master Decoder in a Transformer-based machine translation framework. The figure on the next slide here is just the Master Encoder part of the Week 13 figure as the neural architecture for BERT.

- Since the embedding vectors play a central role in the calculation of Attention, you need to know right off the bat that the size of the embeddings in BERT is either 768 or 1024 depending on which version of BERT you are using. There are two versions as mentioned on Slide 33.

- In the BERT paper, the symbol H represents the size of the embedding vectors.

  [For some reason, that paper refers to embeddings as "Hidden vectors" or "Hidden states." In retrospect, it sounds bizarre to be using those names for the embeddings — especially because of the unique (and semantically rich) role played by the "hidden state" in Recurrent Neural Networks that we covered in my Week 12 lecture.]

BERT Architecture (contd.)

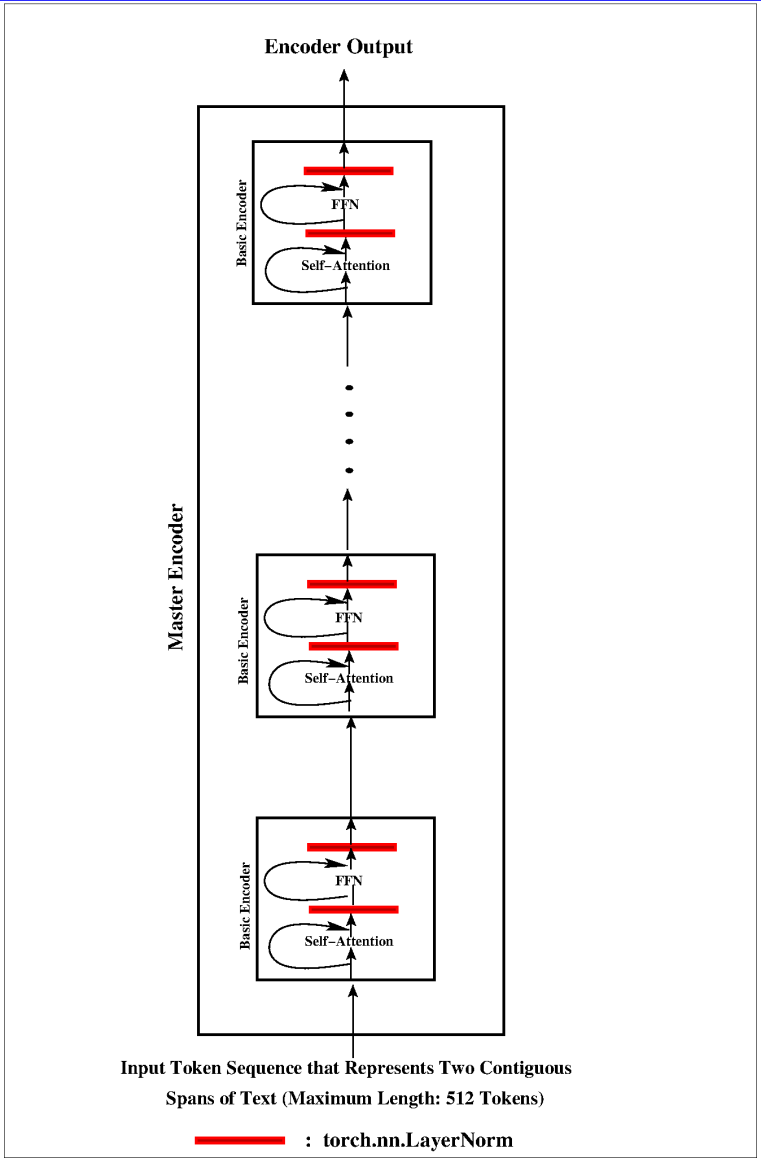


Figure: The BERT architecture, as shown above, is exactly the same as the Master Encoder part of the overall architecture on Slide 41 of my Week 13 slides.

BERT Architecture (contd.)

- What I have shown as Basic Encoders in the figure on the previous slide are called `Transformer Blocks` in the BERT architecture.
- The two versions of BERT are denoted $BERT_{base}$ and $BERT_{large}$.
- In $BERT_{base}$, the number of Transformer Blocks is 12 and the number of Attention Heads is also 12. On the other hand, in $BERT_{large}$, the number of Transformer Blocks is 24, and the number of Attention Heads 16.
- Another point of difference between $BERT_{base}$ and $BERT_{large}$ is the size of the embeddings. In keeping with the numbers presented on Slide 31, the embeddings are of size 768 for $BERT_{base}$ and 1024 for $BERT_{large}$.
- Also note that the number of learnable parameters in $BERT_{base}$ is 110 million and the same in $BERT_{large}$ is 340 million.

BERT Input Format

- What makes BERT so versatile, and as to why it continues to be used for domain-specific LLMs, is the nature of the "formatting" of the input on which it is trained.
- As shown in the caption at the bottom of the figure in Slide 32, the input consists of a pair of token sequence, with each token sequence representing a span of continuous text. The two spans at the input could represent, say, a (*Question*, *Answer*) pair, a (*Sentence*, *Next_Sentence*) pair, etc.
- Although, for most applications, you would want to feed a pair of token sequences at its input, BERT would also accept single token sequence.
- Feeding just one token sequence at the input is important for one of the two modes — the mask based mode — for the generative training of BERT. More on that later.

BERT Input Format (contd.)

- There is an upper limit on the length of a token sequence: 512. This is dictated by the role played by the parameter `max_seq_length` in the constructor of the `SelfAttention` class for the Transformers in my Week 13 lecture.

  [Since people have gotten used to seeing long narratives that the LLMs can now return in response to human supplied prompts, you are likely to wonder if using 512 as a limit on the input to the BERT network is way too limiting for modern times. The answer to that is "Not necessarily." The most basic goal of an LLM is to learn in an unsupervised manner the fundamental language continuity properties, from word-to-word, clause-to-clause, sentence-to-sentence, paragraph-to-paragraph, and so on. BERT can do a pretty good job of learning the word-to-word, clause-to-clause and sentence-to-sentence continuity properties. Consider, for example, the two input sentences in a Question/Answer framework. Ignoring the special tokens for a moment, you have 256 tokens for the Question part and the same number for the Answer part. Since the tokenizers used for LLMs do not break up the most frequently used words, so, for the sake of argument, let's assume our sentences are composed of only the common words that may not get the ax from the tokenizer. Next consider that, in English, a typical sentence has between 20 and 30 words. So a span of 256 words is likely to capture several sentence-to-sentence transitions, including, obviously, word-to-word and clause-to-clause transitions. You might still ask: What about the paragraph-to-paragraph continuity properties? Those can be taken care of in a supervised extension of BERT as discussed later. Supervised extensions of BERT are not computationally demanding.]

- The two token sequences at the input are separated by a special token denoted `SEP`. More precisely, each of the two token sequences ends in the token `SEP`. Obviously, `SEP` is represented by a learnable embedding of its own.

BERT Input Format (contd.)

- For convenience, we refer to the text sequence that comes before the SEP token as "Sentence-A" and the text sequence that comes after the separator token as "Sentence-B".
- A special learnable per-class token, called the classification token, is prepended to the token sequence for each sentence pair input. This token is denoted [CLS]. [**The class token is useful for applications, such as sentiment analysis, in which you want to classify text. When solving a classification problem, you would feed the embedding corresponding to the [CLS] input at the output of the Master Encoder into an MLP for classification.**]
- The figure on the next slide shows the placement of the [CLS] token for the imaginary case when the two-sentence input is limited to 10 tokens.
- In order to give the network a deeper sense of which token belongs to which of the two input sequences, we add to the embedding of each input token a learnable token, designated E_A and E_B in the figure, that indicates the sentence to which the token belongs. Both these tokens will have their own embedding vector representations.

BERT Input Format (contd.)

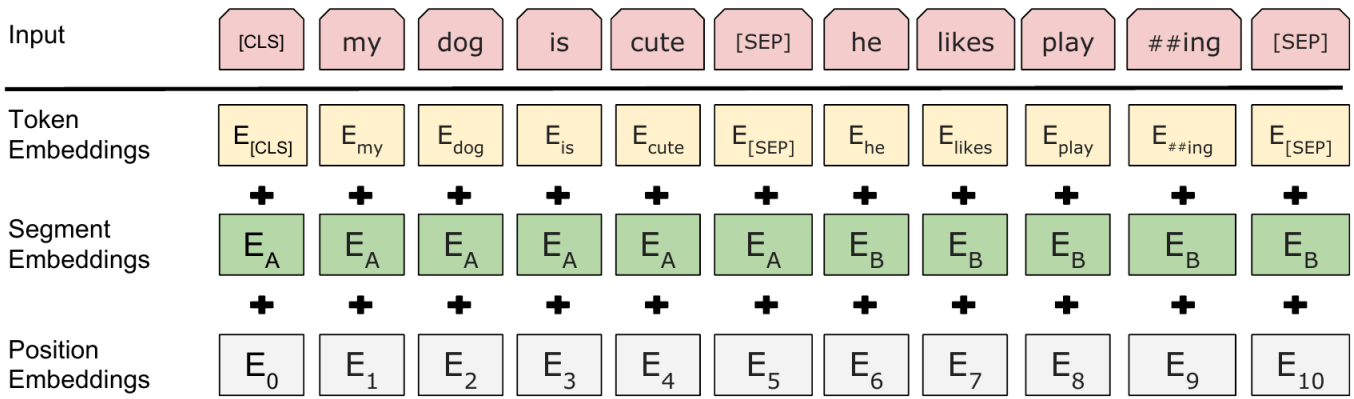


Figure: This figure is taken from the original BERT paper

BERT Input Format and the Tokens

- And, as in the Transformer implementations you saw in my Week 13 lecture, the network also needs to be acquire a sense of where exactly an input token belongs in relation to the other tokens in a sequence.
- As shown in the figure on the previous slide, we designate a per-position token (more accurately, the learnable embedding for a per-position token) for that purpose. The embeddings for the position tokens are added to the embeddings for the input sentence tokens.
- Since BERT allows for a maximum of 512 for the number of tokens at the input, so you will have a total of 512 position tokens, each represented by its embedding.
- BERT uses the WordPiece tokenization algorithm for segmenting the words into tokens. To start with, WordPiece initializes the token vocabulary with the individual characters in the words. It then progressively merges the characters on the basis of the joint probabilities of the merged symbols vis-a-vis their marginal

The Tokenizer in BERT

- The end result is that the words that occur frequently are left alone and those that occur relatively rarely are broken into subwords.
- The subwords generated in this manner can be identified in the output of the tokenizer by the prefix '##', as you will see in the tokenizer output on Slides 41 and 42. The mark '##' is called "*continuing_subword_prefix*" in the parlance of the BERT tokenizer. By the way, there is also an upper limit of 100 characters in a word that is to be subject to tokenization.
- As you would expect, the decomposition of the rarer words in this manner leads to the much more frequently occurring subwords, as demonstrated on Slides 41 and 42.
- One of my motivations for the tokenizer demo on Slides 41 and 42 is to dispel a rather commonly held misconception that the main job of the tokenizer to break up long words into smaller subwords.

The Tokenizer in BERT (contd.)

- The next two slides are a demo of the WordPiece tokenizer in BERT. You will need to install the Python package `transformers` with a command like "`sudo pip install transformers`" if you want to execute the code yourself. Installing `transformers` will automatically pull in the following packages: `safetensors`, `regex`, `huggingface-hub`, `tokenizers`, `transformers`.

- And when you execute the code in this demo, it will further download the files

```
vocab.txt
tokenizer.json
config.json
tokenizer_config.json
```

- As you would expect, the tokenizer vocabulary (about 30,000+ tokens) is in the file `vocab.txt`. The file `tokenizer.json` has the integer indexes for the vocab entries. If want to know where these files are stored, execute the following command at the top of your directory tree:

```
find . -name vocab.txt
```

BERT Tokenizer (contd.)

```
##  demo_bert_tokenizer.py

  from transformers import BertTokenizer                                                    ## (1)

  tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")                            ## (2)

  tokens = tokenizer.tokenize("All work and no play makes Jack a dull boy!")                ## (3)
  print(tokens)
  ##       ['all', 'work', 'and', 'no', 'play', 'makes', 'jack', 'a', 'dull', 'boy', '!']   ## (4)

  tokens = tokenizer.tokenize("Song in Mary Poppins: It's supercalifragilisticexpialidocious")  ## (5)
  print(tokens)
  ##       ['song', 'in', 'mary', 'pop', '##pins', ':', 'it', "'", 's', 'super', '##cal', '##if',
  ##                        '##rag', '##ilis', '##tic', '##ex', '##pia', '##lid', '##oc', '##ious']   ## (6)

  FILE = open("bert_vocab.txt")                                                             ## (7)
  all_tokens = FILE.read().splitlines()  ## readlines() instead will retain '\n' at end of each entry  ## (8)
  FILE.close()

  all_tokens.sort( key=lambda x: len(x), reverse=True )                                     ## (9)

  print("\n\nThe longest ten entries in BERT tokanizer vocab: ", all_tokens[:10])          ## (10)
  ## ['telecommunications', 'interdisciplinary', 'telecommunication', 'responsibilities', 'autobiographical',
  ##    'intercontinental', 'entrepreneurship', 'unconstitutional', 'northamptonshire', 'characterization']
  ##                                                                                       ## (11)

  print("\n\nThe shortest ten entries in BERT tokanizer vocab: ", all_tokens[-10:])       ## (12)
  ##                              ['!', '(', ')', ',', '-', '.', '/', ':', '?', '~']       ## (13)
```

(Continued on the next slide .....)

BERT Tokenizer (contd.)

(..... continued from the previous slide)

```
print("\n\nExperiments with the 'tokenizer.json' file that has
                                    the mappings from the tokens to the integer indexes:\n\n")
import json
FILE = open("tokenizer.json")                                                                    ## (14)
all_entries = json.load(FILE)                                                                    ## (15)
for key in all_entries['model']['vocab']:                                                        ## (16)
    print("%s : %s\n" % (key, all_entries['model']['vocab'][key]))                               ## (17)
inverse_look_up = {v:k for k,v in all_entries['model']['vocab'].items()}                         ## (18)
print("\n\nshowing the token for the index 102: ", inverse_look_up[102])   ## [SEP]              ## (19)
```

- As you can tell from the code starting in Line (14), the mappings from the tokens to the integer index values are stored in the JSON file `tokenizer.json`. We load the JSON file into our script in Line (16) and access the integer index values for the tokens at the nested dictionary at the key `vocab` of the dictionary that is at the key `model`.
- Lines (16) and (17) will print out the integer index values for all 30,000+ tokens that BERT uses.
- We invert the indexes in Line (18) and query the inverted index for the token associated with the integer 102.

Outline

Pre-Training BERT

- The Generative Pre-Training of BERT is carried out with respect to the following two tasks:
  - Masked Language Modeling (MLM)
  - Next Sentence Prediction (NSP)

  .
- Regarding MLM, during unsupervised training, you mask out a certain percentage of the tokens at the input to the BERT network shown in Slide 32. The tokens that are masked are selected at random.
- Subsequently, in keeping with the display in Slide 14, the embedding vectors that correspond to the masked tokens at the input are fed into `nn.Softmax` over the 30,000+ vocabulary of the BERT tokenizer to find the ML (maximum-likelihood) predictions for the masked tokens. The loss thus calculated is backpropagated, as you would expect. BERT pre-training calls for randomly masking 15% of the input tokens.

Pre-Training BERT (contd.)

- And, regarding NSP, again during unsupervised training, the goal is to feed `(sentence-A, sentence-B)` pairs into the BERT network, with 50% of the time `sentence-B` being the actual next sentence to `sentence-A`, and the other 50% of the time a randomly chosen sentence.
- The goal is to carry out a classification task with a loss function like the Binary Cross-Entropy Loss (`nn.BCELoss`) using the labels `isNext` and `notNext`.

Datasets Used for Pre-Training BERT

- The unsupervised pre-training for BERT has been carried out using the following two datasets:
  - The `BookCorpus` that consists of about 7,000 self-published book. The dataset consists of roughly 985 million words. According to Wikipedia, this dataset is not longer available for training.
  - The English Wikipedia with its roughly 2.5 Billion words.

Outline

RoBERTa as a More Powerful BERT

- Shortly after BERT was announced, the following publication "*RoBERTa: A Robustly Optimized BERT Pretraining Approach*" by Liu et al. improved upon some important aspects of BERT to create a more robust pre-training framework:

  `https://arxiv.org/pdf/1907.11692.pdf`

- The authors claim that, their version of BERT outperforms on the same competition datasets that were used for evaluating BERT. Whether or not the difference in the performance numbers is significant, that's for you to decide. As far as what I can tell, BERT still dominates the scene (for those who have not switched over to OpenAI's GPT based models.).

- Note that in the basic architecture of RoBERTa, the formatting of the input data, the unsupervised training protocol, etc., are the same as for BERT. However, RoBERTa was trained with a much larger dataset.

RoBERTa (contd.)

- From my perspective, an important difference between BERT and RoBERTa is that the latter uses the BPE (Byte Pair Encoding) algorithm for tokenization. It is the same tokenizer that is used for the GPT models from OpenAI.
- Here are two significant differences between BPE and the WordPiece tokenizer used in BERT:
  - While, for the most part, WordPiece uses the ASCII characters as the basic units for merging in order to form the subwords, BPE use the bytes directly. This allows BPE to be more general with respect to the different languages. [This is not to imply that WordPiece does not recognize other languages. If you direct the output of the script I showed on Slides 41 and 42 into a text file and scroll the file, you will see the basic symbols from practically all the languages in the 30,000+ vocabulary of the WordPiece tokenizer.]
  - BPE elicits the help of a pre-tokenizer to count the frequencies of all the words in the corpus. It subsequently uses those frequencies for merging the bytes into subwords on the basis of the frequencies of the merged bytes vis-a-vis the frequencies the subwords that were merged.

Outline

But First a Demo of the Power of Pre-Training

- The pre-trained BERT is frequently customized by supervised fine-tuning for applications that involve question answering, sentiment analysis, and that call for language inference.
- Before I describe how you can customize BERT, it's interesting to note that BERT packs a lot of punch even without any further fine-tuning. I illustrate this with the script on the next slide.
- The script is about evaluating which of the two sentences `sentence_B` or `sentence_C` is a better continuation as the next sentence for `sentence_A`

  [If you are new to the US, you are probably thinking that Purdue Pharma in `sentence_B` must refer to the famous Pharmacy department at Purdue University. Purdue Pharma, a pharmaceutical company owned by a family known as Sacklers, is the maker of the painkiller drug OxyContin. It was believed that this drug caused what's known as the Opoid Crisis in the United States with hundreds of thousands of people developing a strong addiction to the drug. Highly questionable business practices by the company were considered to have contributed to the problem.]
- When you execute the script, you get the following answer:

```
CASE 1:
prediction loss:  tensor(1.0133e-05, grad_fn=<NllLossBackward0>)
True Pair
CASE 2:
prediction loss:  tensor(2.7418e-06, grad_fn=<NllLossBackward0>)
True Pair
```

A Demo of the Power of Pre-Training (contd.)

```
## nsp1.py

from transformers import BertTokenizer, BertForNextSentencePrediction                    ## (1)
import torch

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')                           ## (2)
model = BertForNextSentencePrediction.from_pretrained('bert-base-uncased')               ## (3)

sentence_A = "Purdue University is famous for its engineering program \
                                          at both graduate and undergraduate levels."    ## (4)
sentence_B = "The Supreme Court heard arguments over a bankruptcy deal \
              for Purdue Pharma that would give billions of dollars to \
                                                 those harmed by the opoid epidemic."    ## (5)
sentence_C = "Purdue has an enrollment of over 50000 students at its   \
                                                      West Lafayette campus alone."      ## (6)
print("\n\nCASE 1:")                                                                     ## (7)
##  'tokenized' is a dictionary with keys: 'input_ids', 'token_type_ids', 'attention_mask'  ## (8)
tokenized = tokenizer(sentence_A, sentence_B, return_tensors='pt')                       ## (9)
labels = torch.LongTensor([0])                                                           ## (10)
predict = model(**tokenized, labels=labels)                                              ## (11)
print("\n\nprediction loss: ", predict.loss)                                            ## (12)
prediction = torch.argmax(predict.logits)                                                ## (13)
if prediction == 0:                                                                      ## (14)
    print("True Pair")                                                                   ## (15)
else:                                                                                    ## (16)
    print("False Pair")                                                                  ## (17)
```

(Continued on the next slide .....)

A Demo of the Power of Pre-Training (contd.)

(..... continued from the previous slide)

```
print("\n\nCASE 2:")                                                    ## (18)
tokenized = tokenizer(sentence_A, sentence_C, return_tensors='pt')     ## (19)
labels = torch.LongTensor([0])                                         ## (20)
predict = model(**tokenized, labels=labels)                            ## (21)
print("\n\nprediction loss: ", predict.loss)                           ## (22)
prediction = torch.argmax(predict.logits)                              ## (23)
if prediction == 0:                                                    ## (24)
    print("True Pair")                                                 ## (25)
else:                                                                  ## (26)
    print("False Pair")                                                ## (27)
```

- The good news is that it declares that `sentence_C` is significantly closer to `sentence_A` than `sentence_B`.
- The bad news is that it declares both as the true matches. It is to rectify these issues you need to carry out the sort of supervised learning based customization described next.

A Demo of the Power of Pre-Training (contd.)

- **The good news mentioned on the previous slide speaks to the power of unsupervised generative pre-training. Think about it: Without any supervision at all, just by letting loose your neural network on the publicly available data in the internet, it can figure out how to create a continuous narrative for you on a topic for which it is prompted.**
- About the code shown on the previous two slides, the statement in Line (3) will download the pre-trained BERT model as a 440 MB archive consisting of a `safetensors` archive.
- safetensors is the new recommended way to store a model, especially one that is meant for distribution over the internet. In the past, the practice has been (and still continues to be) to store the model weights in a ".bin" file using Python's `pickle` function. But Google folks who created `safetensors` say that it is relatively easy to embed malicious code in a pickled archive. For further information, here is the GitHub link where can find out more about safetensors:

`https://github.com/huggingface/safetensors`

Fine-Tuning BERT

- Getting to the main subject of this section, according to the authors of BERT, "compared to pre-training, fine-tuning is relatively inexpensive. All of the results in the paper can be replicated in at most 1 hour on a single Cloud TPU, or a few hours on a GPU, starting from the exact same pre-trained model."

- For answering questions, the supervised fine-tuning would consist of feeding (Sentence-A, Sentence-B) pairs at the input, where Sentence-A would be the question and Sentence-B (possibly in the form of a short para) the answer.

- At the same time, you would also create sentence pairs in which the second sentence is NOT the answer. The target would be the binary labels `isAnswer` and `isNotAnswer`. In this manner, this would boil down to a binary classification problem with the Binary Cross-Entropy Loss as provided by `nn.BCELoss`.

Fine-Tuning BERT

- Fine tuning of the pre-trained BERT is even easier for a Sentiment Analysis application. Now the input to the BERT network would consist of (Sentence-A, ∅) pairs and the prediction made through the embedding vector for the class-token [CLS] would be either True or False depending on the ground-truth tag for the text in Sentence-A.

- More generally, though, the ability of BERT to predict the next sentence can be leveraged into several different ways of fine-tuning:

  MNLI (Multi-Genre Natural Language Inference): This is a large-scale entailment classification task. Given a pair of sentences, the "goal is to predict whether the second sentence is an *entailment*, *contradiction*, or *neutral* with respect to the first sentence."

  QQP (Quora Question Pairs): This is a binary classification task in which the goal is to determine whether the two questions, as represented by Sentence-A and Sentence-B, are semantically equivalent.

  QNLI (Question Natural Language Inference): This is a version of the Stanford Question Answering Dataset. In this case, in each sentence pair (Sentence-A, Sentence-B), either Sentence-B *contains* the answer to the question in Sentence-A or it does not.

- Appendix B of the BERT paper lists several additional fine-tuning methods.

Outline

BERT is Great, But ...

- BERT has been a huge commercial success story for Google. Many businesses have used it to create smarter customer facing e-commerce webpages. Google itself used BERT in practically every facet of its search engine.
- Here is a quote from an NVIDIA article on BERT:

> *" BERT models are able to understand the nuances of expressions at a much finer level. For example, when processing the sequence 'Bob needs some medicine from the pharmacy. His stomach is upset, so can you grab him some antacids?' BERT is better able to understand that "Bob" "his" and "him" are all the same person. Previously, the query 'how to fill bob's prescriptions' might fail to understand that the person being referenced in the second sentence is Bob. With the BERT model applied, it's able to understand how all these connections relate."*

- However, BERT has one limitation — the 512 token limit at the input to the BERT network.
- BERT's size limitation on the number input tokens is not problem if the goal is to do a good job of learning good sentence-to-sentence continuations.

BERT is Great, But ...(contd.)

- But what about also learning sentence-to-paragraph, paragraph-to-sentence, and paragraph-to-paragraph continuations? Or similar continuations that relate a sentence to longer-than-a-para narrative?

- At this point, it's good to review why BERT was designed with the 512 limit on the number of input tokens. To understand that you have to first come to grips with the fact **that EVERY Transformer based implementation is designed for some expected maximum number of tokens at its input**. As to why:

  [**Every transformer based implementation for any deep learning application is designed for a specific value of what I have labeled `max_seq_length` in my Transformer implementation in DLStudio. You can think of that as the expected maximum number of words in the input sequence. If N_w is the value for this maximum number of words, your attention map at output of any of the BasicEncoders will be a 2D array of shape $N_w \times N_w$. That makes sense because an Attention Map is supposed to tell as to what extent each word at the input attends to every other word in the same input. From the standpoint of the calculations involved, the Attention Map is a result of the dot product $Q \cdot K^T$ of the query Q and the key K tensors. Ignoring the batch axis and also assuming single-headed attention for the sake of argument, at the output of each BasicEncoder, the Q tensor is calculated by multiplying the learnable matrix W_Q of shape $[N_w, M]$ with the transpose of the data matrix that is actually at the output of the encoder, which is also of shape $[N_w, M]$. Since these matrices must be learnable, their sizes have to set in advance. Therefore, the max value for N_w must be known in advance, as must the value for the size M of the embeddings.**]

Approaches Based on Segmenting the Longer Inputs

- If you wanted to learn narrative continuity properties at a level higher than what BERT currently does, one would ask why not just use a large value for max number of input tokens.

- Unfortunately, the solution to BERT's limitation is not as easy as that because of the amount of input data that must be ingested for generative pre-training in general. In general, the size of GPU memory you need goes up quadratically with the max size of the number of tokens at the input.

  [**Consider the following: The Transformer implementation I presented in my Week 13 lecture, I run out of GPU memory if I exceed 4 Attention Heads and 4 Basic Encoders for the embedding size of 256. It would be infeasible for me to be able to run that code with the numbers used in BERT for the same parameters. My guess is that doubling the number of input tokens for BERT would obviously be technically feasible, but it would considerably length the execution time and, also, create a heavier-duty product for its customization for specific applications.**]

- As it turns out, it is possible to extend BERT in its fine-tuning phase to create a more expansive framework that can learn language continuities at much larger level of abstraction.

RoBERT, ToBERT, and Sliding Windows

- For example, the following publication by Pappagari et. al. "*Hierarchical Transformers for Long Document Classification*":

  https://arxiv.org/pdf/1910.10781.pdf

  shows that if your overall goal is just classification it is possible to use the following strategy to learn language continuity properties with input token sequences that exceed the 512 limit used in BERT: (1) You split long input token sequences into 512-token segments in order to conform to the input constraints of BERT. (2) You feed the output of BERT on each of the 512-token input segment into an LSTM-based recurrent layer (I'm sure you'd also be able to use a GRU). Subsequently, your classification is made on the basis of the hidden state of the LSTM.

- The authors of the paper described above called their overall framework RoBERT for "Recurrence over BERT."

RoBERT, ToBERT, and Sliding Windows (contd.)

- As a variant of the method described on the previous slide, the authors of the same paper also claim that can achieve similar results by feeding BERT;s output for each of the 512-token input segments into another Transformer. They referred to that variant as ToBERT for "Transformer over BERT".
- Another way of extending BERT for longer inputs (that I believe would only work for classification tasks) is to use what's known as the "Sliding Window Approach". A google search with a string like "sliding window for extending BERT" will throw up code fragments that are based on this technique.
- The methods presented above work (when they do) because: (1) BERT gives you a very strong model of language continuity — that is definitely the case at the sentence level since it is not unlikely for 512 tokens to include multiple sentences. And (2) the methods described above are all applied in the supervised fine-tuning phase of BERT when you are dealing with small amounts of data.

Outline

The Core Idea of GPT-2

- Here is the core idea that GPT-2 is based on:

  In order to create a great language model, all you have to do is to carry out unsupervised learning based on maximizing the conditional probability of a token given the previously seen tokens. Subsequently, it is possible to induce zero-shot behaviors on the model without any supervised fine-tuning.

- This idea also underlies the more recent versions of the GPT models from OpenAI — **except that it is deployed at a much grander scale, as you will see in the section on GPT-3.**

  [**The core notion mentioned above is not to imply that you would not need to carry out additional fine-tuning of the LM for specific applications. It is merely to emphasize the power of unsupervised learning in imbuing the model with behaviors that you would ordinarily think would only be achievable with post-facto fine-tuning.**]

- GPT-2 realizes this idea by thinking of a corpus as a collection of sentences (that, by the way, are not necessarily the linguistic sentences you are so familiar with) that we may label $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$.

- Let's now consider a hypothetical experiment in which multiple agents are claiming to have "mastered" the corpus in question.

The Core Idea of GPT-2 (contd.)

- Continuing with the thought experiment at the bottom of the previous slide, let's say that each agent supplies us with a value for the following likelihood:

$$p((\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \tag{1}$$

- This likelihood value is the joint probability for all n sentences showing up in the order shown.

- The question now is: **Which agent gets the crown? Obviously, it'll have to be the agent that provides us with the highest value for the likelihood.**

- In another version of the same hypothetical experiment, the same agent might compute the likelihoods associated with the different permutations of the set of sentences in the corpus and assume that it has "understood" the corpus when the likelihood associated with the sentence order $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ is the highest.

From Sentence-Based Modeling to Token-Based Modeling in GPT

- Exactly the same arguments would apply if we think of $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ as a sequence of tokens in a corpus.
- Previously, on Slides 40 through 42, I talked about the WordPiece algorithm that is used for tokenization in BERT. As stated there, WordPiece initializes the token vocabulary with the individual characters in the words. It then progressively merges the characters on the basis of the joint probabilities of the merged symbols vis-a-vis their marginal probabilities.
- The GPT models use the BPE (Byte Pair Encoding) algorithm for tokenization. A distinguishing feature of BPE is that, instead of starting with ASCII characters, it starts with bytes directly as the basic elements of each word in text. It merges the bytes into subwords and, in turn, subwords into longer subwords, on the basis of the frequencies of the merged subwords vis-a-vis the frequencies of the components that are merged.

Masking for Autoregressive Modeling

- Autoregressive modeling of token sequences in a corpus means that we can factorize the likelihood shown in Eq. (1) as:

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_n) \;=\; \prod_{k=1}^{n} p(\mathbf{x}_k \mid \mathbf{x}_1, \ldots \mathbf{x}_{k-1}) \tag{2}$$

- We can use what's known as "*autoregressive masking*" to estimate the likelihoods in the manner dictated by Eq. (2).

- Assuming that the number of tokens in the input sequence to the Transformer is N and that M is the size of the embeddings, the shape of the input (not including the batch axis) will be $[N, M]$. With autoregressive masking, ignoring the batch axis, the mask is a 1-D tensor that is best visualized as follows as the input is processed from left to right:

```
                          <------------------     N   ---------------------->
estimate at pos 0:    SOS   0    0    0    0    0    0    0    ...     0
estimate at pos 1:    SOS   1    0    0    0    0    0    0    ...     0
estimate at pos 2:    SOS   1    1    0    0    0    0    0    ...     0
estimate at pos 3:    SOS   1    1    1    0    0    0    0    ...     0
   ...                                 .....
   ...                                 .....
estimate at pos N:    SOS   1    1    1    1    1    1    1    ...     1
```

Masking for Autoregressive Modeling (contd.)

- To elaborate, at the start of the scan through the input, all of the tokens in the input are masked out by setting all the element of the mask to 0. We then seek to predict the first token by setting the mask element to 1 while the rest stay at 0. After the SOS token, only the first element of the input sequence will be considered for this prediction. Subsequently, through the action of the mask, the prediction at each token position would become a function of all the input tokens and their corresponding predictions up to that position.

- The figure in the next slide depicts using the Decoder side of the Transformer based Encoder-Decoder architecture for an autoregressive estimation of the likelihood of each token given all the previously seen tokens.

[**The main difference between the Transformer Decoder shown in my Week 13 lecture and the one on the next slide is the absence of cross-attention in the latter. Since now we are not engaged in translation, the idea of a cross-attention simply does not apply.**]

Autoregressive Estimation of Token Likelihoods

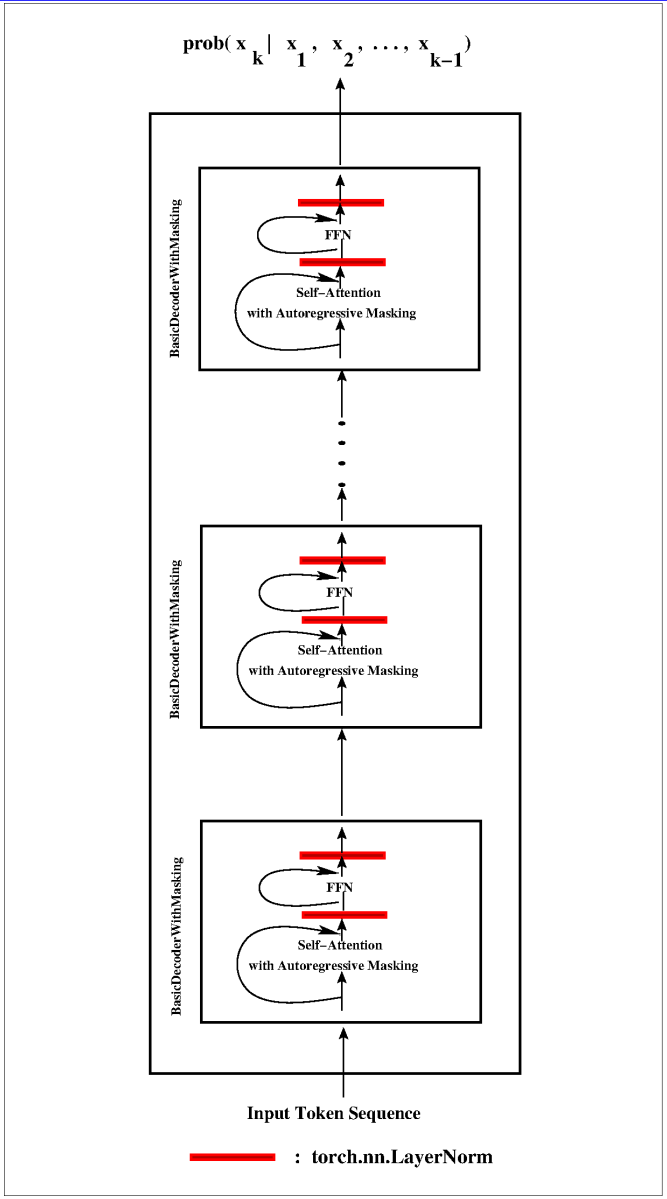


Figure: Using the Decoder part of the Transformer presented in my Week 13 lecture, this figure shows us estimating the likelihood of the token at position k given all the previously seen tokens.

Master Decoder with Autoregressive Masking

- I'll next revisit the decoder classes in the Transformer implementations in DLStudio and discuss their use for autoregressive estimation of token likelihoods.
- Autoregressive masking required for estimating the token likelihoods can be implemented with the `MasterDecoderWithMasking` class that I presented previously in my Week 13 lecture — except for the differences caused by the fact that for the absence of a target language and, consequently, the absence of cross attention.
- As shown on Slide 72, the `MasterDecoderWithMasking` orchestrates the invocation of a stack of `BasicDecoderWithMasking` instances. The `BasicDecoderWithMasking` is presented on Slide 73.
- Lines (H) through (S) of the `MasterDecoderWithMasking` implementation shown on Slide 72 utilize the masking action illustrated on Slide 76. In Line (H) we apply the current mask to the input token sequence and then in Line (R) we increase the length of the mask by one.

The `MasterDecoderWithMasking` Class

- Also note how in Line (B) in the next slide, we define the `BasicDecoderWithMasking` instances that are needed. The linear layer in Line (C) is needed because the output of the decoder must ultimately be mapped as a probability distribution over the entire token vocabulary for the input
- With regard to the data flow through the network, note how the mask is initialized in Line (D). The mask is a vector of one's that grows with the prediction for each output word. We start by setting it equal to just a single-element vector containing a single "1".
- Lines (E) and (F) in the code on the next slide declare the tensors that hold the final output of the `MasterDecoder`. This final output consists of two tensors: (1) one for holding the integer index in the token vocabulary for which the output log-prob is maximum. And (2) the other for holding the actual values of the log-probs over the token vocab. The log-probs are produced by the `nn.LogSoftmax` in Line (L).

MasterDecoderWithMasking (contd.)

```
class MasterDecoderWithMasking(nn.Module):

    def __init__(self, dls, xformer, how_many_basic_decoders, num_atten_heads):
        super(TransformerFG.MasterDecoderWithMasking, self).__init__()
        self.dls = dls
        self.max_seq_length = xformer.max_seq_length
        self.embedding_size = xformer.embedding_size
        self.token_vocab_size = xformer.token_vocab_size                                                    ## (A)
        self.basic_decoder_arr = nn.ModuleList([xformer.BasicDecoderWithMasking( dls, xformer,
                                                num_atten_heads) for _ in range(how_many_basic_decoders)])  ## (B)
        ## Need the following layer because we want the prediction for each token to be a probability
        ## distribution over the token vocabulary. The conversion to probs would be done by the criterion
        ## nn.CrossEntropyLoss in the training loop:
        self.out = nn.Linear(self.embedding_size, self.target_vocab_size)                                   ## (C)

    def forward(self, sentence_tensor, final_encoder_out):                                                  ## (D)
        ##  This part is for training:
        mask = torch.ones(1, dtype=int)                                                                     ## (E)
        ##  A tensor with two axes, one for the batch instance and the other for storing the predicted
        ##  word ints for that batch instance:
        predicted_word_index_values = torch.ones(sentence_tensor.shape[0], self.max_seq_length,
                                                                 dtype=torch.long).to(self.dls.device)      ## (F)
        ##  A tensor with two axes, one for the batch instance and the other for storing the log-prob
        ##  of predictions for that batch instance. The log_probs for each predicted token over the entire
        ##  token vocabulary:
        predicted_word_logprobs  = torch.zeros( sentence_tensor.shape[0], self.max_seq_length,
                                            self.token_vocab_size, dtype=float).to(self.dls.device)         ## (G)
        for mask_index in range(1, sentence_tensor.shape[1]):
            masked_target_sentence = self.apply_mask(sentence_tensor, mask, self.max_seq_length,
                                                                    self.embedding_size)                    ## (H)
            ## out_tensor will start as just the first word, then two first words, etc.
            out_tensor = masked_target_sentence                                                             ## (I)
            for i in range(len(self.basic_decoder_arr)):                                                    ## (J)
                out_tensor = self.basic_decoder_arr[i](out_tensor, final_encoder_out, mask)
            last_word_tensor = out_tensor[:,mask_index]                                                     ## (K)
            last_word_onehot = self.out(last_word_tensor.view(sentence_tensor.shape[0],-1))                 ## (L)
            output_word_logprobs = nn.LogSoftmax(dim=1)(last_word_onehot)                                   ## (M)
            _, idx_max = torch.max(output_word_logprobs, 1)                                                 ## (N)
            predicted_word_index_values[:,mask_index] = idx_max                                             ## (P)
            predicted_word_logprobs[:,mask_index] = output_word_logprobs                                    ## (Q)
            mask = torch.cat( ( mask, torch.ones(1, dtype=int) ) )                                          ## (R)
        return predicted_word_logprobs, predicted_word_index_values                                         ## (S)

    def apply_mask(self, sentence_tensor, mask, max_seq_length, embedding_size):
        out = torch.zeros_like(sentence_tensor).float().to(self.dls.device)
        out[:,:len(mask),:] = sentence_tensor[:,:len(mask),:]
        return out
```

The `BasicDecoderWithMasking` Class

- Shown below the `BasicDecoderWithMasking` class whose instances are used as Basic Encoder layers in the figure on Slide 69.
- The code shown here is different from the same class in my Week 13 slides because now we have no need for cross-attention.
- Also note that the mask that is used in `forward()` is set by the Master Decoder on the previous slide.

```
class BasicDecoderWithMasking(nn.Module):

    def __init__(self, dls, xformer, num_atten_heads):
        super(TransformerFG.BasicDecoderWithMasking, self).__init__()
        self.dls = dls
        self.embedding_size = xformer.embedding_size
        self.max_seq_length = xformer.max_seq_length
        self.num_atten_heads = num_atten_heads
        self.qkv_size = self.embedding_size // num_atten_heads
        self.self_attention_layer = xformer.SelfAttention(dls, xformer, num_atten_heads)
        self.norm1 = nn.LayerNorm(self.embedding_size)
        ## What follows are the linear layers for the FFN (Feed Forward Network) part of a BasicDecoder
        self.W1 =  nn.Linear( self.max_seq_length * self.embedding_size, self.max_seq_length * 2 * self.embedding_size )
        self.W2 =  nn.Linear( self.max_seq_length * 2 * self.embedding_size, self.max_seq_length * self.embedding_size )
        self.norm3 = nn.LayerNorm(self.embedding_size)

    def forward(self, sentence_tensor, mask):
        ## self attention
        masked_sentence_tensor = sentence_tensor
        if mask is not None:
            masked_sentence_tensor = self.apply_mask(sentence_tensor, mask, self.max_seq_length, self.embedding_size)
        Z_concatenated = self.self_attention_layer(masked_sentence_tensor).to(self.dls.device)
        Z_out = self.norm1(Z_concatenated + masked_sentence_tensor)
        ## for FFN:
        basic_decoder_out =  nn.ReLU()(self.W1( Z_out2.view(sentence_tensor.shape[0],-1) ))
        basic_decoder_out =  self.W2( basic_decoder_out )
        basic_decoder_out = basic_decoder_out.view(sentence_tensor.shape[0], self.max_seq_length, self.embedding_size )
        basic_decoder_out =  basic_decoder_out  + Z_out2
        basic_decoder_out = self.norm3( basic_decoder_out )
        return basic_decoder_out

    def apply_mask(self, sentence_tensor, mask, max_seq_length, embedding_size):
        out = torch.zeros(sentence_tensor.shape[0], max_seq_length, embedding_size).float().to(self.dls.device)
        out[:,:,:len(mask)] = sentence_tensor[:,:,:len(mask)]
        return out
```

Outline

Architectural Highlights of GPT-2

- There exist several variants of GPT-2 that are labeled Small, Medium, Large, and Extra Large. The sizes of the embedding vectors are: 768, 1024, 1280, and 1600, respectively. And the number of what I refer to as the BasicDecoders are: 12, 24, 36, and 48. I believe that the number of attention heads is 12 for the case of Small and 20 for the case of Large. The max sequence length is 1024 in all cases. The number of learnable parameters is 117 million for Small, 774 million for Medium, and 1.5 Billion for Large. Additionally, the token vocab is of size 50,257 in all cases.
- Referring to the figure on Slide 69, what the above means is that, in the smallest of the GPTs, you have 12 of what I have labeled as `BasicDecoderWithMasking` in that figure. And, in the largest of the GPTs, you have 48 of those.

Outline

Basic Unsupervised Learning the Same as in GPT-2

- I'll start by repeating here the core idea of GPT-2 that was presented previously on Slide 64:

  "In order to create a great language model, all you have to do is to carry out unsupervised learning based on maximizing the conditional probability of a token given the previously seen tokens. Subsequently, it is possible to induce zero-shot behaviors on the model without any supervised fine-tuning."

- That idea is also at the heart of GPT-3. The main difference — the huge difference — between GPT-2 and GPT-3 is the scale at which the unsupervised learning is carried out. Scale refers both to the size of the model used and the size of the training data.

- Compared to GPT-2's 1.5 Billion learnable parameters, GPT-3 has 175 Billion learnable parameters.

- If one wanted to expand on the "core idea" described above specifically to the case of GPT-3, you would add:

  "When the scale of unsupervised learning undertaken allows for the model to capture its 'understanding' of the textual data at a very general level, you'll also be able to induce significantly enhanced one-shot and few-shot performance."

Model Scale-Up in GPT-3

- Compared to the numbers for GPT-2, the config numbers shown below for GPT-3 are truly mind-blowing. For GPT-2, the number of learnable parameters was just 1.5 Billion.
- These numbers are from Table 2.1 in the 2022 paper "*Language Model are Few-Shot Learners,*" by Brown et al. https://arxiv.org/pdf/2005.14165.pdf.

```
GPT-3 Configuration:

    Number of Learnable Parameters:    175 Billion

    Number of Layers:                  96              [A layer in GPT-3 is the same thing as
                                                        BasicDecoderWithMasking in the figure
                                                        on Slide 69]
    Embedding Size:                    12,288

    Number of Attention Heads:         96

    Size of Hidden in FFN:             128

    Batch Size:                        3.2 Million

    Max Sequence Length:               2048            [also referred to as Context Widnow size]

    Learning Rate:                     0.6 x 10^-4
```

Why is Scale Important in Unsupervised Learning

- Simply said, the larger the scale at which unsupervised learning has been carried out, the greater the likelihood that the model has acquired a broadbased general understanding of the relationships between the different elements in the input data.
- The above conjecture was confirmed by the authors of the GPT-3 paper by experimenting with eight different versions of GPT-3 with different model sizes, ranging from 125 Million to 13 Billion and, finally, to 175 Billion. The experiments involved measuring the performance of the models on 42 accuracy-dominated benchmark datasets. The results are summarized in the figure in Slide 81.
- The tasks "zero-shot", "one-shot", and "few-shot" test the unsupervised-learned model with prompts that may induce additional context-specific training of the model **but without any gradient-based updates or fine-tuning of the learnable parameters**. More on these tasks on the next slide.

Why is Scale Important in Unsupervised Learning (contd.)

- Zero-shot means that you are testing the innate ability of the learned model, that is, without supplying it with any prompts to guide its behavior.
- On the other hand, one-shot and few-shot task involve prompts that provide guidance to the model in the form of analogies. An example of a zero-shot task: "Write me a poem about sunset.". As an example of a one-shot task, you would prompt the model with an example like "2 + 3 = 5" and you would then ask it to find the answer for "6 + 8". For a few-shot version of the same, you would provide multiple examples of what it means to add two numbers.
- For few-shot tasks, the number of examples you can supply depends on how many can be fit in the context-window (the same thing as the max sequence length) of the Transformer. As shown on Slide 78, this parameter is equal to 2048. The symbols used by the authors for the number of examples supplied for few-shot testing is K and its values ranges between 10 and 100.

Why is Scale Important in Unsupervised Learning (contd.)

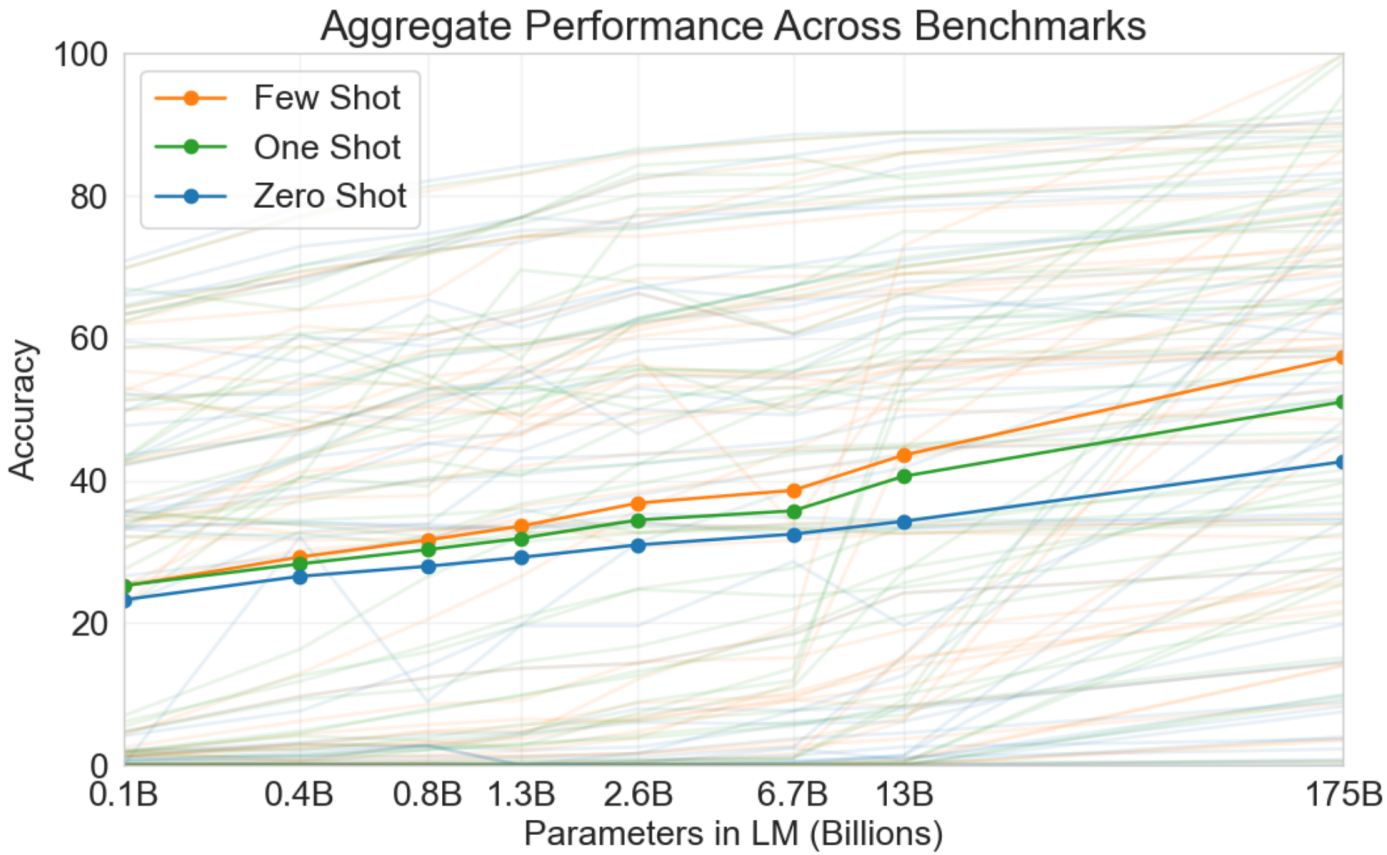


Figure: This figure is from "*Language Model are Few-Shot Learners,*" by Brown et al. `https://arxiv.org/pdf/2005.14165.pdf`.

Outline

In-Context Learning

- In-Context Learning (ICL) refers to a user interacting with a pre-trained LLM through prompts with the goal of imbuing the LLM with awareness of the user's values and preferences.
- It is important to realize while this interaction may create the impression that the LLM is becoming smarter by learning from the examples supplied by the user, acquiring those "smarts" never translates into any modifications to the learnable parameters of the LLM.
- Whatever the LLM learns through its interaction with the user is confined to the context of the *current conversation* between the user and the LLM.
- Should the user terminate the current conversion by, say, clicking a button that may be labeled "Start New Conversation", whatever 'knowledge' was acquired up to that point is simply discarded. Hence the name "Learning in Context".

In-Context Learning (contd.)

- Keep in mind the fact that an LLM like GPT-3's main job is only to make the best possible prediction for the next token. However, by sequencing together these predictions until the next token is one of the termination tokens, the LLM can provide an extended response to a user-supplied prompt.
- Since the prediction for the next best token comes with a likelihood value for the token, we can talk about "*prompt completion probability*" as a measure of the confidence the LLM has in that response to the prompt.
- The prompt completion probabilities can be worked into different types of meaningful prompt-usage-patterns between the LLM and the human.

In-Context Learning (contd.)

- For example, the human may ask the LLM to come up with N responses to a prompt and, subsequently, by choosing one of those as the most preferred, provide guidance to the LLM about what the human thinks is most important.
- Next, I'll briefly review the "ALLIES" framework for generating responses to user-supplied prompts. This approach was presented in the 2023 publication *ALLIES: Prompting Large Language Model with Beam Search,*" that you can download from

  `https://arxiv.org/pdf/2305.14766`

- ALLIES was designed as a solution to the following limitation of LIC: In its interaction with the user, the LLM may find itself needing additional information that it is unable to generate on its own (that is on the basis of what it learned during pre-training). When that happens, how should the LLM weigh its options: Either it can make do with what it can generate on its own; or provide a better answer if it could get hold of the knowledge that it finds missing.

In-Context Learning (contd.)

- For example, the human may ask the LLM to come up with N responses to a prompt and, subsequently, by choosing one of those as the most preferred, provide guidance to the LLM about what the human thinks is most important.
- In the beam-search algorithm, the current *state* of the possible response is represented by the 5-tuple $(q, \mathcal{Q}, \mathcal{E}, r, s)$ where q is the original prompt; $\mathcal{Q}$ the possible set of prompt completions; $\mathcal{E}$ the set of resources relevant to those completions; the current response r, and the associated confidence score s. This 5-tuple is used in a 4-stage framework of ALLIES as shown on Slide 88 and as summarized below:
  - The first stage, Beam Initialization, consists of having the LLM point to the other resources (such as documents) related to the original prompt, summarizing the resources, adding the 5-tuples formed by the summaries to the beam. This would require asking the LLM to create its best responses needed for those tuples.

In-Context Learning (contd.)

- The second stage, Beam Expansion, iteratively pops out the 5-tuple element from the front of the beam, resolves it by comparing its score against the expected scores from the elements. And, if necessary, expand it as described in the previous step.
- The third stage, Beam Pruning, occurs when the beam search has reached its maximum allowable depth. At that point, we retain only a certain number of top-ranked answers.
- And in the last stage, Beam Termination, if the score with the highest ranked retained 5-tuple exceeds our threshold, we terminate the beam search. If beam-search does not arrive as the max depth, it simply returns the highest scoring element.

In-Context Learning

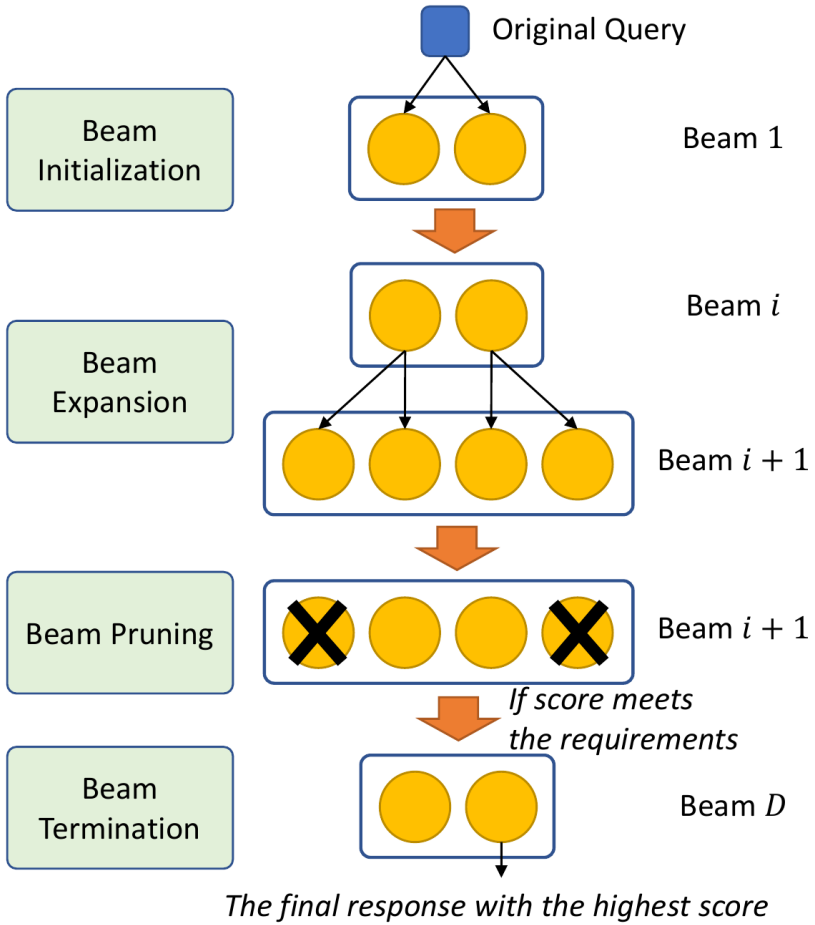


Figure: This figure is from the 2023 publication "*ALLIES: Prompting Large Language Model with Beam Search,*" by Sun et el. `https://arxiv.org/pdf/2305.14766`.

Outline

About GPT-4

- Regarding GPT-4, there is certainly a tech report that OpenAI released recently

  https://arxiv.org/pdf/2303.08774

- However, if your main interest lies in understanding the architecture of GPT-4, here is a quote from Page 2 of the report:

  "Given both the competitive landscape and the safety implications of large-scale models like GPT-4, this report contains no further details about the architecture (including model size), hardware, training compute, dataset construction, training method, or similar."

- That is, for **competitive reasons and safety implications**, OpenAI has decided to **not** release any information regarding the architecture of GPT-4.

- The report focuses solely on the capabilities of GPT-4 as showcased by its performance on various benchmark datasets.

Outline

The Reasons for babyGPT

Here are the main reasons that led to the creation of babyGPT:

- To introduce the students in Purdue's Deep Learning class to the foundational concepts in how to create a Base Language Model through self-supervised learning. Large Language Models start out as Base Models that are subsequently fine-tuned with reinforcement learning. **The focus of this module is solely on Base Modeling.**
- To demonstrate small-scale large-language modeling that, for educational purposes, can be run on a typical university lab GPU.
- To create a self-contained module that, given a set of media URLs, will download the articles from those websites, train a BPE tokenizer from the corpus of articles collected, and let you create a Base Model from the corpus that you can play with through the prompting script in the module.
- babyGPT demonstrates that, for the purpose of teaching and learning, it is possible to create a small-scale end-to-end implementation that downloads a corpus of news media articles, trains a BPE tokenizer for the corpus you downloaded, and, finally, uses the corpus for training an autoregressive model for the next token prediction based on unsupervised learning.

The Major Components of babyGPT

The babyGPT module contains the following Python classes:

1. ArticleGatherer
2. ArticleDataset [supplies the data downloader for training]
3. TrainTokenizer
4. TransformerFG [borrowed from Transformers in DLStudio]
5. MasterDecoderWithMasking [borrowed from Transformers in DLStudio]
6. PromptResponder

The download page for babyGPT is:

`https://engineering.purdue.edu/kak/distBabyGPT`

babyGPT — ArticleGatherer Class

- About the ArticleGatherer, you supply it with a list of URLs to media news sites.
- It then uses the Python `Newspaper` module (which understands the structure of a typical news HTML file) to download the articles from each of those URLs. It is important to keep in mind that ArticleGatherer skips over non-HTML article files at the media websites.
- Unfortunately, many popular news websites now hide their content behind paywalls implemented with JavaScript. [Examples of such websites include www.nyt.com, www.wsj.com, www.bbc.com, etc.]
- For obvious reasons, if the list of the URLs you provide ArticleGatherer consists of mostly such websites, the size of the corpus you create for experimenting with babyGPT could be much to small to be any fun.

babyGPT — ArticleDataset Class

- After you have used `ArticleGatherer` to download the news articles for the training corpus, the next thing you are going to need is a dataloader. That's exactly what's provided by the ArticleDataset class.
- It randomly shuffles all the articles gathered and creates a number of dataloading streams equal to the batch-size that you are using for training babyGPT.
- The data input for the i^{th} batch instance is provided by the i^{th} stream. Logically speaking, you can think of each stream as a concatenation of the news articles that were randomly chosen for that batch instance.

babyGPT — TrainTokenizer Class

- I have already said a great deal about the `TrainTokenizer` class elsewhere in these slides.
- The babyGPT module comes with a pre-trained tokenizer with a vocab size of around 50,000 tokens. I trained this tokenizer using the babyGPT module on the athlete news dataset created by Adrien Dubois. The name of the tokenizer JSON in the Examples directory is: `104_babygpt_tokenizer_49270.json`

babyGPT — TransformerFG Class

- About the `TransformerFG` component of babyGPT, as mentioned already, language modeling is best carried out with Transformer based implementations.
- To that end, I borrowed TransformerFG from DLStudio's Transformers module. TransformerFG is my implementation of the concept of the Transformer as proposed by Vaswani et al. in their seminal paper "Attention is All You Need." The suffix "FG" stands for "First Generation."

babyGPT — MasterDecoderWithMasking Class

- The `MasterDecoderWithMasking` part of babyGPT has also been borrowed from DLStudio's Transformers module.
- Note that unsupervised learning that is needed for autoregressive language modeling only uses the Decoder side of the Encoder-Decoder architecture that would otherwise be needed for a Transformer-based framework for translating one language into another.
- As to the reason for the "Master" prefix in the name of the decoder, a language modeling code typically requires a number of Transformer layers, with each layer using multiple Attention Heads to calculate what's known as Self Attention. In my DLStudio code, I have refered to this layered organization of the Transformers as MasterEncoder and MasterDecoder, and to each Transformer layer as the BasicEncoder and the BasicDecoder.

babyGPT — PromptResponder Class

- About the final component of babyGPT, `PromptResponder`, its purpose is to put the trained babyGPT model to use by having it respond appropriately to the prompts supplied by a user.
- Given a prompt in the form of a sentence fragment, the PromptResponder uses its next-token prediction ability to keep on generating the tokens until it reaches the end-of-sentence token or until it has generated a specified number of sentences through this process.

Dealing with Context Disruption Caused By The <SoS> Token

- What comes in the way of training babyGPT are the textual discontinuities created by how a batch is constructed for each new iteration of training. As explained elsewhere in these, the list of all the documents in the training corpus is first randomized and then divided into a number of token streams, with one stream for each batch instance. (This randomization of the files and the division into token streams is carried out afresh at the beginning of each epoch.)
- Subsequently, when a fresh batch is needed, for each batch instance you "draw" from its corresponding stream a `max_seq_length` number of tokens. The special <SOS> token is placed at the beginning of each such token stream segment and another special token <EOS> at the end.
- This insertion of the <SOS> and <EOS> tokens disrupts the continuity of the token streams as you imagine — which runs contrary to the main point of the exercise which is to learn the continuity properties.

Some Results Produced by babyGPT

- After training babyGPT with the previously cited "athlete news dataset" (around 85MB), here's an example of one of the better result that shows autoregressive predictions (one token at a time) at the end of 8 epochs of training:

```
ground-truth:      round of the French Open .  Wozniacki delivered a ruthless performance
GT Token Seq:      round _ of _ the _ French _ Open . Woz ni ac ki _ delivered _ a _ ru th less _ perfo
   predicted:      round _ of _ the _ French _ Open _ Char ni ac ki _ won _ a _ ru led less _ performan
 detokenized:      round of the French Open Charniacki won a ruledless performance
```

- As you can see, the model makes the following errors in its autoregressive predictions:

```
Woz         =>   Char
ruthless    =>   ruledless
delivered   =>   won
```

- And here is an example of a prompt and the response received from the trained model:

```
Prompt:    basketball stars
Response:  basketball stars LaBron James and the Cleveland Cavaliers  in the
```

Some Results Produced by babyGPT (contd.)

- The reason the response at the bottom of the previous slide is not a complete sentence is because of the "max_seq_length" limits placed by the small GPU (Nvidia 2080) I used for training.
- For the results shown on the previous slide, I used a batch_size of 200, embedding size of 128, max_seq_length of 30, and 4 attention heads.
- **You can download babyGPT from the link on Slide 93.**