A Self-contained Introduction to Large Language Models

Alice Lin[1] and Chih-Jen Lin[1,2]

[1]National Taiwan University
cjlin@csie.ntu.edu.tw
[2]Mohamed bin Zayed University of Artificial Intelligence

July 22, 2025

Abstract

Large language models (LLMs) are a significant technique in Artificial Intelligence. There is no shortage of documents describing the basic concepts. This article, as another attempt to give an introduction of LLMs, aims to help beginners with only basic knowledge of machine learning. We try to be self-contained by giving brief explanation to every basic concept. Further, we have a modularized design to start from a high-level overview and gradually get into details of the GPT-2 model.

1 Introduction

LLMs are used by a large amount of people and organizations, across different industries. With a wide range of capabilities, such as analyzing data and generating content, LLMs assist our daily lives in several ways, enhancing performance and improving productivity.

With the popularity of LLMs, many documents are available to describe the basic concepts. Examples include Naveed et al. (2023) and many online pages.[1]

This article, as another attempt to give an introduction of LLMs, aims to help beginners with only basic knowledge of machine learning. We try to be self-contained by giving brief explanation to every basic concept. Further, we have a modularized design. We begin with a high-level descriptions in Section 2, in which we show the concept of next-token prediction without getting into network details. In Section 3, we introduce auto-regressive models, which are essential in training and predicting sequences. Section 4 then gives details of the network architectures, in particular, the transformer blocks. Finally, Section 5 concludes this document.

[1]https://medium.com/@vipul.koti333/from-theory-to-code-step-by-step-implementation-and-code-breakdown-https://jalammar.github.io/illustrated-gpt2/; https://www.understandingai.org/p/large-language-models-explained-with; https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f

Our discuss focuses on the GPT-2 model (Rashed et al., 2019), though most materials apply to other LLM architectures as well.

We also have slides available for the teaching purpose; see `https://www.csie.ntu.edu.tw/~cjlin/papers/LLMs/`.

2 A High-level Overview of LLMs

2.1 LLM Basics

An LLM learns a model that uses the current

pre-context

to predict

the next token.

We give the following example:

Pre-context		The next token
I	$\rightarrow$	am
I am	$\rightarrow$	a
I am a	$\rightarrow$	machine
I am a machine	$\rightarrow$	learning
I am a machine learning	$\rightarrow$	researcher

This setting is called *autoregressive prediction* in machine learning. Assume we have D documents. For document i, let us assume

$\boldsymbol{x}_{i,1:j}$: contexts from 1st to jth words
$\boldsymbol{y}_{ij}$: $(j+1)$st word.

Then we have many

(target, instance)

pairs for training. For example, we have

$\boldsymbol{x}_i$: "I am a machine learning researcher" and

$$\begin{array}{ll} \boldsymbol{x}_{i,1:1} & \text{I} \\ \boldsymbol{x}_{i,1:2} & \text{I am} \\ \vdots & \end{array}$$

From the basic concept of supervised learning, we can solve an optimization problem

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{D} \sum_{j} \xi(f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j}), \boldsymbol{y}_{ij})$$

to obtain the model, where

- $\boldsymbol{\theta}$ is the model parameters,
- $f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j})$ is a function that gives an approximation of the target $\boldsymbol{y}_{ij}$,
- $\xi(\cdot)$ is the loss function.

However, training a supervised learning model requires that

$$\boldsymbol{y}_{ij} \text{ and } \boldsymbol{x}_{i,1:j}$$

are *vectors*, but they are now word sequences. Having vectors is important because, for example, we can then define a simple loss function such as

$$\|\boldsymbol{y}_{ij} - f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j})\|^2$$

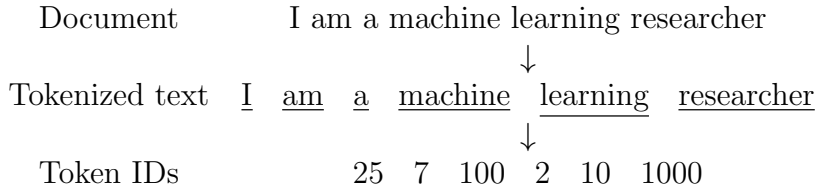
to ensure that $\boldsymbol{\theta}$ leads to an output as close to $\boldsymbol{y}_{ij}$ as possible. From what we learned in supervised learning, if instances and targets are *vectors*, and we agree that the following procedures can be implemented:

- network architectures (i.e., our f function),
- stochastic gradient methods,
- automatic differentiation,

then we "believe" that an LLM can be implemented. We will explain how we can convert $(\boldsymbol{y}_{ij}, \boldsymbol{x}_{i,1:j})$ to vectors. Subsequently, we abuse the notation a bit so that $\boldsymbol{x}_{i,1:j}$ is a word sequence as well as the converted vector.

2.2 Tokenization

We split each document to several tokens and map each token to an integer ID. An example is as follows:



We let "Vocabulary" be the huge dictionary of all words (tokens) considered. Thus, |Vocabulary| is its size. For easy understanding, we consider a word as a token. In practice, we often break a word to several tokens.

If each word is a token, for every word in the sentence we can check the corresponding ID in the Vocabulary dictionary:

the **25th** word in our dictionary is **"I"**,
the **7th** word in our dictionary is **"am"**,
the **100th** word in our dictionary is **"a"**,
the **2nd** word in our dictionary is **"machine"**,
the **10th** word in our dictionary is **"learning"**,
the **1000th** word in our dictionary is **"researcher"**.

2.3 Position Information

We can let

$$\boldsymbol{x}_{i,1:j} \in \{0,1\}^{|\text{Vocabulary}|} \tag{1}$$

be an indicator vector to reflect the occurrence of its words. For example, if

$$\boldsymbol{x}_{i,1:2} \text{ is "I am"} \tag{2}$$

then we have

$$\boldsymbol{x}_{i,1:2} = [\overbrace{\underbrace{0,\ldots,0,1}_{7\text{th}},0,\ldots,0,1}^{25\text{th}},0,\ldots]^T.$$

However, from the indicator vector and the Vocabulary, we can extract only the set of words in $\boldsymbol{x}_{i,1:j}$ instead of the word sequences. That is, we cannot recover the *order* of the words. We need another indicator vector to reveal the *position* information. To do so, first we choose a maximum length T. No matter how long the document is, we always consider only up to the Tth word. Then we can expand the indicator vector in (1) to

$$\boldsymbol{x}_{i,1:j} \in \{0,1\}^{|\text{Vocabulary}|\times T}$$

so that

$$\boldsymbol{x}_{i,1:j} = \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ \vdots \end{array} \begin{array}{c} \begin{array}{cccccccccc} & & & & 7 & & & & 25 & \\ \end{array} \\ \begin{bmatrix} 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 1 & 0 & \ldots \\ 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & 0 & 0 & \ldots \\ & & & & & \vdots & & & & \\ & & & & & \vdots & & & & \end{bmatrix} \end{array}.$$

Our $\boldsymbol{x}_{i,1:j}$ is now a matrix. If we insist on that the input is a vector, we can do a conversion by, for example, concatenating all columns. With the position information, we can exactly recover our word sequence.

2.4 Output of the Network

For output, assume our network can generate

$$\hat{\boldsymbol{y}}_{ij} = f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j}) \in [0,1]^{|\text{Vocabulary}|}$$

indicating the occurrence probability of each word. Consider the example in (2). To predict the next token, we have

$$\boldsymbol{y}_{ij} \text{ is “a”}$$

from the ground truth of the training data. Thus we hope that our predicted $\hat{\boldsymbol{y}_{ij}}$ satisfies

$$(\hat{\boldsymbol{y}}_{ij})_{100} \approx 1$$

and

$$\text{other elements of } \hat{\boldsymbol{y}}_{ij} \approx 0.$$

2.5 Autoregressive Settings

Up to now what we have is

$$\begin{gathered} f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j}) \in [0,1]^{|\text{Vocabulary}|} \\ \uparrow \\ \text{network } f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j}) \\ \uparrow \\ \boldsymbol{x}_{i,1:j} \in \{0,1\}^{|\text{Vocabulary}| \times T} \end{gathered}$$

Everything looks good so far. However, as described later, an issue is that we treat

$$\boldsymbol{x}_{i,1:1}, \boldsymbol{x}_{i,1:2}, \ldots$$

as *independent vectors*. We have inputs like

$$\begin{array}{ll} \boldsymbol{x}_{i,1:1} & \text{I} \\ \boldsymbol{x}_{i,1:2} & \text{I am} \\ \boldsymbol{x}_{i,1:3} & \text{I am a} \end{array}$$

They are related. For example, $\boldsymbol{x}_{i,1:1}$ is a sub-string of $\boldsymbol{x}_{i,1:2}$, so *we should not treat them as independent vectors*. To have efficient training and prediction, we must have a way so that the same operation is not conducted multiple times. We know

$$\boldsymbol{x}_{i,1:j} \text{ includes } x_{i,1}, x_{i,2}, \ldots, x_{i,j}.$$

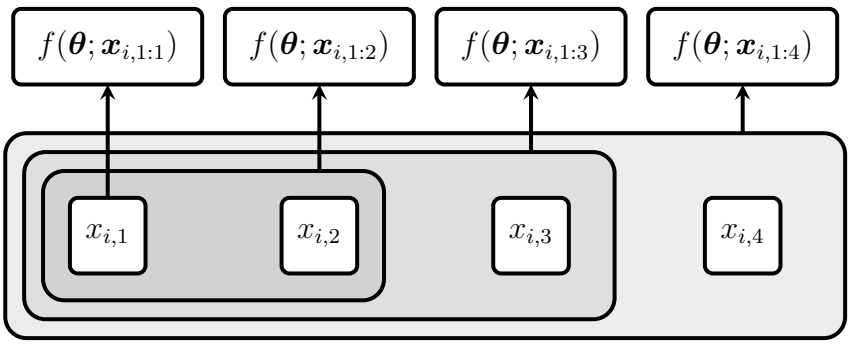


Figure 1: A sequence of next-token predictions

Therefore, we should design $f(\cdot)$ a way so that the following sequence of calculation can be efficiently done.

$$f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:1}), \ldots, f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j}), \ldots \tag{3}$$

That is, we hope things are coupled as indicated in Figure 1. But ensuring that operations used to calculate $f(\boldsymbol{\theta}; \boldsymbol{x}_{1:j-1})$ can also be for $f(\boldsymbol{\theta}; \boldsymbol{x}_{1:j})$ is not an easy task. Subsequently we introduce an auto-regressive setting for our purpose.

2.6 Summary

What we have shown is a high-level illustration of LLM. The remaining task is to design a suitable function f.

3 Auto-regressive Models

3.1 Autoregressive Models

LLM is an autoregressive model, so before giving details of LLM, we discuss basic concepts of autoregressive models. Autoregressive models predict the next component in a sequence by using information from previous inputs in the same sequence. A typical example is time series prediction with applications in stock index prediction, electricity load prediction, etc. Assume our sequence is

$$z_1, z_2, \ldots.$$

The way to train a model is by using data shown in the following table.

training instance	target value
$z_1, \ldots, z_T$	z_{T+1}
$z_1, \ldots, z_{T+1}$	z_{T+2}
$\vdots$	$\vdots$

In practice, data points occurred long time ago may not be important. We can discard them to make training instances have the same number of values:

$$\begin{array}{cc} \text{training instance} & \text{target value} \\ z_1, \ldots, z_T & z_{T+1} \\ z_2, \ldots, z_{T+1} & z_{T+2} \\ \vdots & \vdots \end{array}$$

3.2 LLM Is an Autoregressive Model

The next-token prediction of LLM is a case of auto-regressive settings. Recall we have the setting shown in Figure 1. Note that we aim to have

$$\begin{aligned} f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:1}) &\approx x_{i,2} \\ f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:2}) &\approx x_{i,3} \\ &\vdots \end{aligned}$$

For LLM, the f function is complicated. Thus, we begin with learning how to train a simple auto-regressive model. From the discussion, we will identify important properties to be used for LLM training/prediction.

3.3 Training a Simple Autoregressive Model

Assume we have the following sequence of data

$$z_1, z_2, \ldots$$

and would like to construct a model for one-step ahead prediction. From the observed data, we collect the following (instance, target value) pairs

$$\begin{array}{ll} \boldsymbol{x}_1 = [z_1, \ldots, z_T]^T & y_1 = z_{T+1} \\ \boldsymbol{x}_2 = [z_2, \ldots, z_{T+1}]^T & y_2 = z_{T+2} \\ \vdots & \vdots \end{array}$$

Assume we have collected n training instances. We can then solve a simple least-square regression problem to get a model

$$\min_{\boldsymbol{w}} \sum_{i=1}^{n} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2. \tag{4}$$

Here $\boldsymbol{w}$ includes the model weights. We notice two important properties here. The first property is that we use matrix operations to handle all data together. Specifically, (4) has an analytic solution:

$$\text{optimal } \boldsymbol{w} = (X^T X)^{-1} X^T \boldsymbol{y},$$

where

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \text{ and } X = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_n^T \end{bmatrix} \in R^{n \times T}.$$

For simplicity, we assume that $X^T X$ is invertible. We see that even though y_{T+1} is the target value of the first instance, it is also a feature of the second training instance. Our setting allows the model building by efficient matrix operations. That is, we handle all training data together, even though there are some auto-regressive relationships between them. The reason we can do this is because *our prediction function on training data is the same as the one we use for future prediction.* In testing, for a vector $\boldsymbol{x}$ containing past information, we use $\boldsymbol{w}^T \boldsymbol{x}$ to get our prediction. In training, for any $\boldsymbol{x}_i$, in (1) we use the same way to hope that $\boldsymbol{w}^T \boldsymbol{x}_i$ is close to y_i. This is the second crucial property we will use in our LLM design.

4 The Architecture of GPT-2

4.1 Network Architecture

4.1.1 Network Design

Recall that we hope to have a function f that can efficiently calculate

$$f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:1}), \ldots, f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j}), \ldots$$

as shown in Figure 1. The situation is similar to the least-square approximation discussed earlier for time-series prediction. The difference is that instead of a linear function, here we use a more complicated one. Different types of neural networks can serve as our f. For example, we can modify *convolutional networks* as the main component of our f. See details in, for example, `https://dlsyscourse.org/slides/transformers.pdf`. We will describe the most used network for LLMs: transformer. However, it is possible that in the future we can develop better networks.

4.1.2 Overall Architecture

The architecture used by LLMs contains several transformer blocks. Transformer (Vaswani et al., 2017) is an effective network for many applications. The overall architecture of an LLM is as follows.

SoftMax: next token probability

↑

Linear layer

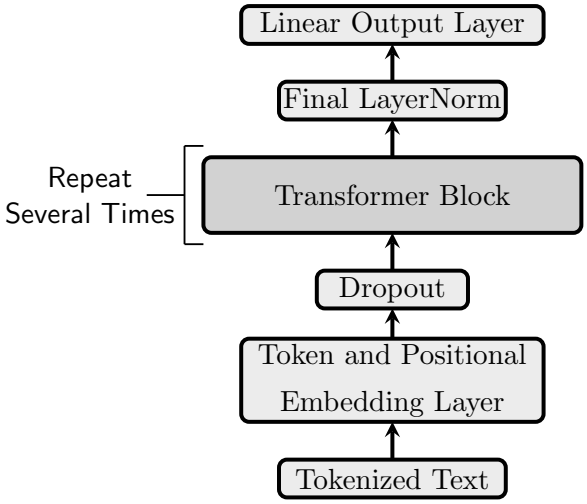
↑

Several transformer blocks

↑

Token + position embedding

We specifically discuss the architecture used in the implementation in GPT-2-small (Radford et al., 2019). Their code is at `https://github.com/openai/gpt-2/blob/master/src/model.py`. Similar architectures have been adopted in other places, such as NanoGPT (`https://github.com/karpathy/nanoGPT`). Precisely, what GPT-2 does is the following network.



For the initialization of model weights, some common ways are available. For example, we can randomly draw values from normal distribution with zero mean.

4.1.3 Input and Embedding Vectors

Now let us discuss the input of the network. To begin, we assume that

- each word (token) in our Vocabulary corresponds to an embedding vector, and
- each position of $1, \ldots, T$ corresponds to an embedding vector.

We then combine these two embedding vectors as one vector. Various ways are possible for the combination. For example, we can concatenate the two vectors as a longer one. Or we can sum up the two vectors. Then $\boldsymbol{x}_{1:T}$ becomes the following matrix.

$$\begin{matrix} x_1 \\ \vdots \\ x_T \end{matrix} \begin{bmatrix} \text{——} \\ \vdots \\ \text{——} \end{bmatrix} \in R^{T \times d} \tag{5}$$

where d is the dimension of the embedding vector. This matrix becomes the input of the architecture. Up to now, it seems that we assume all documents have the same length T. Of course, this assumption is in general untrue. What we do is:

- if document length $> T$, use only the first T tokens, and
- if document length $< T$, we add "empty" values after the end of the document.

The main reason of using a fixed document length is for easily conducting operations. Totally we have

$$|\text{Vocabulary}|$$

vectors for word embedding, and

$$T$$

vectors for position embedding. For each of the T words (tokens) in the document, we extract

a word embedding vector

and

a position embedding vector.

All these embedding vectors are trainable parameters.

4.2 Transform Blocks and Other Details

4.2.1 A Transformer Block

For each transformer block, we let

- Z be the input matrix, and
- Z^{out} be the output matrix.

We manage to have that

$$Z^{\text{out}} \in R^{T \times d}$$

has the same dimensionality as Z. By doing so, the output can be the input of the next block. We repeat this process for several blocks. Typically a transformer block involves

- a multi-head attention layer, and
- feed-forward layers.

Usually, we surround each of the two components with a normalization layer and a residual connection. Thus the mathematical operations in a block are as follows.

$$\tilde{Z} = \text{LayerNorm}(Z) \tag{6}$$

$$Z \leftarrow Z + \text{DropOut}(\text{MultiHead}(\tilde{Z})) \tag{7}$$

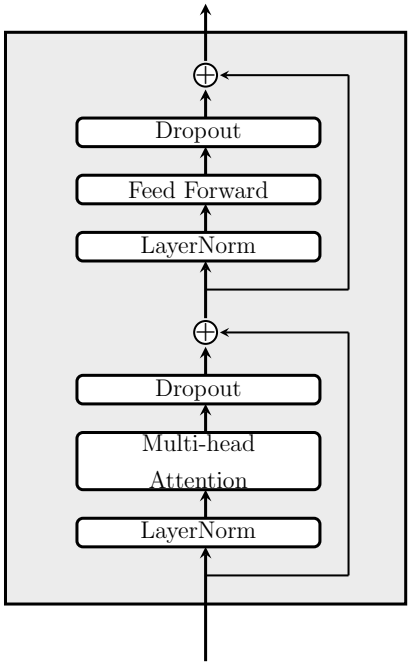
$$\tilde{Z} = \text{LayerNorm}(Z) \tag{8}$$

$$Z^{\text{out}} = Z + \text{DropOut}(\text{GELU}(\tilde{Z}W_1)W_2) \tag{9}$$

We can re-write (21)-(7) in just one line:

$$Z \leftarrow Z + \text{DropOut}(\text{MultiHead}(\text{LayerNorm}(Z)))$$

but to have a symbol $\tilde{Z}$ as the input of the attention layer, we use two equations. Subsequently we discuss each operation in detail. Besides the attention layer in (7), what else in one transform block (e.g., layer normalization or dropout) may slightly vary across LLM implementations. Our operations in (21)-(9) can be illustrated in the following figure.[2]



In the figure, $\oplus$ means the residual connection, which will be explained later.

[2]Modified from `https://sebastianraschka.com/pdf/slides/2024-acm.pdf`

4.2.2 Transformer and Attention

Eq. (7) is the core of a transformer block: a multi-head self-attention layer. We start with discussing single-head attention. If the input matrix is

$$\tilde{Z} \in R^{T \times d},$$

the attention operation is

$$\text{SoftMax}(\frac{\tilde{Z}W_Q W_K^T (\tilde{Z})^T}{\sqrt{d}})\tilde{Z}W_V. \tag{10}$$

We consider three trainable weight matrices

$$W_Q \in R^{d \times d}, W_K \in R^{d \times d}, W_V \in R^{d \times d}$$

to convert the input matrix $\tilde{Z}$ to

$$\tilde{Z}W_Q \in R^{T \times d}, \quad \tilde{Z}W_K \in R^{T \times d}, \quad \tilde{Z}W_V \in R^{T \times d}.$$

In (10), we can combine $W_Q W_K^T$ as one single weight matrix. However, people still write them separately because in more general situations, we may consider

$$W_Q \in \mathbb{R}^{d \times d'} \text{ and } W_K \in \mathbb{R}^{d \times d'}$$

with $d' < d$. That is, $W_Q W_K^T$ becomes a low-rank approximation of the $d \times d$ weight matrix. Then we need two matrices instead of one. We will see this situation in multi-head attention. In (10), the SoftMax function is applied on each row $\boldsymbol{z}$ of an input matrix in the following way.

$$\text{SoftMax}(\mathbf{z}) = \begin{bmatrix} \frac{\exp(z_1)}{\sum_j \exp(z_j)} \\ \vdots \\ \frac{\exp(z_T)}{\sum_j \exp(z_j)} \end{bmatrix}. \tag{11}$$

Let us briefly talk about the attention operation in (10). If the SoftMax$(\cdot)$ part is not there, then (10) reduces to

$$\tilde{Z}W_V,$$

which is no more than a feed-forward operation. What

$$\text{SoftMax}(\frac{\tilde{Z}W_Q W_K^T (\tilde{Z})^T}{\sqrt{d}})$$

give are weights for the T words in $\tilde{Z}$. Thus in (10) we do a *weighted average* of words in $\tilde{Z}$. The purpose is to transform the representation of each word based on its relationship to other words in the same document. We do not get into details here because our focus is not on explaining the attention mechanism.

In practice, we extend the single-head attention to multi-head for capturing different types of relationships between words in the document. Specifically, we combine results of h heads:

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O, \tag{12}$$

where

$$\text{head}_i = \text{SoftMax}(\frac{\tilde{Z}W_Q^i(W_K^i)^T(\tilde{Z})^T}{\sqrt{d}})\tilde{Z}W_V^i \in R^{T\times d/h}. \tag{13}$$

Due to the use of h heads, we now have

$$W_Q^i \in R^{d\times d/h}, W_K^i \in R^{d\times d/h}, W_V^i \in R^{d\times d/h}. \tag{14}$$

Earlier we talked about if $W_Q^i(W_K^i)^T$ can become just one matrix. We cannot do that here because W_Q^i and W_K^i are no longer squared matrices. In (12), Concat() is a function which concatenates matrices together. That is,

$$\begin{aligned} & \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \\ = \; & [\text{head}_1, \ldots, \text{head}_h] \in R^{T\times d}. \end{aligned}$$

Another advantage of using multiple heads is that the computations for head_i, $\forall i$ can be done in parallel. We further have in (12) that

$$W_O \in R^{d\times d}. \tag{15}$$

The use of W_O is like we have a linear layer after concatenation.

4.2.3 Layer Normalization

First let us give the mathematical operations in (21) and (8). For any row $\boldsymbol{z}$ of the input matrix, the normalized row is

$$\text{Normalize}(\boldsymbol{z}) = \boldsymbol{a} \odot \frac{\boldsymbol{z} - \text{mean}(\boldsymbol{z})}{\text{std}(\boldsymbol{z})} + \boldsymbol{b}, \tag{16}$$

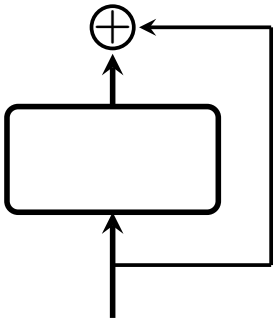
where $\text{mean}(\cdot)$ and $\text{std}(\cdot)$ are the mean and standard deviation, and $\odot$ means the component-wise product between two vectors. In (16),

$$\boldsymbol{a} \in R^d, \boldsymbol{b} \in R^d$$

are learnable parameters shared across rows of the input matrix. The reason of applying layer normalization is to avoid too large or too small gradient values. In deep learning, we have operations across layers to calculate the gradient. Such a long sequence of operations may cause very large or small values (think about the multiplication of several numbers). Several techniques are available to address this issue of too large or too small gradient values, and layer normalization is one of them. As we can see, the normalization operation in (16) avoids values being extreme.

4.2.4 Residual Connections

Residual connection is another technique to make the problem of too small and too large gradient values less serious. It also improves the overall training stability. The operation is simply to sum the input and output of one (or several) neural network layer. Usually we use the following flowchart to represent residual connections:



The residual connection can be applied to any network layer with the same input/output dimensionality.

4.2.5 Dropout Operations

Dropout is a technique to prevent overfitting in neural networks. From Srivastava et al. (2014), "the key idea is to randomly drop units (along with their connections) from the neural network during training." Under each mini-batch of the stochastic gradient method, the "thinned" network is fixed so we can do the sub-gradient calculation. Thus, in different batches, the networks are slightly different. It is like that we are doing an ensemble of several networks. Therefore, we need a rate p as the probability that a neuron is retained. In the prediction stage, we do not remove any neurons or their connections. Instead, we multiply every weight of the dropout layer by the rate p. In some places (e.g., PyTorch), p means the rate of elements being removed. Thus, in the prediction stage we should multiply every weight of the dropout layer by $1 - p$. Interestingly, what PyTorch does is to scale the training output by a factor $1/(1 - p)$, so in prediction, the dropout layer does not do anything.[3]

Dropout has been used in different types of architectures. For example, in some implementations such as Nano GPT, a attention dropout is applied. Specifically, after the softmax function in (11), some elements in the $R^{d \times d}$ matrix are not used.

[3]This seems to be what GPT-2 has done. They release only the prediction code, in which we do not see any dropout operation.

4.2.6 Feed-forward Layers

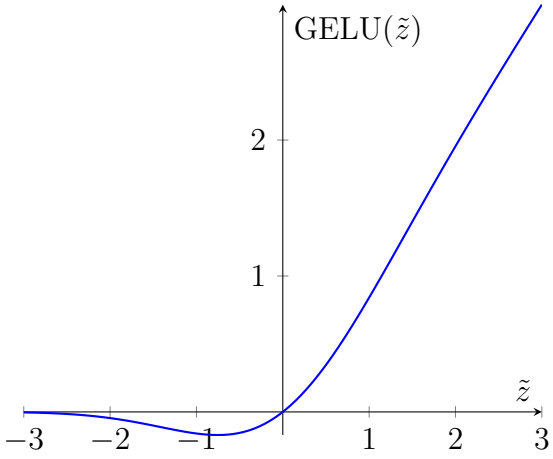
In each transformer block, after multi-head attention, GPT-2 considers two feed-forward layers. In (9), the GELU (Gaussian Error Linear Units) activation function is as follows.

$$\text{GELU}(\tilde{z}) = \tilde{z} \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{\tilde{z}}{\sqrt{2}}\right)\right], \tag{17}$$

where $\text{erf}(\tilde{z})$ denotes the error function, defined by:

$$\text{erf}(\tilde{z}) = \frac{2}{\sqrt{\pi}} \int_0^{\tilde{z}} e^{-t^2}\, dt.$$

The GELU function shown below is a smooth version of the ReLU activation function.



In GPT-2,

$$W_1 \in R^{d\times 4d} \text{ and } W_2 \in R^{4d\times d}. \tag{18}$$

From (9), we see that to multiply with $\tilde{Z}$ and to have the same output size as the input, the number of rows of W_1 and the number of columns of W_2 must be both d. However, the choice of $4d$ appears to be arbitrary.

4.2.7 Final Linear Output Layer

After all transformer blocks, we get an output matrix:

$$Z^{\text{out}} \in \mathbb{R}^{T\times d}.$$

We must convert each row vector to an index in the Vocabulary set as our next-word prediction. To this end, we have a final linear layer with weight matrix

$$W^{\text{final}} \in \mathbb{R}^{d\times|\text{Vocabulary}|}.$$

Then from

$$Z^{\text{out}} \times W^{\text{final}} \in \mathbb{R}^{T\times|\text{Vocabulary}|}, \tag{19}$$

for every token in $1, \ldots, T$, we select the index corresponding to the largest of the |Vocabulary| values as the prediction. Interestingly, people use the same word embedding vectors for the input matrices as the weights of the final linear layer. Recall we said that all word and position embedding vectors are trainable parameters. By this setting, instead of two $|\text{Vocabulary}| \times d$ matrices, we save the space by using only one.

4.2.8 Number of Parameters

In large language models, people often show the number of parameters to reflect the model size. For example, the total number of the model "GPT2-small" is said to have around 124 millions of parameters. We show details to calculate the number of parameters. To do so, we check weights in different parts of an LLM model:

- weights in transformer blocks,
- weights in the final linear layer, and
- weights in the input matrices.

In each transformer block, from (14), (15), and (18), we have

$$W_Q^i \in R^{d \times d/h}, W_K^i \in R^{d \times d/h}, W_V^i \in R^{d \times d/h}, i = 1, \ldots, h,$$

$$W_O \in R^{d \times d},$$

and

$$W_1 \in R^{d \times 4d}, \, W_2 \in R^{4d \times d}.$$

Thus, the total number is

$$\begin{aligned} & 4 \times d^2 + 4 \times d^2 + 4 \times d^2 \\ =& 12 \times d^2. \end{aligned}$$

For the final linear layer, the number of weights is

$$|\text{Vocabulary}| \times d.$$

Now let us check the remaining parts. The input matrix is the combination of two parts: token embedding and position embedding. Recall we said that all word and position embedding vectors are trainable parameters. For token embedding, because according to document contents, we find each token's corresponding embedding, the space needed is

$$|\text{Vocabulary}| \times d.$$

However, we have mentioned that the same weights are used for the final linear layer, so no extra space is needed. For position embedding, because we have T possible positions, the number of weights is

$$T \times d.$$

For GPT-2-small:

- Number of attention blocks = 12,
- $d = 768$,
- $|\text{Vocabulary}| = 50{,}257$,[4]
- $T = 1{,}024$.

The sum is

$$\begin{aligned} & 12 \times d^2 \times \text{number of blocks} + |\text{Vocabulary}| \times d + \\ & T \times d \\ = \; & 12 \times 768^2 \times 12 + 50{,}257 \times 768 + 1{,}024 \times 768 \\ = \; & 124{,}318{,}464 \approx 124 \text{ Million}. \end{aligned}$$

The GPT-2 paper (Rashed et al., 2019) wrongly stated that the number of parameters is 117 million, though later they stated 124 million in other places.

4.3 Masked Self-Attention

4.3.1 Next-token Prediction

Recall that we have a sequence of next-token predictions; see Figure 1. However, in the earlier description of the architecture for training, we may not always take this situation into account. For example, in our self-attention operation, we calculate the relationship between *all words* in the word sequence. This situation violates a condition mentioned earlier for training an auto-regressive model: the training decision function should be the same as the prediction decision function. In self-attention, we should sequentially do

$$\text{SoftMax}(\frac{\left[\text{“I”}\right] W_Q W_K^T \left[\text{“I”}\right]^T}{\sqrt{d}})_{1\times 1} \left[\text{“I”}\right]_{1\times d} W_V$$

$$\text{SoftMax}(\frac{\begin{bmatrix}\text{“I”}\\ \text{“am”}\end{bmatrix} W_Q W_K^T \begin{bmatrix}\text{“I”}\\ \text{“am”}\end{bmatrix}^T}{\sqrt{d}})_{2\times 2} \begin{bmatrix}\text{“I”}\\ \text{“am”}\end{bmatrix}_{2\times d} W_V$$

$$\vdots$$

[4] In some subsequent implementation, |Vocabulary| is increased to 50,304, the nearest multiple of 64 for efficiency.

The reason is that to have

$$\text{input: “I”}$$

and

$$\text{output: “am,”}$$

which corresponds to

$$x_{i,1} \to f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:1}) \approx x_{i,2}$$

in Figure 1, we can only calculate the word relationships of the current input document. Now "I" is the only word in our document. Therefore, the suitable setting is different from what we did earlier.

4.3.2 Masked Self-attention

The formulation we gave earlier was

$$\begin{bmatrix} \text{“I”} \\ \vdots \\ \text{“researcher”} \end{bmatrix} W_Q W_K^T \begin{bmatrix} \text{“I”} \\ \vdots \\ \text{“researcher”} \end{bmatrix}^T \in R^{T \times T}.$$

To be consistent with the prediction, what we should use is only the *lower triangular* part of the above matrix:

$$\begin{bmatrix} (1,1) & & & \\ (2,1) & (2,2) & & \\ \vdots & \vdots & \ddots & \\ (T,1) & \cdots & \cdots & (T,T) \end{bmatrix}.$$

Therefore, in the training procedure, we must *mask* all entries above the diagonal. In practice, people just assign these entries to $-\infty$:

$$\begin{bmatrix} (1,1) & -\infty & \cdots & -\infty \\ (2,1) & (2,2) & -\infty & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ (T,1) & \cdots & \cdots & (T,T) \end{bmatrix}.$$

Then in the SoftMax operation, because

$$e^{-\infty} = 0,$$

we have the desired matrix.

4.3.3 Feed-forward Layers

An interesting question is if we have the same issue in other operations. Let us briefly check the feed-forward layers. Recall in (9) we calculate

$$\text{GELU}(\tilde{Z}W_1)W_2. \tag{20}$$

To have the same setting as prediction, we should sequentially do

$$\begin{aligned}
&\text{GELU}(\big[\text{“I”}\big]_{1\times d}(W_1)_{d\times 4d})W_2\\
&\text{GELU}(\begin{bmatrix}\text{“I”}\\ \text{“am”}\end{bmatrix}_{2\times d}(W_1)_{d\times 4d})W_2\\
&\vdots
\end{aligned}$$

The operations are precisely the same as those in (20), so we are fine. Up to this point, we see that operations like (20) lead us to have a desired property for training an auto-regressive model: we assemble all the next-token predictions together in the training process. This makes efficient matrix computation for fast training.

4.3.4 Prediction

In (19), we wrote that the final output of the network is

$$Z^{\text{out}} \times W^{\text{final}} \in \mathbb{R}^{T\times|\text{Vocabulary}|}.$$

Then for every token in $1,\dots,T$, we select the index corresponding to the largest of the $|\text{Vocabulary}|$ values as the prediction. However, this setting is for training. For our given sequence,

$$z_1, z_2, \dots.$$

we have the following training and target pairs

training instance	target value
z_1	z_2
z_1, z_2	z_3
$\vdots$	$\vdots$

We consider all of them *together* However, in prediction, each time we consider only one pair. That is,

$$z_1,\dots,z_T \rightarrow \text{ an estimate of } z_{T+1}$$

Therefore, if

$$Z^{\text{out}} = \begin{bmatrix}(\boldsymbol{z}_1^{\text{out}})^T\\ \vdots \\ (\boldsymbol{z}_T^{\text{out}})^T\end{bmatrix},$$

all we need is

$$(\boldsymbol{z}_T^{\text{out}})^T \times W^{\text{final}} \in \mathbb{R}^{1\times|\text{Vocabulary}|}.$$

To get $\boldsymbol{z}_T^{\text{out}}$, let us check the needed operations. Consider the input

$$Z = \begin{bmatrix} \boldsymbol{z}_1^T \\ \vdots \\ \boldsymbol{z}_T^T \end{bmatrix}.$$

Eqs. (21)-(9) become

$$\begin{aligned} \tilde{\boldsymbol{z}}_T^T &= \text{LayerNorm}(\boldsymbol{z}_T^T) \\ \boldsymbol{z}_T^T &\leftarrow \boldsymbol{z}_T^T + \text{DropOut}(\text{MultiHead}(\tilde{\boldsymbol{z}}_T^T)) \\ \tilde{\boldsymbol{z}}_T^T &= \text{LayerNorm}(\boldsymbol{z}_T^T) \\ \boldsymbol{z}_T^{\text{out}} &= \boldsymbol{z}_T^T + \text{DropOut}(\text{GELU}(\tilde{\boldsymbol{z}}_T^T W_1)W_2) \end{aligned}$$

5 Conclusions

In this article, we provide an introduction to LLMs that is simple to understand. Starting from the concept of LLMs to an explaination of GPT-2's architecture, the article is easy to follow through. As LLMs are still evolving and will likely become an even more powerful tool in the future, this article hopes to help more people have a better understanding of LLMs.

Acknowledgements

This work was supported by National Science and Technology Council of Taiwan grant 110-2221-E-002-115-MY3. The authors thank Ming-Wei Chang for some valuable discussion in the early stage of writing this document.

References

H. Naveed, A. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Barnes, and A. Mian. A comprehensive overview of large language models, 07 2023.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners, 2019.

A. Rashed, J. Grabocka, and L. Schmidt-Thieme. Multi-label network classification via weighted personalized factorizations, 2019.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15 (1):1929–1958, 2014.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. 2017.