The Mathematics Underlying Transformers and ChatGPT*

Yongge Wang (UNC Charlotte)

December 7, 2023

Abstract

Since the inception of the transformer deep learning model, as outlined in the 2017 paper titled "Attention Is All You Need" [19], it has risen to prominence and now boasts widespread applications across various domains. Notably, the transformer architecture has found remarkable success in applications such as ChatGPT. In this tutorial, we aim to demystify the mathematical principles underpinning the transformer architecture.

1 Gradient descent

In the realm of machine learning, we frequently encounter the task of identifying a local minimum of a differentiable function $f(x)$. It's worth highlighting that the function $f(x)$ attains its minimum value at a specific point x_0 only if the derivative of $f(x)$ at x_0 equals zero. Consequently, the problem can be simplified by seeking the root or zero-point of the derivative function $f'(x)$. Newton's root-finding algorithm may be used to approximate the roots of a real-valued function $f(x)$. The most basic version starts with an initial guess value x_0 of the root. Then it continously computes

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

until the value $|x_{n+1} - x_n|$ is small enough.

In practical applications, Cauchy's gradient descent method is commonly utilized to find a local minimum of a differentiable function $f(x)$. This algorithm entails iteratively moving in the direction opposite to the gradient of the function at the current point until hopefully it converges to a value of x that minimizes $f(x)$. As an illustrative example for a single-variable function $f(x)$, if we begin from the point x_0 , we can establish the following relationship for $n > 0$:

$$x_{n+1} = \begin{cases} x_n - \delta_n & \text{if } f'(x_n) > 0 \\ x_n + \delta_n & \text{if } f'(x_n) < 0 \end{cases} \tag{1}$$

where $\delta_n > 0$ are small values. Alternatively, equation (1) can be conveniently expressed as

$$x_{n+1} = x_n - \gamma \cdot f'(x_n) \tag{2}$$

where $\gamma > 0$ is a learning rate.

The gradient descent method for multi-variable functions works in a similar manner to the example above for single-variable functions. For a function $f(\mathbf{x})$ of multiple variables with $\mathbf{x} = (x_1, \cdots, x_m)^T$, the gradient of $f(\mathbf{x})$ at a point $\mathbf{a}$ is defined as follows:

$$\nabla f(\mathbf{a}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{a}), \cdots, \frac{\partial f}{\partial x_n}(\mathbf{a}) \right)$$

*ChatGPT3.5 has been used to improve the presentation of this paper. In particular, after we finish the draft of each paragraph, we submit the paragraph to ChatGPT3.5 to re-write the paragraph with better English presentation.

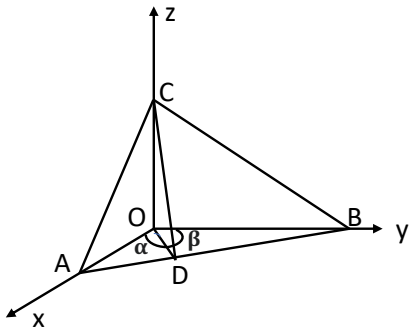

Figure 1: Gradient example

For multi-variable functions, the equation (2) may be expressed as follows:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \cdot \nabla f(\mathbf{x}_n). \tag{3}$$

Here, $\gamma_n > 0$ represents positive step sizes that are small enough and are commonly referred to as learning rates.

Example 1 *We'll illustrate the concept of a two-variable function's gradient using an example. In Figure 1, imagine a plane labeled as ABC serving as the tangent plane to a function $z = f(x, y)$ at the point $(0, 0, C)$. To clarify, let's denote Δx as the vector OA, Δy as OB, and Δz as OC. In this context, the partial derivatives are defined as $\frac{\partial f}{\partial x} = \frac{\Delta z}{\Delta x}$ and $\frac{\partial f}{\partial y} = \frac{\Delta z}{\Delta y}$.*

Since the plane COD is orthogonal to plane ABC and CD is the shortest distance from the point $(0, 0, C)$ to the line AB., the direction OD points precisely in the opposite direction of the gradient of function $f(x, y)$ at the point $(0, 0, C)$. Assuming that the coordinates of the point D is (x_1, y_1), we have

$$x_1 = OD \cdot \cos\alpha = OD \cdot \frac{OD}{\Delta x} = \frac{OD^2}{\Delta x} = \frac{OD^2}{\Delta z}\frac{\Delta z}{\Delta x} = \frac{OD^2}{\Delta z}\frac{\partial f}{\partial x}$$

and

$$y_1 = OD \cdot \cos\beta = OD \cdot \frac{OD}{\Delta y} = \frac{OD^2}{\Delta y} = \frac{OD^2}{\Delta z}\frac{\Delta z}{\Delta y} = \frac{OD^2}{\Delta z}\frac{\partial f}{\partial y}$$

From this, we deduce the gradient of function f as:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

2 Cost functions, stochastic gradient descent, and softmax

Let's consider a multi-variable function $f(\mathbf{x})$ which we aim to understand. For this function, we know that $y_i = f(\mathbf{x}_i)$ for $i = 1, \cdots, n$ where the set $\{(\mathbf{x}_i, y_i) : i = 1, \cdots, n\}$ constitutes our training dataset. Furthermore, we can express the function $f(\mathbf{x})$ using parameters $\theta = (\theta_1, \cdots, \theta_m)^T$, and for the sake of simplicity, we may represent this function as $f(\theta, \mathbf{x})$ by extending our notations. For instance, in the context of linear regression, we can express the function as $f(\theta, \mathbf{x}) = \theta^T \cdot (\mathbf{x}, 1)$.

Given the training dataset represented as $\{(\mathbf{x}_i, y_i) : i = 1, \cdots, n\}$, we can determine the values of θ that minimize a predefined cost function (also known as a loss function). One common example of such a cost function is the Mean Squared Error (MSE), which is defined as:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \left(y_i - f(\theta, \mathbf{x}_i)\right)^2. \tag{4}$$

To find the optimal θ that minimizes the function $J(\theta)$ as expressed in Equation (4), one can employ gradient descent methods.

Calculating the gradient of the function $J(\theta)$ iteratively at specific points is typically a computationally intensive task. In practical scenarios, an alternative approach involves approximating the gradient using a randomly chosen subset of the data, a method known as stochastic gradient descent (SGD). Instead of using the entire training dataset, one may choose to work with a single sample pair during each iteration. In this setup, at each iteration, a random variable j is selected, and the gradient $\nabla_\theta \left(y_i - f(\theta, \mathbf{x}_i)\right)^2$ is utilized to estimate the gradient $\nabla_\theta J(\theta)$.

To provide a concise summary, stochastic gradient descent for optimizing the value of θ proceeds as follows:

1. Begin with an initial value for θ and select a learning rate γ.
2. Continue iterations until the convergence criteria are satisfied.
   - Randomly shuffle the samples in the training set.
   - For $i \in \{1, \cdots, m\}$, update θ as follows:

$$\theta = \theta - \gamma \cdot \nabla_\theta \left(y_i - f(\theta, \mathbf{x}_i)\right)^2$$

softmax function: The softmax function is a frequently used tool for the normalization of probability distributions. For an N-dimension input vector $\mathbf{x}$, the function transform this vector into a probability distribution, where each probability is directly proportional to the exponential of the corresponding input number. To clarify, prior to applying the softmax function, some components of the input vector may be negative or greater than 1. As a result, the components may not collectively sum up to 1, rendering them unsuitable for interpretation as probabilities. However, after the application of the softmax function, every component of the vector will be confined within the $(0, 1)$ interval, and they will sum up to precisely 1, making them amenable to interpretation as probabilities. Notably, the larger input values will yield correspondingly higher probabilities. More formally, for a vector $\mathbf{x} = (x_1, \cdots, x_N)$, the softmax transformation is defined as

$$\text{softmax}(\mathbf{x}) = (\bar{x}_1, \cdots, \bar{x}_N) \text{ with each } \bar{x}_i \text{ calculated as } \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}.$$

3 Neural networks

First we define the ReLU(rectified linear unit) function:

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Then a single neuron can be described as follows:

$$f(x) = \text{ReLU}(w_1 x + w_0)$$

Here, the ReLU function introduces non-linearity to the linear expression $w_1 x + w_0$ and w_0 is often referred as the bias. A single neuron with multi-variables can be described as

$$f(\mathbf{x}) = \text{ReLU}(\mathbf{w}^T \cdot (\mathbf{x}, 1))$$

A more intricate neural network can be created by combining the solitary neuron mentioned earlier in a stacked fashion, where each neuron transmits its output as input to the next neuron, thus leading to a more sophisticated function. In the realm of machine learning, commonly used neural networks include, but are not limited to, Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformers. CNNs find their primary application in the analysis of visual images and function as feed-forward neural networks. On the other hand, RNNs involve the utilization of hidden states and are typically used for processing sequential data.

As an example, let's consider a sequence of data denoted as $x_1, \cdots, x_n$ and an initial hidden state labeled as h_1. A recurrent neural network (RNN) can be defined by the following function for $i = 1, \cdots, n-1$: $(y_{i+1}, h_{i+1}) = f(h_i, x_i)$. Here, y_{i+1} represents the output, while h_{i+1} represents the updated hidden state. RNN has been extensively used in natural language processing applications until the introduction of transformer-based models.

As another example, in the context of image analysis, it is frequently necessary to employ multiple filters to extract different local features. This could be achieved by linking patches in the input layer to a single neuron in a subsequent layer and employing sliding windows to establish these connections. This specific neural network architecture can be achieved through the use of convolutional neural networks rather than fully connected neural networks.

4 Word embeddings

To achieve effective natural language processing (NLP), one crucial objective is the development of a system enabling computers to grasp the meaning of individual words. The significant breakthrough in this realm came from the pioneering concept introduced by Firth [6] in 1957, suggesting that a word's meaning is shaped by the context in which it appears. This idea is often expressed as "a word is characterized by the company it keeps". This is generally accomplished through word embeddings, which involve the mapping of a vocabulary into numerical vectors in the real number space. These real-valued vectors effectively encode the essence of a word in such a way that when two word representations are closer in the vector space, they are expected to share a similar meaning.

ChatGPT has the capability to utilize pre-existing word embedding schemes, or it can generate its own word embeddings through training. To grasp the rationale behind training word embeddings, we will elucidate the word2vec approach, as discussed in [7, 8, 12]. It's worth noting that this approach is currently deemed obsolete and should be substituted with transformer-based techniques.

The training data used in word2vec consists of a collection of center words w_i and their associated contexts c_i, denoted as $\{(w_i, c_i) : i = 1, \cdots, T\}$. For instance, in a training sentence such as "a word is defined by the company it keeps", any word within this sentence can be regarded as a center word. If we were to choose "company" as the center word, then its associated contexts would be:

$$\{\text{a, word, is, defined, by, the, it, keeps}\}.$$

The straightforward "bag of words" model generates identical predictions for each position, aiming to assign a reasonably high probability to all words within the context. In particular, the training model's objective is to maximize the likelihood function, as initially introduced in the skip-gram model by Mikolov in 2013 [11]:

$$L(\theta) = \prod_{i=1}^{T} \prod_{w_j \in c_i} p(w_j | w_i; \theta) \tag{5}$$

where θ represents the neural network parameters that we aim to optimize, and $p(w_j|w_i;\theta)$ is the conditional probability. The task of maximizing the likelihood function $L(\theta)$ in (5) is essentially equivalent to minimizing the following cost function

$$J(\theta) = -\frac{1}{T} \sum_{i=1}^{T} \sum_{w_j \in c_i} \log p(w_j | w_i; \theta) \tag{6}$$

In word2vec, the conditional probability $p(w_j|w_i;\theta)$ is defined using the softmax functioin as follows:

$$p(w_j | w_i; \theta) = \frac{e^{v_{w_i} \cdot u_{w_j}}}{\sum_{w' \in V} e^{v_{w_i} \cdot u_{w'}}} \tag{7}$$

In this equation, v_w represents the vector representation of the center word w, while u_w denotes the vector representation of the context word w. Moreover, V encompasses the entire vocabulary. It is important to note that two distinct vector representations, namely u_w and v_w, are assigned to each word w.

In the preceding paragraphs, we discussed the primary objective of the training model, which is to reduce the cost function $J(\theta)$ as outlined in Equation (6). Nevertheless, as highlighted in [7], *it remains unclear why minimizing $J(\theta)$ results in the creation of good embeddings for all words in the vocabulary set V. Moreover, it is not evident why defining conditional probabilities as shown in Equation (7) is useful for obtaining high-quality word embeddings; this aspect is essentially an assumption. On the flip side, empirical evidence demonstrates that word2vec has proven to be highly successful in achieving good word embeddings.*

In the following paragraphs, we will provide a concise overview of the procedure for computing the gradient in the model training process. When we merge Equations (6) and (7), it results in the subsequent expression:

$$J(\theta) = -\frac{1}{T}\sum_{i=1}^{T}\sum_{w_j \in c_i}\left(v_{w_i}\cdot u_{w_j} - \log\sum_{w'\in V} e^{v_{w_i}\cdot u_{w'}}\right) \tag{8}$$

Thus we have

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial v_{w_i}} &= -\tfrac{1}{T}\textstyle\sum_{i=1}^{T}\sum_{w_j\in c_i}\left(u_{w_j} - \frac{\partial \log\sum_{w'\in V} e^{v_{w_i}\cdot u_{w'}}}{\partial v_{w_i}}\right) \\
&= -\tfrac{1}{T}\textstyle\sum_{i=1}^{T}\sum_{w_j\in c_i}\left(u_{w_j} - \frac{1}{\sum_{w'\in V} e^{v_{w_i}\cdot u_{w'}}}\frac{\partial \sum_{w'\in V} e^{v_{w_i}\cdot u_{w'}}}{\partial v_{w_i}}\right) \\
&= -\tfrac{1}{T}\textstyle\sum_{i=1}^{T}\sum_{w_j\in c_i}\left(u_{w_j} - \frac{\sum_{w'\in V} e^{v_{w_i}\cdot u_{w'}} u_{w'}}{\sum_{w'\in V} e^{v_{w_i}\cdot u_{w'}}}\right) \\
&= -\tfrac{1}{T}\textstyle\sum_{i=1}^{T}\sum_{w_j\in c_i}\left(u_{w_j} - \sum_{w'\in V} p(w'|w_i;\theta)u_{w'}\right)
\end{aligned} \tag{9}$$

It's important to highlight that in Equation (9), the final term $\sum_{w'\in V} p(w'|w_i;\theta)u_{w'}$ represents the expected probability, with u_{w_j} denoting the observed value. As a result, the model's performance is considered satisfactory when the observed value is close to the expected value.

If we represent each word as a vector in a d-dimensional real number space, we can think of θ as a vector containing $2d|V|$ elements, where θ encompasses all the model parameters. Consequently, the computational cost for calculating and conditional probability in Equation (7) and for calculating $\nabla_\theta J(\theta)$ scales with the size of this vector, which is $2d|V|$. In practice, $|V|$ is often as large as 10^5–10^7. While hierarchical softmax [13] or Noise Contrastive Estimation (NCE) [9] or stochastic grading descent can serve as computationally efficient alternatives to the full softmax in Equation (7), word2vec employs a distinct technique known as negative sampling.

In the following paragraphs, we will describe the negative sampling technique outlined in [7]. Let's define D as the collection of all pairs comprising a center word and its associated context that we extract from the training text. Consider a pair (w, c), representing a center word and its context. We use $p(D = 1|w, c; \theta)$ to denote the probability that this particular (w, c) pair originated from the corpus data. In a corresponding manner, $p(D = 0|w, c; \theta) = 1 - p(D = 1|w, c; \theta)$ represents the probability that (w, c) did not originate from the corpus data. Rather than optimizing the likelihood function $L(\theta)$ as shown in Equation (5), word2vec aims to maximize the likelihood that all observations are indeed derived from the dataset:

$$L'(\theta) = \prod_{(w,c)\in D} p(D = 1|w, c; \theta) \tag{10}$$

The authors in [7, 12] provide a definition for the probability $p(D = 1|w, c; \theta)$ using sigmoid functoin as follows. *However, it is unclear whether this meets the definition of probability, as the total probability sum may not necessarily equal one*:

$$p(D = 1|w, c; \theta) = \frac{1}{1 + e^{-v_c\cdot v_w}}.$$

Then the task of maximizing the likelihood $L'(\theta)$ is equivalent to minimizing the following cost function

$$J'(\theta) = -\sum_{(w,c)\in D}\log\frac{1}{1 + e^{-v_c\cdot v_w}} = \sum_{(w,c)\in D}\log\left(1 + e^{-v_c\cdot v_w}\right) \tag{11}$$

It should be emphasized that achieving optimal results for $J'(\theta)$ is straightforward when we configure θ in a way that ensures $p(D = 1|w, c; \theta)$ equals 1 for all pairs (w, c). To tackle this issue, a solution is to

create a random dataset, denoted as D', which comprises mismatched word-context pairs, often referred to as negative samples. In this context, the objective is to maximize the probability

$$\prod_{(w,c)\in D'} p(D'=0|w,c;\theta) = \prod_{(w,c)\in D'} (1-p(D'=1|w,c;\theta)) = \prod_{(w,c)\in D'} \left(1-\frac{1}{1+e^{-v_c\cdot v_w}}\right) = \prod_{(w,c)\in D'} \frac{1}{1+e^{v_c\cdot v_w}}$$

That is, our goal is to minimize the loss function

$$J(\theta) = J'(\theta) - \sum_{(w,c)\in D'} \log \frac{1}{1+e^{v_c\cdot v_w}} = \sum_{(w,c)\in D} \log\left(1+e^{-v_c\cdot v_w}\right) + \sum_{(w,c)\in D'} \log\left(1+e^{v_c\cdot v_w}\right) \tag{12}$$

5 Transformers

The transformer architecture employs the encoder-decoder structure. The encoder takes an input sequence of symbol representations $\mathbf{x} = (x_1, \cdots, x_n)$ and transforms it into a sequence of continuous representations $\mathbf{z} = (z_1, \cdots, z_n)$. Once we have the sequence $\mathbf{z}$, the decoder produces an output sequence $(y_1, \cdots, y_m)$ by generating one symbol at a time. At each step in the decoding process, the decoder incorporates the previously generated symbols as additional input when generating the next symbol. A visual representation of the transformer architecture can be seen in Figure 2, with Nx being 6, indicating that the block labeled Nx is repeated 6 times.

The transformer model, as described in Vaswani et al.'s paper [19], utilizes a word embedding size of $d_{\text{model}} = 512$. In this model, the input sentence $\mathbf{x}$ is initially broken down into a sequence of tokens (words) denoted as $\mathbf{x} = (x_1, \cdots, x_n)$. Each word is associated with a 512-dimensional real-number vector, and this representation is subject to updates during the learning process. As a result, the input sequence can be transformed into a sequence of real-number vectors: $u^{x_1}, \cdots, u^{x_n}$.

To incorporate positional information into the input vector sequence $u^{x_1}, \cdots, u^{x_n}$, we require a series of position vectors. Consider $pos_i = (p_{i,1} \cdots, p_{i,512})$, a 512-dimensional real-number vector defined as follows:

$$p_{i,j} = \begin{cases} \sin \frac{i}{10000^{\frac{j}{512}}} & \text{if } j \text{ is even} \\ \\ \cos \frac{i}{10000^{\frac{j-1}{512}}} & \text{if } j \text{ is odd} \end{cases}$$

The input vectors are augmented with position vectors by letting $\bar{u}^{x_1} = u^{x_1} + pos_1, \cdots, \bar{u}^{x_n} = u^{x_n} + pos_n$. It's important to note that these position vectors remain constant throughout the transformer model's operation and are not updated during the learning process.

The sequence of augmented input vectors, denoted as $\bar{u}^{x_1}, \cdots, \bar{u}^{x_n}$, can be converted into a matrix A with dimensions $n \times 512$. We can set $Q = A$, $K = A$, and $V = A$ as the query, key, and value matrices, respectively. Then the dot-product based self-attention can be computed as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k represents the dimension of the input vectors, which in this case is 512. It's worth mentioning that QK^T forms an $n \times n$ matrix which serves as a representation of the connections between these words. The matrix QK^T is divided by $\sqrt{512}$ and then the softmax is applied to each row. The $n \times 512$-dimensional attention matrix Attention(Q,K,V) is expected to encode information about each word and its relationships with all other words, with each row capturing such associations. To elaborate, the element at the (i, j) position in the matrix QK^T is determined by $\mathbf{q}_i \cdot \mathbf{k}_j = |\mathbf{q}_i||\mathbf{k}_j| \cos(\theta)$, where $\mathbf{q}_i$ represents the ith row of matrix Q, $\mathbf{k}_i$ is the jth row of matrix K, and θ is the angle between them. In essence, the element at the (i, j) position of the matrix QK^T is larger if and only if the ith word and the jth word are closer in meaning.

Instead of employing a single attention mechanism, the transformer model subdivides the 512-dimensional vector into $h = 8$ segments and employs $h = 8$ separate attention heads. This approach enables us to use individual heads to capture various facets of a word's meaning and its attention to other words simultaneously.

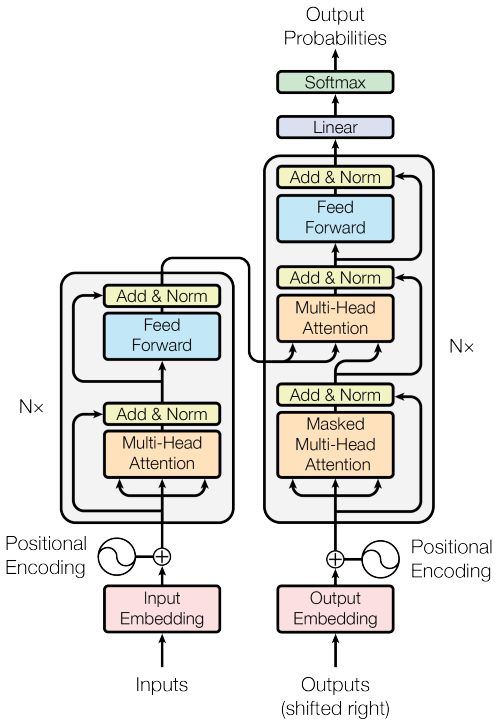

Figure 2: The Transformer - model architecture (from [19])

Now, let's define d_k as $d_{\text{model}}/h = 512/8 = 64$. For i ranging from 1 to 8, we have weight matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$, and $W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ for the query matrix Q, key matrix K, and value matrix V, and output matrix, respectively. These weight matrices are subject to learning during the training process. For each $i = 1, \cdots, 8$, we have

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V).$$

Then we can concatenate these heads to obtain the multi-head attention sub-layer output

$$\text{MultiHead}(Q, K, V) = [\text{head}_1, \cdots, \text{head}_h]\, W^O$$

After the application of the multi-head attention operation to the augmented input vectors $\bar{u}^{x_1}, \cdots, \bar{u}^{x_n}$, we proceed to the "Add & Norm" operation. In this "Add" operation which is also called residual connectoin, the output of the multi-head attention is summed with the input to the multi-head attention operation. In other words, we have $\mathbf{x} + f(\mathbf{x})$, where $f(\mathbf{x})$ represents the multi-head attention operation. This concept was originally introduced in the context of residual learning building blocks by He et al. [10]. For the normalization operation, as described by Ba et al. [1], we first compute two values: $\mu = \frac{\sum_{i=1}^{N} a_i}{N}$, which is the average of all values a_i in the layer to be normalized, and $\sigma = \sqrt{\frac{\sum_{i=1}^{N}(a_i - \mu)^2}{N}}$, which is the standard deviation calculated from these values. Then, we normalize each a_i as follows:

$$\bar{a}_i = \frac{a_i - \mu}{\sigma + \varepsilon}$$

where ε is a small added value so that $\bar{a}_i$ are not too large when ϵ is very small.

The outcome of the preceding procedure, referred to as the "Add & Norm" operation, serves as the input to a fully connected feed-forward network. This network is individually and consistently applied to every word position, corresponding to each row within the $n \times 512$ matrix. It involves two linear transformations separated by a ReLU activation function:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where W_1 is of dimension $d_{\text{model}} \times d_{ff}$ and W_2 is of dimension $d_{ff} \times d_{\text{model}}$. The paper [19] used a parameter of $d_{ff} = 2048$. The FFN's output is directed into the "Add & Norm" operation to derive a sub-layer output, concluding the operations within the Nx box for one iteration. To obtain the final output $\mathbf{z}$ for the Encoder, these operations in the Nx box must be repeated six times.

The output $\mathbf{z}$ from the Encoder serves as an input for the Decoder's "Multi-Head Attention". To elaborate, the sequence $\mathbf{z}$ is transformed into the matrices Q and K, which are subsequently provided as input to the Decoder's "Multi-Head Attention". The matrix V input for the Decoder's "Multi-Head Attention" is sourced from the "Masked Multi-Head Attention" and the "Add & Norm" operation within the Decoder module. The operations within the Decoder are quite similar to those within the Encoder, with one notable exception being the "Masked Multi-Head Attention". The Decoder enables each position, in other words, each row of the $m \times 512$ dimensional output matrix, to attend to all positions within the Decoder, up to and including that specific position. This constraint is crucial to prevent the flow of information in a leftward direction within the Decoder and thereby preserve its auto-regressive nature. To achieve this, Transformer implement a masking process in the scaled dot-product attention mechanism. During this process, we set all values in the input of the softmax corresponding to illegal connections to $-\infty$.

6 Reinforcement Learning from Human Feedback (RLHF)

In the Foundation Model paradigm [2], following pretraining of the large language model using the transformer, fine-tuning the model becomes necessary along with the integration of human preferences directly into the model. This process typically involves Reinforcement Learning from Human Feedback (RLHF).

6.1 Reinforcement learning and Markov decision processes (MDP)

Reinforcement learning is typically explained through a Markov decision process (MDP), which comprises a tuple $(S, A, \{P_{sa} : s \in S, a \in A\}, \gamma, R)$. Here, S is a set of states, A is a set of actions, $\{P_{sa} : s, a\}$ are the state transition probabilities, $\gamma \in [0, 1)$ is the discount factor, and $R : S \to \mathcal{R}$ is the reward function.

An MDP starts with a state s_0, where an agent selects an action $a_0 \in A$. Following the aaction a_0, the MDP's state transitions randomly to a successor state s_1 according to the probabilistic distribution P_{s_0,a_0}. Subsequently, the agent chooses another action $a_1 \in A$, leading to another random transition of the MDP's state to a successor state s_2, and so on. After a sequence of actions, the total reward that the agent obtains is

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

The goal in reinforcement learning is to choose actions over time so as to maximize the expected value of the total reward

$$E\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots\right]$$

A policy is a function $\pi : S \to A$ that maps the states to the actions. The value function for a policy π can be defined as

$$V^{\pi}(s) = E\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots | s_0 = s, \pi\right].$$

Given a fixed policy π, its value function V^{π} satisfies the Bellman equations which is essential for the value iteration and policy iteration learning process.

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s,\pi(s)}(s') V^{\pi}(s').$$

For an MDP, the optimal value function is defined as $V^*(s) = \max_{\pi} V^{\pi}(s)$. Then the optimal policy could be defined as

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{s,a}(s') V^*(s').$$

In the large language model, we are more intereted in the policy iteration learning algorithm which procedds as follows.

- initialize π
- repeat until convergence
  - Let $V = V^{\pi}$
  - for each state s, let

$$\pi(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{s,a}(s') V^*(s').$$

6.2 Fine-tuning language models using RLHF

In this section we use instructGPT as a case study to demonstrate the enhancement of a pre-trained large language model's performance. In reinforcement learning terms, we regard the language model as a policy π. Specifically, we assume that we have obtained a pretrained language model π_0 through a transformer-based deep learning procedure. Starting from π_0, Figure 3 outlines the three distinct stages involved in instructGPT training [16]:

- This initial language model (policy) π_0 is fine-tuned using additional texts generated by humans (labelers), resulting in a new model π_1.
- Human labelers are engaged to assess the outputs of model π_1 by ranking them. These rankings serve as training data for a reward model.
- The refinement of the model/policy π_1 is accomplished through reinforcement learning.

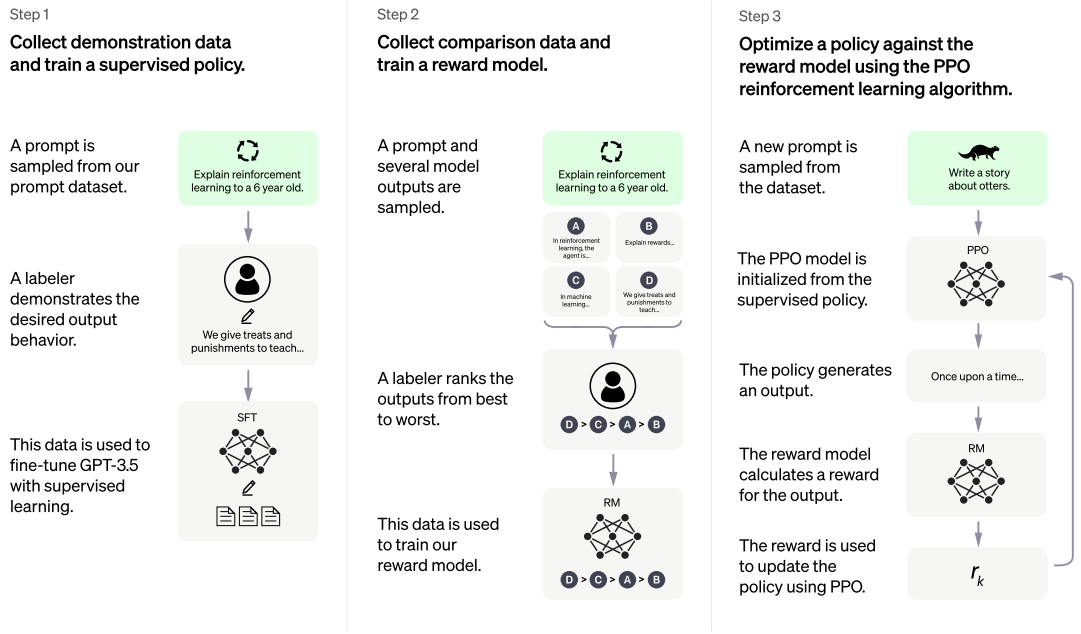

Figure 3: Three steps of instructGPT (from [16])

Step 1 follows a simple process, while Step 2 involves deriving the reward function for reinforcement learning. Our aim is to create a reward function that takes a text sequence and produces a numerical reward, reflecting human preference. We plan to construct this function using a transformer neural network. To generate training data for this reward function transformer, we sample prompts from a predefined dataset, passing them through the language model π_1 to generate new text. Human labelers then rank the text outputs produced by π_1.

In instructGPT, for each prompt input, the labelers are presented from $K = 4$ to $K = 8$ π_1-generated responses to rank. The loss function for reward function training could be defined differently for different models. For instructGPT, it is defined as

$$\text{loss}(\theta) = \frac{1}{\binom{K}{2}} E_{(x,y_w,y_l)\sim D} \left[\log\left(\sigma(r_\theta(x, y_w) - r_\theta(x, y_l))\right)\right]$$

where $r_\theta(x, y)$ is the scalar output of the reward model for prompt x with completion y with parameters θ, y_w is the preferred completion ouf of the pair of y_w and y_l, and D is the dataset of human comparisons. Alternatively, one may use cross entropy to define the loss function (see, e.g., [4]) which is based on the famous Elo ranking system [5].

After deriving the reward function in step 2, we refine model π_1 through reinforcement learning (RL) in step 3, yielding the fine-tuned model π_2. To prevent π_2 from producing gibberish merely to fulfill the reward function, we aim to minimize the Kullback-Leibler divergence between the outputs of π_1 and π_2. In particular, when considering the parameters θ, if the reward value from the output π_2 is r_θ and the Kullback-Leibler divergence between the outputs of π_1 and π_2 is D_{KL}, then the final reward $r_\theta - D_{KL}$ is used in reinforcement learning (RL) to update these parameters.

As a conclusion of this section, we briefly describe the Kullback-Leibler divergence. For two discrete probability distributions p and q on the same sample space X, the Kullback-Leibler divergence $D_{KL}(p||q)$ (also known as relative entropy or I-divergence), is defined as

$$D_{KL}(p||q) = \sum_{x\in X} p(x) \log\left(\frac{p(x)}{q(x)}\right).$$

That is, $D_{KL}(p||q)$ is the expected logarithmic difference between the probabilities p and q, where the expectation is taken using the probabilities p.

7 Parameters used in practice

Table 1 shows the Transformeter parameters used in LaMDA [18], GPT-2 [17], GPT-3 [3], and GPT-4 [15].

Table 1: Transformer parameters for LaMDA and GPTs

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_k	d_{ff}	context size	training data
LaMDA	137 Billion	64	8192	128	128	65536		1.56T words
GPT-2	1542 Million	48	1600	12		3072	1024	40GB
GPT-3	175 Billion	96	12288	96		49152	2048	570GB
GPT-4	170 Trillion	120	?	?			32k	10T words
GPT-4 Turbo	?	?	?	?			128k	?

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[5] Arpad E Elo and Sam Sloan. The rating of chessplayers: Past and present. *(No Title)*, 1978.

[6] John Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, pages 10–32, 1957.

[7] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[8] Google. word2vec. `https://code.google.com/archive/p/word2vec/`, 2013.

[9] Michael U Gutmann and Aapo Hyvärinen. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of machine learning research*, 13(2), 2012.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[13] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *International workshop on artificial intelligence and statistics*, pages 246–252. PMLR, 2005.

[14] Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*, 2017.

[15] OpenAI. Gpt-4 technical report, 2023.

[16] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[17] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[18] R Thoppilan, D De Freitas, J Hall, N Shazeer, A Kulshreshtha, HT Cheng, A Jin, T Bos, L Baker, Y Du, et al. Lamda: Language models for dialog applications. arxiv 2022. *arXiv preprint arXiv:2201.08239*.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.