MedAlpaca Technical Report

Muhammad Ridho 2206130776

Master Degree of Mathematics

Faculty of Mathematics and Natural Sciences

University of Indonesia

mridho@sci.ui.ac.id

1. Fundamental

Transformer Architecture

Let us explore how a transformer architecture predicts text based on some provided input, often referred to as *prompt*. In line with successful models, such as the GPT and LLAMA series, we focus on the setting, where the model iteratively predicts the next word pieces (i.e., tokens) based on a given sequence of tokens. This procedure is coined *autoregressive* since the prediction of new tokens is only based on previous tokens. Such conditional sequence generation tasks using autoregressive transformers are often referred to as *decoder-only* settings. Although there is a maximum *context length* in practice, we will work with sequences of arbitrary length for ease of presentation. We define the shorthand notation $S^* := \bigcup_{n\in\mathbb{N}} S^n$ for a set S to denote the set of sequences $\boldsymbol{s} = (s^{(i)})_{i=1}^n \subset S$ with arbitrary length $n \in \mathbb{N}$. For a function $\mathcal{F}\colon S_1 \to S_2$, we denote by $\mathcal{F}^*\colon S_1^* \to S_2^*$ the *entrywise* applied mapping given by

$$\mathcal{F}^*(\boldsymbol{s}) := (\mathcal{F}(s^{(i)}))_{i=1}^n. \tag{1}$$

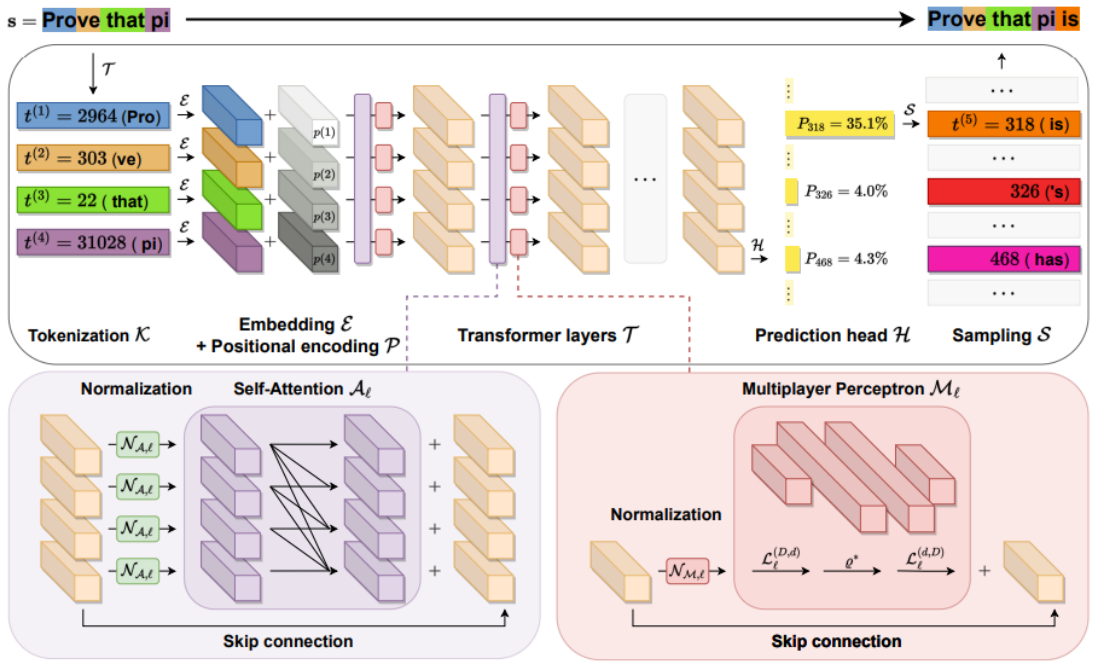


Figure 2: Illustration of the operations of an LLM for the input text "`Prove that pi`". The token indices, as well as the probabilities for the next token, are taken from GPT-2 [29] using the implementation in the `transformers` library [41]. The highest probability for the next token is assigned to `318` corresponding to the word "`is`".

Tokenization $\mathcal{K}$. First, we want to clarify how we define word pieces, i.e., tokens. Mathematically speaking, we seek an injective mapping $\mathcal{K}\colon A^* \to T^*$ from the given text, i.e., a sequence $\boldsymbol{a} = (a^{(i)})_{i=1}^N$ of characters in an alphabet A to a sequence of $n \leq N$ tokens $(t^{(i)})_{i=1}^n$, where typically $T \coloneqq \{1, 2, \dots, M\}$.

To represent text on a computer, we could encode every character $a^{(i)}$ individually, similar to *Unicode*. While this would lead to a small vocabulary T of tokens, it yields long sequences where individual tokens do not capture any linguistic information. For LLMs, one often employs *subword tokenization* [34], which creates a vocabulary of subwords by analyzing large text corpora and iteratively merging frequently occurring sequences of characters. Akin to a compression problem, one balances the length n of the sequences and the size[6] M of the vocabulary. As an example, the GPT-4 tokenizer[7] splits the word "discontinuity" into the subwords "dis" (prefix), "contin" (subword capturing the root of "continuous"), and "uity" (suffix). Another example can be found in Figure 2.

Embedding $\mathcal{E}$. To use these tokens (given by the indices of the subwords in the vocabulary) in a neural network, we embed each token $t^{(i)}$ into the same Euclidean space $E \coloneqq \mathbb{R}^d$. Intuitively, we seek a map $\mathcal{E}\colon T \to E$, such that the distance $\|\mathcal{E}(t^{(i)}) - \mathcal{E}(t^{(j)})\|$ corresponds to the linguistic similarity of the subwords represented by the tokens $t^{(i)}$ and $t^{(j)}$. In practice, such an embedding is often initialized with a sequence of M random initial embeddings and learned jointly with the transformer model from data.

Positional Encoding $\mathcal{P}$. Since $\mathcal{E}$ operates on each token $t^{(i)}$ independently, the embeddings $\mathcal{E}(t^{(i)})$ do not contain information on the position i of the (sub)words within a sentence[8]. Thus, one typically adds so-called *positional encodings*, which can be described by a mapping $\mathcal{P}\colon E^* \to E^*$. A commonly used choice is of the form

$$\mathcal{P}((e^{(i)})_{i=1}^n) \coloneqq (e^{(i)} + p(i))_{i=1}^n, \tag{2}$$

where $p\colon \mathbb{N} \to E$ can be a prescribed injective function, e.g., a sinusoid [40], or learned (similar to the embedding $\mathcal{E}$) [28].

In summary, tokenization $\mathcal{K}$, followed by an application of $\mathcal{E}$ to each token and the positional encoding $\mathcal{P}$, maps the text $\boldsymbol{a} \in A^*$ to a sequence of embeddings

$$\boldsymbol{e} \coloneqq (\mathcal{P} \circ \mathcal{E}^* \circ \mathcal{K})(\boldsymbol{a}) \in E^*. \tag{3}$$

where the length of $\boldsymbol{e}$ depends on $\boldsymbol{a}$ and the tokenization algorithm.

Transformer $\mathcal{T}$. The transformer can be represented as a neural network $\mathcal{T}\colon E^* \to E^*$. It is trained to map a sequence of embeddings $\boldsymbol{e}$ to another sequence of the same length containing contextual information. Based on the desired autoregressive structure, where the prediction of the next token only depends on the previous tokens, we want the i-th element of $\mathcal{T}(\boldsymbol{e})$ to contain information about all the embeddings $(e^{(j)})_{j \leq i}$, however, to be independent of $(e^{(j)})_{j>i}$.

The transformer is typically defined by a composition of $L \in \mathbb{N}$ blocks, consisting of *self-attention* maps $\mathcal{A}_\ell$, entrywise applied *normalizing layer* $\mathcal{N}_{\mathcal{A},\ell}$, $\mathcal{N}_{\mathcal{M},\ell}$, and *feed-forward multiplayer perceptrons* $\mathcal{M}_\ell$, i.e.,

$$\mathcal{T} \coloneqq \left(\left(\mathrm{Id} + \mathcal{M}_L^* \circ \mathcal{N}_{\mathcal{M},L}^*\right) \circ \left(\mathrm{Id} + \mathcal{A}_L \circ \mathcal{N}_{\mathcal{A},L}^*\right)\right) \circ \cdots \circ \left(\left(\mathrm{Id} + \mathcal{M}_1^* \circ \mathcal{N}_{\mathcal{M},1}^*\right) \circ \left(\mathrm{Id} + \mathcal{A}_1 \circ \mathcal{N}_{\mathcal{A},1}^*\right)\right). \tag{4}$$

In the above, Id denotes the identity mapping, commonly known as a *skip* or *residual connection*, and the addition is understood entrywise. The indices of the layers $\mathcal{N}$, $\mathcal{M}$, and $\mathcal{A}$ in (4) indicate the use of different trainable parameters in each of the layers. Let us describe these layers in more detail below.

Layers: Normalization $\mathcal{N}$. The normalizing layer can be interpreted as a re-parametrization with a learnable mean and standard deviation to stabilize training. For instance, using *layer normalization* $\mathcal{N}\colon E \to E$, we compute

$$\mathcal{N}(e) = \frac{\mathrm{diag}(s)}{\sigma}(e - \mu) + m, \tag{5}$$

where $\mu = \frac{1}{d}\sum_{i=1}^d e_i$ and $\sigma^2 = \frac{1}{d}\sum_{i=1}^d (e_i - \mu)^2$ are the mean and variance of $e \in E$, and $s, m \in E$ are learnable parameters [1].

Layers: Multilayer Perceptrons $\mathcal{N}$. The Multilayer perception (MLP) is a standard feed-forward neural network consisting of compositions of affine mappings and nonlinear activation functions. Let us define by $\mathcal{L}^{(m,n)}\colon \mathbb{R}^m \to \mathbb{R}^n$ an affine mapping $\mathcal{L}^{(m,n)}(x) \coloneqq Wx + b$, where the *weight matrix* $W \in \mathbb{R}^{n\times m}$ and the *bias vector* $b \in \mathbb{R}^m$ are learnable. Moreover, let $\varrho\colon \mathbb{R} \to \mathbb{R}$ be an activation function, e.g., the *GELU activation function* $\varrho(x) \coloneqq x\,\Phi(x)$, where Φ is the standard Gaussian cumulative distribution function [17]. A typical MLP $\mathcal{M}\colon E \to E$ used in transformers is then given by

$$\mathcal{M} \coloneqq \mathcal{L}^{(d,D)} \circ \varrho^* \circ \mathcal{L}^{(D,d)}, \tag{6}$$

where $D \in \mathbb{N}$ with $D \geq d$.

Layers: Self-Attention $\mathcal{A}$. As can be seen in (4) and Figure 2, the self-attention layer $\mathcal{A}\colon E^* \to E^*$ is the only layer that combines embeddings of different tokens; in other words, it *attends* to other tokens. Let us denote the input to the layer by $(e^{(i)})_{i=1}^n$ and focus on the i-th output. We first compute the (normalized) inner products

$$s_j^{(i)} = \frac{1}{\sqrt{k}} \left\langle \mathcal{L}_{\text{query}}^{(k,d)}(e^{(i)}), \mathcal{L}_{\text{key}}^{(k,d)}(e^{(j)}) \right\rangle, \quad j = 1, \dots, i, \tag{7}$$

with given $k \in \mathbb{N}$. On a high level, we can interpret $\boldsymbol{s}^{(i)} = (s_j^{(i)})_{j=1}^i \subset \mathbb{R}$ as similarities between the embedding $\mathcal{L}_{\text{query}}^{(k,d)}(e^{(i)})$ of the i-th token (i.e., the so-called *query*) and the embeddings $\mathcal{L}_{\text{key}}^{(k,d)}(e^{(j)})$ of the other tokens (i.e., *keys*); to satisfy the autoregressive structure, we only consider $j \leq i$. To normalize $\boldsymbol{s}^{(i)}$ to probabilities, we can further use a *softmax layer* $\operatorname{softmax}\colon \mathbb{R}^* \to \mathbb{R}^*$ given by

$$\operatorname{softmax}(\boldsymbol{s}^{(i)})_j \coloneqq \frac{\exp\left(s_j^{(i)}\right)}{\sum_{k=1}^i \exp\left(s_k^{(i)}\right)}, \quad j = 1, \dots, i. \tag{8}$$

We can now interpret $\operatorname{softmax}(\boldsymbol{s}^{(i)})_j$ as the probability for the i-th query to "attend" to the j-th key. The self-attention layer $\mathcal{A}$ can then be defined as

$$\mathcal{A}(\boldsymbol{e})_i \coloneqq \mathcal{L}^{(k,d)} \left(\sum_{j=1}^i \operatorname{softmax}(\boldsymbol{s}^{(i)})_j \mathcal{L}_{\text{value}}^{(k,d)}(e^{(j)}) \right), \quad i = 1, \dots, n, \tag{9}$$

where the outputs of $\mathcal{L}_{\text{value}}^{(k,d)}$ are often referred to as the *values* of the token embeddings $e^{(j)}$, and where the learnable affine layer $\mathcal{L}^{(k,d)}$ maps the weighted average of values back to $E = \mathbb{R}^d$.

Note that in practice, one typically considers a sum of $h \in \mathbb{N}$ such attention layers (so-called *heads*), each with dimension $k = d/h$ [40, 21]. Moreover, instead of considering vectors of variable length i, a mask enforces the autoregressive structure so that all operations can be efficiently batched.

Prediction Head $\mathcal{H}$. The *prediction head* or *un-embedding layer* can be represented as a mapping $\mathcal{H}\colon E^* \to \Delta^M$, where

$$\Delta^M \coloneqq \left\{ P \in [0,1]^M \colon \sum_{i=1}^M P_i = 1 \right\} \tag{10}$$

denotes the probability simplex in $\mathbb{R}^M$. It maps the sequence of transformed embeddings $(\tilde{e}^{(i)})_{i=1}^n \coloneqq \mathcal{T}(\boldsymbol{e})$ to a vector $P \in \Delta^M$, where P_i describes the probability of predicting $i \in T$ as the next token. Since the transformed embedding of the last token, i.e., $\tilde{e}^{(n)}$, contains information about the whole input text, a simple approach is to use a linear mapping composed with a softmax layer and define

$$P \coloneqq (\operatorname{softmax} \circ \mathcal{L}^{(M,d)})(\tilde{e}^{(n)}). \tag{11}$$

Sampling $\mathcal{S}$. There are multiple *sampling* strategies $\mathcal{S}\colon \Delta^M \to T$ to arrive at the final prediction for the next token $t^{(n+1)}$, see, e.g., [18]; the arguably simplest one, so-called *greedy sampling*, predicts the token with the highest probability, i.e.,

$$t^{(n+1)} = \mathcal{S}(P) \coloneqq \underset{i=1,\dots,M}{\arg\max}\, P_i, \tag{12}$$

see Figure 2. One can then apply the same operations to the extended sequence $\boldsymbol{t} = (t^{(i)})_{i=1}^{n+1}$, i.e.,

$$t^{(n+2)} \coloneqq (\mathcal{S} \circ \mathcal{H} \circ \mathcal{T} \circ \mathcal{P} \circ \mathcal{E}^*)\,(\boldsymbol{t}) \tag{13}$$

to iteratively compute further tokens[9]. Due to the autoregressive structure, this can efficiently be done by caching the previous (intermediate) results and only considering the computations for the new token.

Training

During training, we transform text corpora into sequences of tokens, such that, for a given sequence $(t_i)_{i=1}^n$, we already know the next token t_{n+1} based on the underlying text. One can thus compute the deviation D between the predicted probabilities P of the next token and the ground-truth t_{n+1}, typically using a *cross-entropy loss*; in practice, this procedure can be parallelized to compute average losses across many predictions. Using *automatic-differentiation*, one then computes the derivative $\nabla_\theta D$ of the average loss D with respect to the learnable parameters $\theta \in \mathbb{R}^p$ of the transformer $\mathcal{T}$, the embedding $\mathcal{E}$, the prediction head $\mathcal{H}$ (and the positional encoding $\mathcal{P}$ if it is trainable). Updating the parameter by subtracting a sufficiently small multiple $\lambda \in (0, \infty)$ of the derivative, i.e., $\theta_{k+1} = \theta_k - \lambda \nabla_\theta D$, one can iteratively minimize the loss—a method known as *stochastic gradient descent*. This is the essential mechanism by which word occurrence probabilities are estimated by training from raw data. With substantial engineering efforts, more elaborate versions of such training schemes can be parallelized on large GPU clusters and scaled to immense amounts of data. To get an idea of the dimensions, the largest LLaMA2 model with $p = 70 \cdot 10^9$ parameters was trained for more than 1.7 million GPU hours on about 2 trillion tokens of data from publicly available sources [39].

Training Costs and Emmisions

Training LLMs, as described in the previous section, is a computationally very intensive process and, therefore, costly to carry out in terms of electricity usage (assuming all the hardware would be in place). However, information about training costs and CO_2 emissions is not consistently provided in the literature. Notable exceptions include the LaMDA model. The authors [37] report that a total of 451MWh was consumed during training, and, as a result, approximately 26 tons of CO_2 were emitted. Using historic US prices[10] of 0.148 dollars per kWh, this amounts to a cost of 66,748 dollars. We note that costs may vary by country and by the energy source used to produce energy [35]. The GLaM model consumes, when trained on the largest dataset, similarly 456MWh and emits 40.2 tons of CO_2, which places it thus in a similar category to the LaMDA model in terms of cost and emission.

However, more modern LLMs incur significantly more energy consumption and emissions. For instance, the training of LLaMA2 (using 1.7 million hours on GPUs with a power consumption of about 400W) emitted more than 291 tons of Carbon dioxide equivalent (CO_2-eq) [39]. LLM vendors (such as OpenAI) typically do not release information about the costs (either in terms of consumed megawatt-hours or (rented) GPU-hours) of training their models, so only vague estimates are possible, which are nonetheless staggering. For example, training the older-generation GPT-3 model [4] was estimated, using GPU-hour prices from that time, to run up costs of approximately 4.6 million dollars [20].

Now let's jump on the most important concepts which LLaMA has included.

Pre-normalization Using RMSNorm

RMSNorm : Root Mean Square Layer Normalization

LLaMA normalizes the input of each transformer sub-layer, instead of normalizing the output. Inspiration of including pre-normalization is taken from GPT3.

RMSNorm is extension of Layer Normalization (LayerNorm). Reason behind using RMSNorm is the computational overhead in LayerNorm. This makes improvements slow and expensive. RMSNorm achieves comparable performance against LayerNorm but reduces the running time by 7%~64%.

Let first understand LayerNorm, It has two properties.

a. re-centring : It make model insensitive to shift noises on both inputs and weights.

b. re-scaling: It keeps the output representations intact when both inputs and weights are randomly scaled.

RMSNorm claims that most of the benefits comes from re-scaling. RMSNorm does re-scaling invariance and regularizes the summed inputs simply according to the root mean square (RMS) statistic.

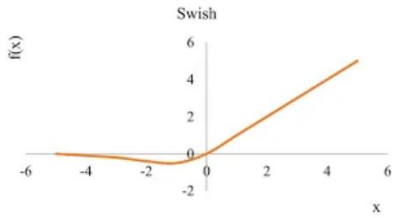
$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}.$$

a_i : activation of i-th neuron.

$g \in \mathbb{R}_n$ is the gain parameter used to re-scale the standardized summed inputs. Intuitively, RMSNorm simplifies LayerNorm by totally removing the mean statistic in LayerNorm.

SwiGLU

To understand SwiGLU activation function we need to understand Swish activation function. Inspiration of using SwiGLU in LLaMA is taken from PaLM.



$$\text{SwiGLU}(x, W, V, b, c, \beta) = \text{Swish}_\beta(xW + b) \otimes (xV + c)$$

Rotary Embeddings (RoPE)

RoPE is a type of position embedding which encodes absolute positional information with rotation matrix and naturally incorporates explicit relative position dependency in self-attention formulation.

Advantage of RoPE:

- Can be expanded to any sequence lengths

- Decaying inter-token dependency with increasing relative distances.
- Capability of equipping the linear self-attention with relative position encoding.

The key idea is to encode relative position by multiplying the context representations with a rotation matrix. RoPE decays with the relative distance increased, which is desired for natural language encoding.

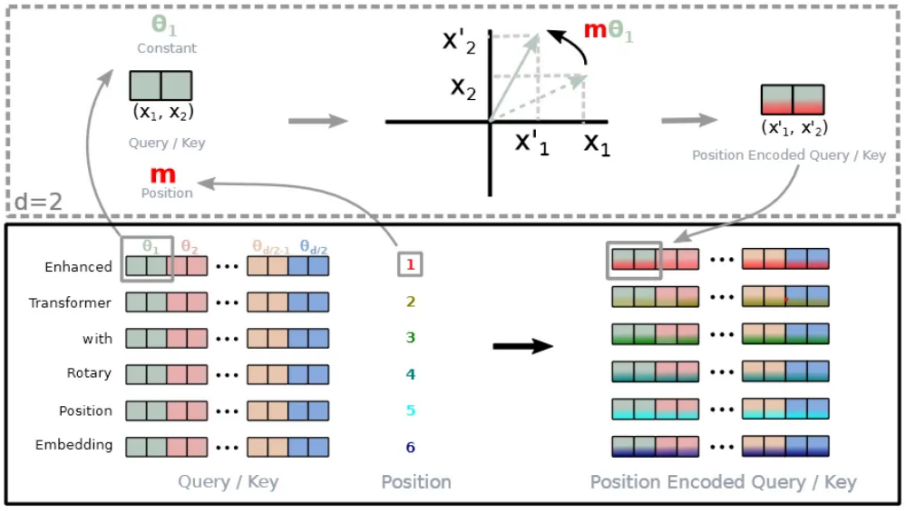


Figure 1: Implementation of Rotary Position Embedding(RoPE).

Inspiration of using RoPE in LLaMA is taken from GPTNeo.

Alpaca

Alpaca is a small but capable 7B language model developed by researchers at Stanford University's Centre for Research on Foundation Models. It was fine-tuned from Meta AI's LLaMA 7B model and trained on 52K instruction-following demonstrations generated in the style of self-instruct using Open AI's text-davinci-003. On the self-instruct evaluation set, Alpaca shows numerous behaviours analogous to Open AI's text-davinci-003 but is also surprisingly small and easy/cheap to reproduce.

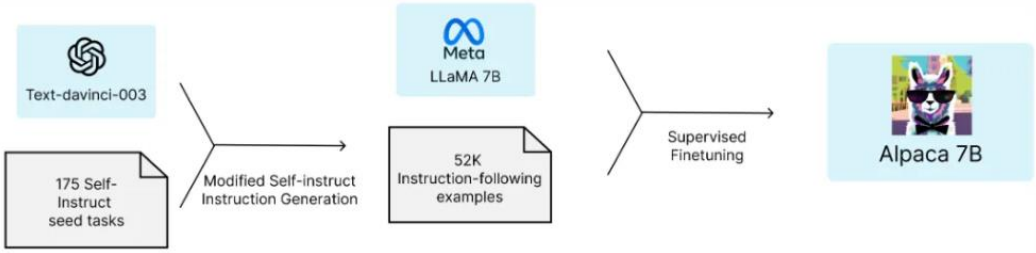


Image from Stanford CRFM

MedAlpaca

MedAlpaca is a large language model meticulously fine-tuned for medical dialogue and QA applications upon the frameworks of Alpaca and Alpaca-LoRA. These models have been trained using an array of medical texts, including medical flashcards, wikis, and dialogue datasets. We use the code and checkpoint from the official library10. We run generation on

two GeForce RTX 2080 GPUs with the following settings: temperature 1.0, max new tokens 512, others are default.

The max number of the main loop, knowledge loop, and response loop is 3. The factuality, consistency, and entailment threshold are -1.0, -5.0, and 0.8, respectively. The demo number for the factuality scorer is 1 for PubMedQA, MEDIQA2019, and LiveMedQA2017; 2 for MASH-QA, and MedQuAD.

2. Conceptual

Training Data

The training data for this project was sourced from various resources. Firstly, we used Anki flashcards to automatically generate questions, from the front of the cards and anwers from the back of the card. Secondly, we generated medical question-answer pairs from Wikidoc. We extracted paragraphs with relevant headings, and used Chat-GPT 3.5 to generate questions from the headings and using the corresponding paragraphs as answers. This dataset is still under development and we believe that approximately 70% of these question answer pairs are factual correct. Thirdly, we used StackExchange to extract question-answer pairs, taking the top-rated question from five categories: Academia, Bioinformatics, Biology, Fitness, and Health. Additionally, we used a dataset from ChatDoctor consisting of 200,000 question-answer pairs, available at https://github.com/Kent0n-Li/ChatDoctor.

Source	n items
ChatDoc large	200000
wikidoc	67704
Stackexchange academia	40865
Anki flashcards	33955
Stackexchange biology	27887
Stackexchange fitness	9833
Stackexchange health	7721
Wikidoc patient information	5942
Stackexchange bioinformatics	5407

Model Training

Models are built upon the LLaMA (Large Language Model Meta AI) foundation models. LLaMA represents a cutting-edge large language model released by Meta, demonstrating their commitment to open science. It is available in various sizes, including 7 billion, 13 billion, 33

billion, and 65 billion parameters. In this study, we fine-tuned the 7 and 13 billion parameter LLaMA variants.

We trained each model for five epochs, employing a learning rate of $2e^{-5}$ for the 7b model and $1e^{-5}$ for the 13b model, using a cosine learning rate scheduler. Gradient accumulation facilitated training with an effective batch size of 256. Given that this training impacts all model parameters, the hardware requirements are substantial. Consequently, we explored alternative training procedures.

First, we implemented Low-Rank Adaptation (LoRA) for weight updates to adapt the pre-trained language models to our specific tasks. LoRA is a method that involves freezing the pre-trained model weights and incorporating trainable rank decomposition matrices into each layer of the Transformer architecture. This approach substantially diminishes the number of trainable parameters and GPU memory requirements for downstream tasks, making it more efficient compared to full fine-tuning and significantly reducing training time.

To further decrease memory and compute demands, we employed 8-bit matrix multiplication for the feed-forward and attention projection layers, along with an 8-bit optimizer. All models trained with LoRA underwent three epochs of training at a learning rate of $2e^{-5}$.

Evaluation Procedure

To evaluate the performance of the fine-tuned language models, we devised an assessment methodology centered on their zero-shot performance across the United States Medical Licensing Examination (USMLE) Step 1, Step 2, and Step 3 self-assessment datasets. We excluded all questions containing images, as our primary interest lies in the models' language capabilities, and they lack visual abilities. We instructed the models to present answers in the format "Option: Answer" (e.g., "A: Penicillin"). If a model's output did not adhere to this format, they were prompted up to five times until the response was generated in the desired format. If the model failed to provide the response in the desired format, the last response was retained.

Interestingly, most of the fine-tuned models typically produced answers in the correct format after the first prompt, while only the base LLaMA models required multiple prompts. We conducted separate evaluations for each model, measuring their accuracy on the USMLE Step 1, Step 2, and Step 3 datasets individually. This approach allowed usto gain a comprehensive understanding of the models' performance across the various stages of the medical licensing examination.

3. Practical

To evaluate the performance of the model on a specific dataset, you can use the Hugging Face Transformers library's built-in evaluation scripts. Please refer to the evaluation guide for more information. Inference

You can use the model for inference tasks like question-answering and medical dialogues using the Hugging Face Transformers library. Here's an example of how to use the model for a question-answering task:

```
from transformers import pipeline



pl = pipeline("text-generation", model="medalpaca/medalpaca-13b", tokenizer="medalpaca/medalpaca-13b")

question = "What are the symptoms of diabetes?"

context = "Diabetes is a metabolic disease that causes high blood sugar. The symptoms include increased thirst, frequent urination, and unexplained weight loss."

answer = pl(f"Context: {context}\n\nQuestion: {question}\n\nAnswer: ")

print(answer)
```

This code uses the Hugging Face Transformers library to create a text generation pipeline. The specific model used for text generation is "medalpaca/medalpaca-13b," and its associated tokenizer is also specified.

Here's a breakdown of the code:

from transformers import pipeline: Import the pipeline class from the Hugging Face Transformers library.

pl = pipeline("text-generation", model="medalpaca/medalpaca-13b", tokenizer="medalpaca/medalpaca-13b"): Create a text generation pipeline (pl) using the model "medalpaca/medalpaca-13b" and its associated tokenizer. This pipeline is configured for text generation tasks.

question = "What are the symptoms of diabetes?": Define a question related to the context.

context = "Diabetes is a metabolic disease that causes high blood sugar. The symptoms include increased thirst, frequent urination, and unexplained weight loss.": Provide a context related to the question. This context will be used as input for text generation.

answer = pl(f"Context: {context}\n\nQuestion: {question}\n\nAnswer: "): Use the text generation pipeline to generate an answer based on the provided context and question. The input to the pipeline is a formatted string that includes the context and question.

print(answer): Print the generated answer.

In summary, this code demonstrates how to use a pre-trained text generation model to generate an answer given a context and a question. The specific model used is "medalpaca/medalpaca-13b," which is a large language model capable of generating coherent and contextually relevant text based on the input provided.

4. References

- Han, T. et al. (2023). MedAlpaca – An Open-Source Collection of Medical Conversational AI Models and Training Data. https://arxiv.org/abs/2304.08247v2.
- Frieder, S. et al. (2023). Large Language Models for Mathematicians. https://arxiv.org/abs/2312.04556v1.
- Kumar, A. (2023). LLaMA: Concepts Explained (Summary). https://akgeni.medium.com/llama-concepts-explained-summary-a87f0bd61964.
- Ji, Z. et al. (2023). Towards Mitigating Hallucination in Large Language Models via Self-Reflection. Association for Computational Linguistics.
- Alammar, J. (2020). The Illustrated Transformer. https://jalammar.github.io/illustrated-transformer/.