Building Large Language Modeling (LLMs)

PRE-TRAINING

Language Modeling Recap

It is the probability distribution of the sequences of tokens/words.

Consider the following examples that:

1. P (the, mouse ate, the cheese) = 0.02. This model gives the probability distribution for these words to exist as the correct sentence.
2. P (the, the, mouse, ate, cheese) = 0.0001. The model gave this probability distribution because it can have syntactic knowledge of this sentence not being exactly right as repeating two words.
3. P (the, cheese, ate, the, mouse) =0.001 as the model would know mouse eats the cheese and not the other way around.

Language models are thus generative models.

Auto-Regressive Language Models

Generates text sequentially, predicting the next word (or token) based on the previously generated words.

$$P(x1,\ldots,XL) = p(x1)p(x2|x1)p(x3|x1,x2) \ldots.$$

This is one way of the type of language modeling so it has its pros and cons, one downside of this model is its working based on a for loop like nature where it is trying to continuously predict the next word given the initial words and so on thus it is much slower and there is no parallelism like in the BERT models which utilize the Auto-Encoding Models where the entire text is processed at once.

The following are the key steps of following by such models:

1. Tokenization of Inputs.
2. Pass through a model (Usually a Transformer based model).

Remember if our Auto regressive model is not using just statistical modeling approaches and instead a deep learning architecture (like transformer, RNN or CNNs) to predict the next word in sequence, we would call it a neural autoregressive model.

3. Output a probability distribution over the next token.
4. Sample (Inference) or compare to the actual token (Training).

Training: Adjust weights to increase the likelihood of the correct next token using cross-entropy loss.

Inference: performed after training when the model is ready to work upon unseen real-world data.

<u>*Neural Autoregressive Language Models*</u>

The following are the detailed steps for this approach:

1. Embed tokens into vector representations.
2. Process through a neural network (transformer).
3. Output a representation of the context.
4. Linear layer maps to vocabulary size.

The AR model processes the text and generates hidden representations using multiple transformer layers, before outputting the word the last layer converts these hidden representations into a logits vector where each value corresponds to a word in the vocabulary. The size of this output vector is equal to the number of words (tokens) in the vocabulary.

If the model outputs a hidden state of size 1024, the linear layer transforms this 1024-dimensional hidden state into a vector of size 50,000.

5. SoftMax yields a probability distribution over the next token.

Loss Function:

Cross-entropy loss is a way to measure how well a model's predicted probability distribution matches the actual labels.

- If the model assigns **high probability to the correct word**, the loss is **low.**
- If the model assigns **low probability to the correct word**, the loss is **high**

<u>*Tokenization*</u>

Talking further on the tokenization step of the LLMs as they hold great importance in the overall process. Tokenization is an important step for an LLM, so we need to know how to tokenize our words.

- Poor Generality:

  If the whole word is tokenized it makes the system more specific and minor spelling mistakes can create completely new and unknown tokens which cause these models to not recognize the word and ay struggle to understand the meaning.

- Poor Efficiency:
  If the words are tokenized with a character level tokenization this would create long sequences, increasing computational cost (quadratic complexity in transformers).

Therefore, our goal is to assign tokens a common subsequence of an avg 3-4 letters per token. One of the better ways of tokenization are thus given below:

1. <u>BYTE PAIR ENCODING (BPE):</u>

Algorithm:

> Start with a large text corpus; assign each character a token.
> Iteratively merge the most frequent adjacent token pairs into new tokens
> Repeat until the desired vocabulary size is reached.

<u>Vocabulary size</u> refers to the total number of unique tokens that the tokenizer learns and uses from the dataset. It represents how words and sub words are represented in the model. If we set the vocabulary size too small the words break into very small tokens which become inefficient and if set the vocabulary size too large, we would have too many tokens which would increase memory and complexity and may cause poor generalization. Therefore. The goal is to find an optimal balance where common words remain whole and rare words are broken into useful sub-words.

<u>Pre-tokenization</u> is a preprocessing step before BPE or other tokenization algorithms that split the test into manageable chunks based on language specific rules.

Example in BPE:

"ChatGPT tokenizer"

- Pre-tokenization might split it into "ChatGPT" and "tokenizer" before applying sub-word tokenization.
- Then, "tokenizer" can be broken into "token" + "izer" efficiently.

2. Q&A ON TOKENIZATION:

Algorithm:

STEP NO. 1

There are two main parts of the Q&A model:

1. Question -> “What is tokenization?”

2. Context (Passage) -> “Tokenization is the process of splitting text into tokens for NLP.”

Both need to be combined into a single input for the model. Most transformer-based models (like BERT) follow a specific format:

[CLS] Question [SEP] Context [SEP]

For our example: [CLS] What is tokenization? [SEP] Tokenization is the process of splitting text into tokens for NLP. [SEP]

- [CLS] → Special token marking the start (useful for classification tasks).
- [SEP] → Separates the question from the context.

STEP NO. 2

Now the tokenization is done. The tokenizer breaks the words into sub-words when necessary. Here you’d have to note:

["[CLS]", "What", "is", "token", "##ization", "?", "[SEP]",

"Token", "##ization", "is", "the", "process", "of", "splitting", "text", "into", "tokens", "for", "NLP", ".", "[SEP]"].

- “tokenization" splits into ["token", "##ization"] because it's not in the vocabulary as a whole word.
- "Token" in the context remains separate because capitalized words are treated differently.

STEP NO. 3

Each token is mapped to a **unique ID** from the model’s vocabulary.
Example using BERT (vocab size: **30,522** tokens):

[101, 2054, 2003, 19204, 4697, 1029, 102,

19204, 4697, 2003, 1996, 2833, 1997, 8550, 3793, 2046, 19204, 2005, 17953, 1012, 102]

STEP NO. 4

Since we have **two segments** (Question + Context), we need a way to tell the model **which part is which**.

- **Segment ID = 0** → For the **question**
- **Segment ID = 1** → For the **context passage**

[0, 0, 0, 0, 0, 0, 0,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

STEP NO. 5

To tell the model **which tokens to focus on** (1) and **which to ignore (padding)** (0).

Since we don't have padding here, all values are **1**:

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

KEY NOTES AND SUMMARY:

Small Tokens for Robustness (Typos & Misspellings)

Why Keep Smaller Tokens?

- If tokenization only worked at the **word level**, it would fail on typos.
- Instead, **sub-word tokenization** ensures even **misspelled words** can be understood.
- Example:
    - "tokenization" → [token, ##ization] (normal case)
    - "t0kenization" → [t0, ken, ##ization] (still somewhat interpretable)

Robustness:
Smaller token retention ensures **typos, rare words, and informal text** still make sense.

Uniqueness of Tokens & Context Disambiguation

Each Token Has a Unique ID, but Context Matters

- Every token in a vocabulary gets a unique numerical ID.
- But the same token can have different meanings based on context.
- Example: "bank"
    - "I deposited money in the **bank**." → (finance meaning)
    - "The boat is near the river **bank**." → (geography meaning)

    - "bank left during the turn." → (aviation meaning)

Therefore, LLMs use **context** (surrounding words) to figure out the meaning.

Encoding Rule: Use the Largest Token Possible

Why Prefer "token" Over "t"?

- Tokenization algorithms follow the longest match rule:
    - If "token" exists in vocabulary, prefer "token"
    - Avoid breaking it into smaller parts like ["t", "o", "k", "e", "n"]
- This improves efficiency and reduces the total number of tokens.

Example in Byte Pair Encoding (BPE):

- "unhappiness" is tokenized as:
    - [un, ##happiness] (better) instead of
    - [un, ##hap, ##pi, ##ness] (worse, more tokens used).

Drawbacks of Tokenization

Math: Whole Numbers Get Tokenized Together

- Numbers (like 327) are often treated as single tokens instead of breaking them into digits.
- This limits generalization, since a model trained on 327 may not know 328 unless seen.
- Some modern models use character-level tokenization for numbers to improve this.

Code: Indentation & Special Formatting

- Programming languages (like Python) have strict formatting rules, such as:
    - Indentation (4 spaces)
    - Line breaks
    - Special symbols ({}, ;, :=, etc.)
- Standard NLP tokenizers aren't optimized for code, so custom tokenization is needed.
- Example: GPT-4 has improved code tokenization for better Python handling.

Newer models like Code Llama & GPT-4 Turbo have custom tokenization rules for programming languages.

Future of Tokenization

Will We Move to Character/Byte-Level Models?

- Current transformers rely on subword tokenization for efficiency.
- But tokenization adds complexity and sometimes fails on unseen words.
- Some modern models (like Gemini & Mistral) explore character/byte-level tokenization to remove this step.

Why is the Byte-Level the Future?

- More flexible → No need for a predefined vocabulary.
- Handles any language → Even those without spaces (e.g., Chinese, Thai).
- Better for typos & rare words.

Evaluation:

What is perplexity?

Perplexity is a mathematical measure of how well a model predicts a dataset. It is calculated as:

$$\text{Perplexity} = 2^{\text{avg per-token loss}}$$

Lower Perplexity means a better model; a perplexity of 1 means a perfect prediction while a perplexity if vocabulary size means the model is just randomly guessing.

- If a model has PPL = 70, it means it "hesitates" between 70 possible tokens at each step.
- Modern LLMs (GPT-4, Claude, LLaMA) have PPL <10, meaning they are far more confident.

Depending on the tokenizer; a model using byte-level BPE has a different PPL than a word-based tokenizer, also they are varied based on the text being tested. However, this method is not used much in academia now because the modern LLMS require more robust benchmarks.

Common NLP Benchmarks?

1. HELM (Holistic Evaluation of Language Models)
    - Tests different tasks (example: summarization, Q&A, reasoning).
    - Measures biases, efficiency and generalization.
2. Hugging Face Open Leaderboard
    - Tracks LLM rankings based on standardized dataset.

3. MMLU (Massive Multi-task Language Understanding)

  - A multiple-choice test for LLMs across many subjects like science, history etc.
  - The model is then evaluated based on the highest probability of choosing the right answer. This is an easier method of evaluation where we don't need to check if the model's wording is correct rather than just if it has picked the right letter. Moreover, it avoids ambiguity as some answers could be correct in different ways, but MMLU forces exact matches.

Challenges

1. Inconsistency Across Evaluations

Different benchmarks give different scores for the same model.

Example:

  - LLaMA-65B scored 63.7% on one benchmark but only 48.8% on another.
  - This happens because benchmarks test different skills.

2. Train-Test Contamination

If test data was included in training, scores become artificially high.

How to detect this?

  - Check if the model scores higher on the original test order than on a shuffled version.
  - If original order scores much higher, the model likely memorized the answers.

Data

Idea: Use all the clean internet.

Note: The internet is dirty and not representative of what we want. Therefore, following are some practical steps we may take to sort our data from the internet before use:

1. Download all the internets. Common crawl: 250 billion pages, > 1PB (> 1e6 GB). (as of August-2024)
2. Text extraction from HTML (challenges: math, boilerplate).

3. Filter undesirable content (e.g., NSFW, harmful content, PII).
4. Deduplicate (URL/document/line). Example: All headers/footers/menu in forums are always the same.
5. Heuristic filtering. Remove low-quality documents (e.g., number of words, word length, outlier tokens, dirty tokens).
6. Model-based filtering. Predict if a page could be referenced by Wikipedia.
7. Data mix. Classify data categories (code/books/entertainment). Reweight domains using scaling laws to get high downstream performance.

- **Also**: Learning rate (lr) annealing on high-quality data, continual pretraining with longer context.

Learning rate annealing is essentially a strategy to reduce the learning rate as training progresses, allowing the model to make larger updates early on when it's still far from optimal, and smaller, more refined updates as it gets closer to a solution.

Continual Pretraining means continuously improving the model by training it on new, high-quality datasets instead of training it just once and stopping.

Scaling Laws Concept

- More data, larger models, and more computes improve performance predictably (no overfitting).
- Linear relationship on a log scale: compute, data, or parameters vs. test loss (log perplexity).
- Data quality impacts scaling more than minor architectural tweaks.
- Compute drives progress (per Moore's Law variants); focus on systems/data over small architecture changes. (This is what the OpenAI researcher/ instructor @ CS229 Building LLMs said in August 2024 but DeepSeek R1 outperformed OpenAI-o1 model with their adjustment to the Transformer Architecture by introducing the Mixture of Experts Model).

Through these scaling laws we can compare the architectures like transformers with LSTMs to understand which works better than the other. Training large models is computationally expensive and time-consuming. Instead of tuning hyperparameters (like learning rate, batch size, weight decay) directly on a huge model, researchers first train and fine-tune a smaller version of the model. This small model helps determine the best hyperparameters efficiently. Once optimal hyperparameters are found on the small model, they are scaled and applied to larger models. Scaling laws suggest that certain patterns in training behavior remain consistent as models grow larger. This means hyperparameters tuned on a smaller model can often be extrapolated to larger models with minor adjustments.

Understanding the difference between Parameters and Tokens: (Values let for better concept)

Tokens typically refer to the context window size (also called the token limit).

◆ Context Window = How Much Text the Model Can "Remember" at Once

- Let's say if a model has a 150K token context length, it means it can process and attend to up to 150,000 tokens at once.
- In practical terms, this is how much text (words, sentences, paragraphs) the model can "see" in one interaction before it forgets earlier parts.

70B parameters means:

- The model has 70 billion internal values that help it predict the next token based on training.
- This includes weights, biases, attention heads, and neurons in its neural network.
- Think of it as a giant web of interconnected numbers that have been adjusted during training to generate meaningful responses.

Chinchilla: is a large language model (LLM) developed by DeepMind that follows a more optimal scaling law for training large models. It was introduced in 2022 as an improvement over previous LLMs like GPT-3.

Chinchilla demonstrated that training a smaller model on more data leads to better performance at the same computational cost.

Example Reference:

Llama 3 (400B Parameters) Training: 15.6T tokens, 3.8e25 FLOPs, 16,000 H100 GPUs, ~70 days (~26M GPU hours).
Cost: ~$52M (GPU rental at $2/hour) + ~$25M (50 employees at $500k/year) = ~$75M.
Carbon Footprint: ~4,000 tons CO2e (equivalent to 2,000 JFK-London flights).

Note: Below Biden’s 1e26 FLOPs scrutiny threshold.

Mixture of Experts Model utilized by DeepSeek R1:

"Mixture-of-Experts", or MoE; the idea behind MoE is that instead of processing input by a single large model, we divide the model into smaller networks, called experts, where each expert specializes in something unique - like a different domain or a different type of syntax. And when we process the input, we don't pass it to all experts. Instead, we use an additional smaller network, called router, which looks at the given input and maybe says something like: "Hmm, I think only the first and the last experts should process this input". And for the next input, it picks a different set of experts and so on. MoE is applied to every layer of the transformer network, and it replaces all dense layers. And so, if this works, then this is great, because first, we’re processing the input in these specialized sub-networks that can do the task better.

And second, and that's even more important here, we're saving a lot of computation.

Because all other experts are being completely ignored. MoE as a concept is nothing new. It dates to the early '90s, and for example, it was recently used in an LLM called Mistral. But the problem is that it can be hard to get it working well as many things can go wrong here. For example, how can you separate the knowledge and make sure that each expert is specialized? Also, there is a well-known problem called "routing collapse", where the router learns to use only one or very few experts every single time and then ignores the rest. So, you need to somehow encourage the router to select.

POST-TRAINING

Supervised Fine-Tuning (SFT)

Fine-tuning a pre-trained LLM on human-provided desired answers. (Learn more about it in the notes of Finetuning Models Deep Learning Concepts Repo @ github.com/harris-giki.

Key to ChatGPT’s leap from GPT-3 (research tool) to a widely known model. This method is slow and costly as it requires gathering human data, but we can bypass that using LLMs to scale the data collection.

Reinforcement Learning from Human Feedback (RLHF)

SFT allows AI models to mimic human behaviors, but this approach has inherent limitations:

- Human Bound: While humans excel at judgement, they may struggle with generating high quality content; (writing an entire book). SFT will directly inherent these limitations.
- Hallucination Issues: Even when trained on correct data, an SFT model may generate outputs that sound plausible but contain inaccuracies, such as citing nonexistent references.
- Cost Consideration: Gathering high-quality human-generated responses for training is expensive, making large-scale supervised learning impractical.

Therefore, rather than merely imitating human responses, RLHF aims to maximize human preference—selecting outputs that are most favorable according to human evaluators.

RLHF Process

1. A fine-tuned language model generates two different responses to the same instruction.
2. Human evaluators compare and rank these responses, selecting the better one.
3. The model is then fine-tuned to favor responses that received positive feedback ("good" or "green") over those that received negative feedback ("bad" or "red").

RLHF Methods

1. Proximal Policy Optimization (PPO)

PPO is a reinforcement learning (RL - Unlike SFT, which maximizes likelihood, RLHF optimizes a model to maximize human preference using a reward system.) technique that optimizes a model based on reward-based model's outcomes, consider the following approach for this method.

- A classifier is trained on human preferences (green/red responses) that assigns a continuous reward score (logits via SoftMax), this may also be called as Reward model.
- The reward model is then used alongside the LLM to give the reward score to the predictions made by the LLMs. Based on the those results, the LLM is adjusted to make more sound predictions.

Application in ChatGPT: ChatGPT follows an iterative pipeline:

Supervised Fine-Tuning (SFT) → 2. Reward Model Training → 3. PPO Optimization

2. Direct Preference Optimization (DPO)

DPO is a more straightforward alternative to PPO that eliminates reinforcement learning altogether.

- Instead of using RL, DPO directly optimizes the likelihood of human-preferred responses and minimizes the likelihood of disliked ones.
- The objective in DPO is to maximize the log-likelihood of human-preferred answers while minimizing the log-likelihood of disfavored answers. This approach aligns mathematically with the optimization goal of PPO, ensuring that the model improves preference alignment without requiring reinforcement learning steps.

Why PPO Was Used Initially? OpenAI had prior expertise in RL techniques (notably via John Schulman). PPO also enabled the use of unlabeled data through reward modeling.

Why DPO Is Now Preferred? DPO achieves results comparable to PPO but is significantly simpler, making it the standard approach in both open-source and industry applications.

Data Collection for RLHF

The success of RLHF depends heavily on high-quality human-labeled data:

- Human Labeling Process: Evaluators follow detailed guidelines to compare outputs.
- Example Task: Given the instruction, *"Tell me about self-driving cars"*, evaluators rank two AI-generated responses based on accuracy, coherence, and informativeness.
- Challenges: Human annotation is time-consuming and costly, making scalability a key concern.

Correctness vs. Superficial Metrics

- Observation: People often prioritize superficial aspects (e.g., output length, form) over correctness when evaluating models.
- RLHF Impact: When applying Reinforcement Learning from Human Feedback (RLHF), increased RLHF iterations result in longer model outputs.

    - Example: ChatGPT’s verbose responses annoy users due to excessive RLHF lengthening outputs.