Natural Language Processing

Class 10: LLMs: Data, modeling, tokenization, and sampling

Adam Faulkner

November 11, 2025

The prompt-based learning revolution

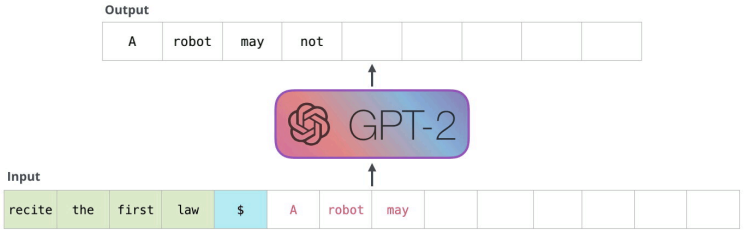
- With the release of GPT-3, LLM researchers introdcued a revolutionary new way of solving NLP problems traditionally solved using traditional machine learning: **prompt-based learning**
- Rather than training individual models —either via traditional methods or BERT-based finetuning—one can solve NLP tasks by simply providing text to an LLM trained via causal language modeling task and ask the LLM to condition on this task to solve the task.
- When single test instance is given at prompt-time we call this setting **sero-shot**; when one or more examples are provided to the LLM to illustrate the task, we call this setting **few-shot**.

Tokenization

- Tokenization has also been key to the development of contemporary LLMs.
- Our early discussion of **ngram models** introduced us to the most basic form of tokenization: splitting on whitespace.
- But this is the least efficient form of tokenization and doesn't address the **out-of-vocabulary** problem: what if, during inference, the model encounters an **ngram** that it hasn't seen during training?
- Tokenization schemes such as **byte-pair-encoding** and **WordPiece** (introduced by BERT's developers) address these issues,

The autoregressive language modelling task

- As we learned in a previous class, GPT (Generative Pretrained Transformer)-style models are instances of **causal** language modelling: these models predict the next token in a sequence of tokens, and the model can only attend to tokens to its left.
- These models are trained on an *autoregressive language modelling task*: after each token is produced, that token is added to the sequence of inputs and that new sequence becomes the input to the model in its next step



Can all NLP tasks be solved using next-word prediction?

- Causal LLMs, particularly the GPT family of LLMs, introduced the novel idea that **all NLP tasks could be cast as word prediction tasks**.
- Another framing: All NLP tasks can be recast as cases of **conditional generation**.
- Conditional generation is the task of generating text conditioned on an input piece of text.
- The idea leverages the concept of a **prompt**, a piece of input text that is fed to an LLM, which then continues generating text token by token, conditioned on the prompt.

Classification as text completion

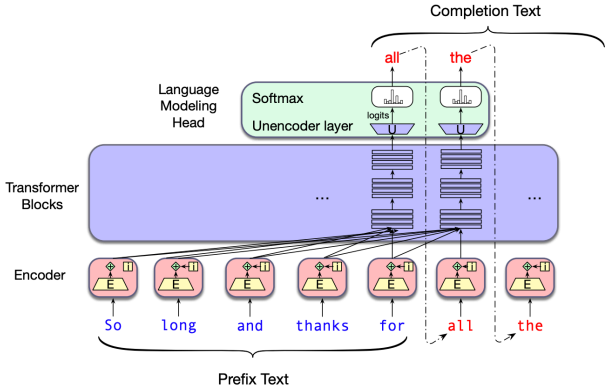


Figure 1: Left-to-right (also called autoregressive) text completion with transformer-based LLMs. As each token is generated, it gets added to the context as a prefix for generating the next token.

Classification as word prediction: Sentiment analysis

- For example, we can cast sentiment analysis as a text-completion task by giving a language model a context like:

```
The sentiment of the sentence "The movie was great" is:
```

- We let the LLM complete the sentence and then evaluate the following conditional probability of the words ”positive” and the word ”negative” to see which is higher:

P(positive|The sentiment of the sentence ”The movie was great” is:)

P(negative|The sentiment of the sentence ”The movie was great” is:)

Question-answering as word prediction

- We can cast the task of question-answering as word prediction by giving an LLM a question and a token such as `A:` suggesting that an answer should come next:

```
Q: Who wrote the book The Origin of Species"?  A:
```

- We then ask a language model to compute the probability distribution over possible next words given this prompt:

$$P(w|\text{Q: Who wrote the book "The Origin of Species"? A:})$$

- We then let the LLM continue predicting words, with each word adding to the existing context until we get the answer: `Charles Darwin`

$$P(\text{Charles}|\text{Q: Who wrote the book "The Origin of Species"? A:})$$

$$P(\text{Darwin}|\text{Q: Who wrote the book The Origin of Species"? A: Charles})$$

Text summarization as word prediction

- Conditional generation can even be used to accomplish tasks that must generate longer responses than simply ”positive” or ”Charles Darwin”–in fact entire summaries of texts can be produced via word-prediction
- We can cast summarization as language modeling in an LLM prompt by providing the text that we want summarized follow the text by a token like `TL;DR` (”too long; didnt’ read”) which is a prevalent marker introducing summaries in the LLM’s training data (reddit posts, emails, etc.)
- We can then do conditional generation: give the language model this prompt, and then have it generate the following words, one by one, and take the entire response as a summary.

Text summarization as word prediction

- So, the following text (from a well-known summarization corpus)

  ```
  The only thing crazier than a guy in snowbound Massachusetts
  boxing up the powdery white stuff and offering it for sale
  online? People are actually buying it. For $89, self-styled
  entrepreneur Kyle Waring will ship you 6 pounds of Boston-area
  snow in an insulated Styrofoam box enough [...] TL;DR:
  ```

  should produce something like

  ```
  Kyle Waring will ship you 6 pounds of Boston-area snow in an
  insulated Styrofoam box enough for 10 to 15 snowballs, he
  says. But not if you live in New England or surrounding
  states
  ```

Data used in LLM pretraining

- Web text is usually taken from corpora of automatically-crawled web pages like the **common crawl**, a series of snapshots of the entire web produced by the nonprofit Common Crawl (`https://commoncrawl.org/`) that each have billions of webpages.
- Various versions of common crawl data exist, such as the Colossal Clean Crawled Corpus (C4), a corpus of 156 billion tokens of English that is filtered in various ways (deduplicated, removing non-natural language like code, sentences with offensive words from a blocklist).

Data used in LLM pretraining

The Pile: an 825 GB English text corpus that is constructed by publicly released code, containing again a large amount of text scraped from the web as well as books and Wikipedia

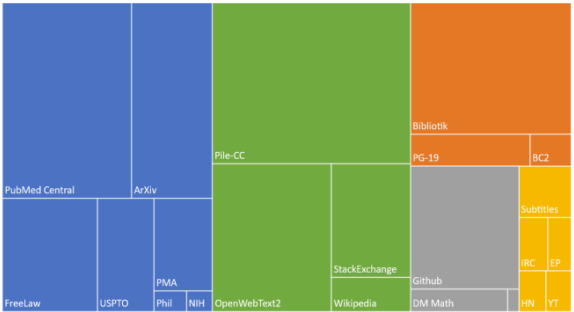
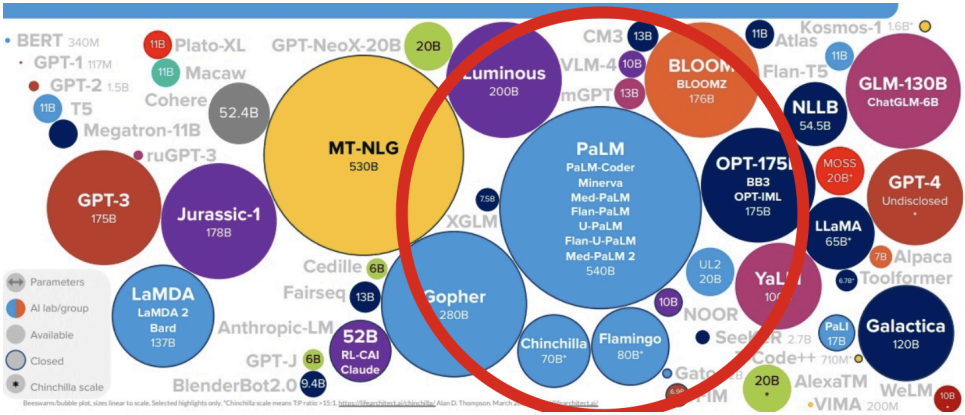


Figure 2: The Pile corpus, color coded as academic, internet, prose (books), dialogue (movie subtitles, chat data), and misc..

What does "N parameters" really mean?



What does "N parameters" really mean?

- The performance of LLMs has been shown to be mainly determined by 3 factors:
  1. model size (the number of parameters not counting embeddings)
  2. dataset size (the amount of training data)
  3. the amount of compute used for training.
- Taken together, we can improve a model by adding parameters (adding more layers or having wider contexts or both), by training on more data, or by training for more iterations.
- The relationships between these factors and performance are known as **scaling laws**

What does "N parameters" really mean?

- When reporting out the number of parameters of a new model, practitioners can use the following **scaling law** calculation with d as the input and output dimensionality of the model, d_{attn} as the self-attention layer size, and d_{ff} the size of the feedforward layer):

$$\begin{aligned} N &\approx 2dn_{layer}(2d_{attn} + d_{ff}) \\ &\approx 12n_{layer}d^2 \\ &\text{(assuming } \ d_{attn} = d_{ff}/4 = d) \end{aligned}$$

- So, GPT-3, with $n = 96$ layers and dimensionality $d = 12288$, has $12 \times 96 \times 122882 \approx 175$ billion parameters.

Tokenization: Byte-Pair Encoding

- Instead of defining tokens as words or as characters (as in Chinese), we can use our data to automatically tell us what the tokens should be.
- A better tokenization algorithm will also help us address the **out-of-vocabulary** (OOV) problem, i.e, when an LM encounters a word that it hasn't seen during training.
- To deal with this OOV errors modern tokenizers automatically induce sets of tokens that include tokens smaller than words, called **subwords**

Tokenization: Byte-Pair Encoding

- Contemporary tokenization algorithms such as **Byte-pair encoding** and **WordPiece** have two parts: a **token learner**, and a **token segmenter**
- The token learner takes a raw training corpus (sometimes roughly preseparated into words, for example by whitespace) and induces a vocabulary, a set of tokens.
- The token segmenter takes a raw test sentence and segments it into the tokens in the vocabulary

Tokenization: Byte-Pair Encoding

- The simplest and most widely used tokenization algorithm is **byte-pair encoding**, which works as follows:

1. Start with a vocabulary that is just the set of all individual characters (the alphabet, numbers, etc.).
2. Examining the training corpus, choose the two symbols that are most frequently adjacent (say 'A','B')
3. The new merged symbol 'AB' is added to the vocabulary
4. Every adjacent 'A' B' in the corpus is replaced with the new AB.
5. Steps 2 - 4 are repeated, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm.

Tokenization: Byte-Pair Encoding

An example using an input corpus of 18 word tokens with counts for each word (the word *low* appears 5 times, the word *newer* 6 times, and so on), which would have a starting vocabulary of 11 letters:

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	
2 n e w _	

Tokenization: Byte-Pair Encoding

The BPE algorithm first counts all pairs of adjacent symbols: the most frequent is the pair `e r` because it occurs in *newer* (frequency of 6) and *wider* (frequency of 3) for a total of 9 occurrences. We then merge these symbols, treating `er` as one symbol, and count again:

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

Tokenization: Byte-Pair Encoding

Now the most frequent pair is `er _`, which we merge; our system has learned that there should be a token for word-final `er`, represented as `er _`:

corpus		vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2	l o w e s t _	
6	n e w er_	
3	w i d er_	
2	n e w _	

Tokenization: Byte-Pair Encoding

Next `n e` (total count of 8) gets merged to `ne`:

corpus		vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2	l o w e s t _	
6	ne w er_	
3	w i d er_	
2	ne w _	

Tokenization: Byte-Pair Encoding

- Our final vocabulary for this tiny corpus is

  `_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_`

- Once we've learned our vocabulary, the token segmenter is used to tokenize a test sentence. The token segmenter just runs the merges we have learned from the training data on the test data.
- By the end, if the test data contains the character sequence `n e w e r _`, it would be tokenized as a full word since we have `newer_` in our vocabulary
- But the characters of a new OOV word like `l o w e r_` would be merged into the two tokens `low er _`

Tokenization: WordPiece

- Originally developed to pretrain BERT.
- Very similar to BPE in terms of the training, but the actual tokenization is done differently
- As with BPE, WordPiece (WP) starts from a small vocabulary including the special tokens used by the model and the initial alphabet.
- WP identifies subwords by adding a special prefix (like for BERT) and each word is initially split by adding that prefix to all the characters inside the word, so *word* becomes `w ##o ##r ##d`

Tokenization: WordPiece

- WP's count and merge rules are somewhat different from that of BPE.
- Instead of selecting the most frequent pair, WP computes a score for each pair, using the following formula:

$$score_{WP} = (freq\_of\_pair) / (freq\_of\_first\_element \times freq\_of\_second\_element)$$

Tokenization: WordPiece

- By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary.
- This is particulary important for "productive" word prefixes that signal negation in English such as "un."
- So ("un", "##able") is unlikely to be merged right away since the two pairs "un" and "##able" will likely each appear in a lot of other words and have a high frequency.
- But ("hu", "##gging") will probably be merged faster since "hu" and "##gging" are likely to be less frequent individually, mainly because "hu" is not a productive prefix in english.

Tokenization: WordPiece

- Unlike BPE, which uses counts to prioritize merge decisions, WP uses the results of $SCORE_{WP}$ to decide the next pair to merge.
- Suppose our corpus contains the following words and counts: *hug* (10), *pug* (5), *pun* (12) *bun* (4), and *hugs* (5).
- Our initial vocabulary is then

  `[b, h, p, ##g, ##n, ##s, ##u]`

  and, after splitting, the corpus becomes

  `(h ##u ##g, 10), (p ##u ##g, 5), (p ##u ##n, 12), (b ##u ##n, 4), (h ##u ##g ##s, 5)`

Tokenization: WordPiece

- Given

  `(h ##u ##g, 10), (p ###u ##g, 5), (p ##u ##n, 12), (b ##u ##n, 4), (h ##u ##g ##s, 5)`

  which pair should be merged first?
- After calculating $score_{WP}$ for each pair, the highest rank is `(#g, ##s)`. We merge this as `##gs`, add it to the vocabulary and make the replacement in the corpus:

  `(h ##u ##g, 10), (p ###u ##g, 5), (p ##u ##n, 12), (b ##u ##n, 4), (h ##u # #gs, 5)`
- Our next highest scored pair is `(h, ##u)` which becomes `hu`:

  `(hu ##g, 10), (p ###u ##g, 5), (p ##u ##n, 12), (b ##u ##n, 4), (h ##u # #gs, 5)`

  and so on until we reach a chosen vocabulary size.

Tokenization: WordPiece

- When tokenizing a test text, rather than adopting the merge rules learned from the training corpus, as with BPE, WP starts with the word to tokenize, finds the longest subword in the word that is also in the vocabulary, and then splits on it
- So, given the word `bugs`, `b` is the longest subword starting at the beginning of the word that is in the vocabulary, so we split there and get `[b, ##ugs]`. Then `u` is the longest subword starting at the beginning of `##ugs` that is in the vocabulary, so we split there and finally get `[b, ##u, ##gs]` (`##gs` was also in the vocabulary so it gets added).

The importance of sampling strategies

- The most common method for decoding in large language models is **sampling**
- Sampling from a model's distribution over words means to choose random words according to their probability assigned by the model
- A good sampling strategy will find the optimal tradeoff between **quality** and **diversity**
- Methods that emphasize the most probable words tend to produce generations that are rated by people as more accurate, more coherent, and more factual, but also more boring and more repetitive
- Methods that give a bit more weight to the middle-probability words tend to be more creative and more diverse, but less factual and more likely to be incoherent or otherwise low-quality

Greedy decoding

- **Greedy decoding**: Generate the most likely word given the context
- Make a choice that is locally optimal, whether or not it will turn out to have been the best choice with hindsight
- At each time step in generation, the output y_t is chosen by computing the probability for each possible output (every word in the vocabulary) and then choosing the highest probability word, (or the *argmax*):

$$\hat{w}_t = argmax_w \in VP(w|\mathbf{w}_{<t})$$

- The main problem with greedy decoding is that because the words it chooses are (by definition) extremely predictable, the resulting text is generic and often quite repetitive

Greedy decoding

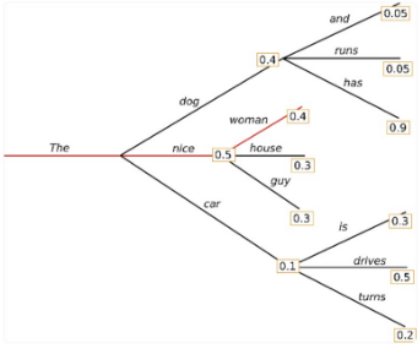


Figure 3: Generating the sentence *The nice woman* using greedy decoding

Beam search

- A popular alternative to greedy decoding is the **beam search** algorithm
- Instead of choosing the best token to generate at each timestep, we keep k possible tokens at each step
- k is called the **beam width**, on the metaphor of a flashlight beam that can be widened or narrowed

Beam search

The beam search algorithm

1. At the first decoding step, compute a softmax over the entire vocabulary, assigning a probability to each word
2. Select the k-best options from this softmax output. These initial k outputs are the **search frontier** and these k initial words are called **hypotheses**
3. At subsequent steps, each of the k best hypotheses is extended incrementally: a softmax is calculated over V to extend the hypothesis to every possible next token. Each of these $k \times V$ hypotheses is scored by $P(y_i|x, y < i)$: the product of the probability of the current word choice multiplied by the probability of the path that led to it.
4. Keep only the k best hypotheses, so there are never more than k hypotheses at the frontier of the search

Beam search

The beam search algorithm (continued)

5. This process continues until an EOS is generated indicating that a complete candidate output has been found
6. At this point, the completed hypothesis is removed from the frontier and the size of the beam is reduced by one
7. Repeat steps 1 - 6 until the beam size is 0

Beam search

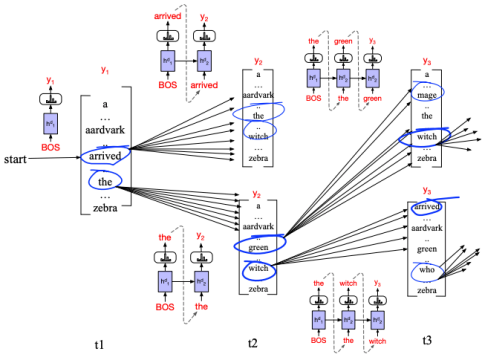


Figure 4: Beam search decoding of *the green witch*, beam size = 2. At each time step, we choose the k best hypotheses, form the V possible extensions of each, score those $k \times V$ hypotheses and choose the best $k = 2$ to continue.

Beam search

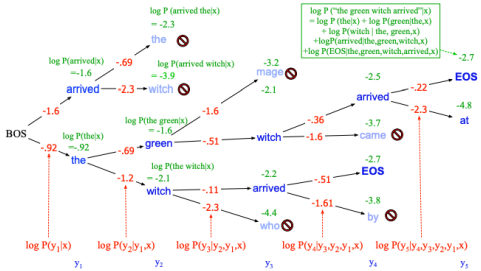


Figure 5: Beam search decoding of *the green witch arrived*, beam size = 2.We maintain the log probability of each hypothesis in the beam by incrementally adding the logprob of generating each next token. Only the top k paths are extended to the next step.

Top-k sampling

- Basically a generalization of greedy decoding to k tokens (top-k sampling with $k = 1$ is just greedy decoding)
- The top-k sampling algorithm
  1. For each word in the vocabulary V, use the language model to compute the likelihood of this word given the context
  2. Sort the tokens by their likelihood; keep top k
  3. Renormalize the scores of the k words to be a legitimate probability distribution
  4. Randomly select a word from the new distribution

Top-p or nucleus sampling

- In **top-p sampling** or **nucleus sampling** , we keep the top p percent of the probability mass, rather than the top k words
- Sample tokens with the highest probability scores until a specified threshold p is reached
- Tends to generate text that is more varied and creative then greedy or top-k sampling

Temperature sampling

- Divide the logits by a temperature parameter τ before we normalize it by passing it through the softmax
- Higher values of τ lead to greater variability, more creative text
- Lower values of τ lead to boring, generic text
- Why this works:
  - Softmax pushes high values toward 1 and low values toward 0
  - The lower τ is, the larger the scores being passed to the softmax since dividing by a smaller fraction ≤ 1 results in making each score larger
  - When larger numbers are passed to a softmax the result is a distribution with increased probabilities of the most high-probability words and decreased probabilities of the low probability words

Temperature sampling

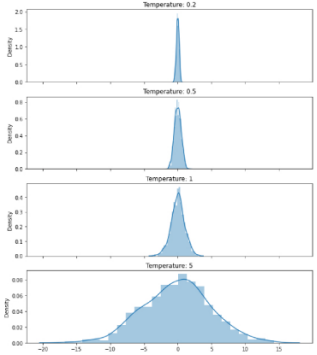


Figure 6: Temperature sampling: As the temperature nears 0, the probability of the most likely word approaches 1.

Next class: November 18

- Next: Aligning LLMs to human preferences and instructions
- Reading
  - Jurafsky & Martin Chapter 9: Post-training: Instruction Tuning, Alignment, and Test-Time Compute
  - Training language models to follow instructions with human feedback
  - Direct Preference Optimization: Your Language Model is Secretly a Reward Model
  - Mixtral of Experts