

Transformers and Large Language Models for Natural Language Processing

Eyal Gur

August 7, 2024

Part I

Basics of Neural Networks

1 A General Supervised Learning Task

We aim at finding a *prediction function* f such that $f(\mathbf{x}; \mathbf{w}) \in \mathbb{R}^k$ is the true outcome of event $\mathbf{x} \in \mathbb{R}^d$, where \mathbf{w} are the parameters (weights) that define the function. Given a dataset $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ of events with their true outcomes (labels), we can approximate this function by finding \mathbf{w} that best approximate $f(\mathbf{x}_i; \mathbf{w}) \approx \mathbf{y}_i$ for any data point $i = 1, 2, \dots, N$.

Continuous Outcomes. Assume that the labels represent a continuous outcome, as in regression tasks. We can assume that $\mathbf{y}_i \sim \text{Normal}[f(\mathbf{x}_i; \mathbf{w}), \sigma \mathbf{I}_k]$ for all data points i and for some $\sigma > 0$.

We can find the parameters (expectations and standard deviation) using the maximum-likelihood estimation (MLE) approach. Finding the optimal σ by differentiating the log-likelihood function and plugging it back, we arrive after algebraic manipulations to the equivalent problem

$$\min_{\mathbf{w}} \left\{ \mathcal{L}(\mathbf{w}) \equiv \frac{1}{N} \sum_{i=1}^N L_i(\mathbf{w}) \equiv \frac{1}{N} \sum_{i=1}^N \|f(\mathbf{x}_i; \mathbf{w}) - \mathbf{y}_i\|^2 \right\},$$

which is the *mean squared error* (MSE) loss function (up to a scalar multiplication).

Discrete Outcomes. Assume that the labels represent a discrete outcome, as in classification tasks. Hence, there are k possible outcomes, each is a one-hot vector of size k . Then, the outcomes follow a categorical distribution and for any data point i we have $\mathbf{y}_i \sim \text{Cat}[\mathbf{p}(f(\mathbf{x}_i; \mathbf{w}))]$, for $\mathbf{p}: \mathbb{R}^k \rightarrow \mathbb{R}^k$ a function that maps k possible outcomes to k outcome probabilities, one for each of the k classes (these probabilities depend on the true outcome function f).

We can find the parameters $\mathbf{p}(f(\mathbf{x}_i; \mathbf{w}))$ (the probabilities) by applying MLE, which results with the equivalent problem

$$\min_{\mathbf{w}} \left\{ \mathcal{L}(\mathbf{w}) \equiv \frac{1}{N} \sum_{i=1}^N L_i(\mathbf{w}) \equiv -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^k (\mathbf{y}_i)_j \log \left(\mathbf{p}(f(\mathbf{x}_i; \mathbf{w}))_j \right) \right\},$$

which is the *cross-entropy* (CE) loss function (up to a scalar multiplication).

We can further assume that $\mathbf{p}(\mathbf{z}) = \text{softmax}(\mathbf{z})$ for optimization and statistical advantages, though any other function that converts a vector of k real numbers into a probability distribution of k possible outcomes can be taken.

As for the prediction function f , which appears in the minimization problems above, it can be modeled using neural network architectures. In this case, due to increasing size of the networks, the loss function $\mathcal{L}(\mathbf{w})$ can be minimized using the Stochastic Gradient (SG) method or its variants.

2 Training Feed-Forward Neural Networks

In a *feed-forward* NN with M layers, we denote by $\phi_m^k(\mathbf{x})$ the output of the m -th layer at the k -th iteration. Then, this output is forwarded as an input to the $m+1$ -th layer, hence $\phi_{m+1}^k(\mathbf{x}) \equiv f_{m+1}(\phi_m^k(\mathbf{x}); \mathbf{w}_{(m+1)}^k)$, where f_m is the function of the m -th layer and $\mathbf{w}_{(m)}^k$ are its weights at the k -th iteration. Notice that $\phi_M^k(\mathbf{x}) = f(\mathbf{x}; \mathbf{w})$ and $\phi_0^k(\mathbf{x}) = \mathbf{x}$.

2.1 Forward-pass and Back-propagation

In order to calculate the gradients $\nabla L_i(\mathbf{w})$, we notice that

$$\begin{aligned} \frac{\partial f}{\partial \mathbf{w}_{(M)}}(\mathbf{x}; \mathbf{w}^k) &= \frac{\partial f_M}{\partial \mathbf{w}_{(M)}}(\phi_{M-1}^k(\mathbf{x}), \mathbf{w}_{(M)}^k), \\ \frac{\partial f}{\partial \mathbf{w}_{(M-1)}}(\mathbf{x}; \mathbf{w}^k) &= \frac{\partial f}{\partial \mathbf{w}_{(M)}}(\mathbf{x}; \mathbf{w}^k) \cdot \frac{\partial f_{M-1}}{\partial \mathbf{w}_{(M-1)}}(\phi_{M-2}^k(\mathbf{x}), \mathbf{w}_{(M-1)}^k), \\ &\vdots \\ \frac{\partial f}{\partial \mathbf{w}_{(1)}} &= \frac{\partial f}{\partial \mathbf{w}_{(2)}}(\mathbf{x}; \mathbf{w}^k) \cdot \frac{\partial f_1}{\partial \mathbf{w}_{(1)}}(\phi_0^k(\mathbf{x}), \mathbf{w}_{(1)}^k), \end{aligned}$$

where the differentiations are taken element-wise. In general, it holds that

$$\frac{\partial f}{\partial \mathbf{w}_{(m)}}(\mathbf{x}; \mathbf{w}^k) = \prod_{i=m}^M \frac{\partial f_i}{\partial \mathbf{w}_{(i)}}(\phi_{i-1}^k(\mathbf{x}), \mathbf{w}_{(i)}^k)$$

and we notice that the gradients at m -th layer do not depend on the gradients of the previous $m-1, m-2, \dots, 1$ layers. However, the gradients of the m -th layer do depend on $\phi_{m-1}^k(\mathbf{x})$,

which by itself depends on the outputs of all previous $m - 1, m - 2, \dots, 1$ layers. Finally, we have that

$$\nabla L_i(\mathbf{w}^k) = (\nabla_1 L_i(\mathbf{w}^k), \dots, \nabla_M L_i(\mathbf{w}^k)).$$

where we denote

$$\nabla_m L_i(\mathbf{w}^k) \equiv \frac{\partial L_i}{\partial \mathbf{w}_{(m)}}(\mathbf{w}^k) \cdot \frac{\partial f}{\partial \mathbf{w}_{(m)}}(\mathbf{x}; \mathbf{w}^k).$$

This means that the gradient $\nabla L_i(\mathbf{w}^k)$ can be calculated efficiently as follows:

1. *Forward-pass*: at each layer m and for each data point \mathbf{x}_i , we evaluate $\phi_m^k(\mathbf{x}_i)$ and pass this information to the next $m + 1$ layer until we obtain $\phi_M^k(\mathbf{x}) = f(\mathbf{x}; \mathbf{w}^k)$.
2. *Back-propagation*: First, we calculate $\frac{\partial L_i}{\partial \mathbf{w}_{(m)}}(\mathbf{x}_i; \mathbf{w}^k)$ for all m and all data points. Then, we calculate $\frac{\partial f_m}{\partial \mathbf{w}_{(m)}}(\phi_{m-1}^k(\mathbf{x}_i); \mathbf{w}_{(m)}^k)$ at each layer m for all data points, and propagate this information to the previous $m - 1$ layer.

The functions $\partial f_m / \partial \mathbf{w}_{(m)}$ and $\partial L_i / \partial \mathbf{w}_{(m)}$ can be constructed only once, when the network is deployed. This is outlined in Algorithm 1.

Algorithm 1 Training of a Feed-Forward Neural Network with SG

```

1: Initialization: Set  $\mathbf{w}^0$  and  $k = 0$ .
2: for epoch do
3:   for mini-batch  $\mathcal{B}$  do
4:     for layer  $m = 1, 2, \dots, M$  do
5:       Calculate  $\phi_m^k(\mathbf{x}_i)$  for each  $i \in \mathcal{B}$  and then forward-pass.
6:     end for
7:     Calculate  $L_i(\mathbf{w}^k)$  and  $\frac{\partial L_i}{\partial \mathbf{w}_{(m)}}(\mathbf{w}^k)$  for each  $i \in \mathcal{B}$  and layer  $m$ .
8:     for layer  $m = M, M - 1, \dots, 1$  do
9:       Calculate  $\frac{\partial f_m}{\partial \mathbf{w}_{(m)}}(\phi_{m-1}^k(\mathbf{x}_i); \mathbf{w}_{(m)}^k)$  for each  $i \in \mathcal{B}$  and then back-propagate.
10:       $\mathbf{w}_{(m)}^{k+1} := \mathbf{w}_{(m)}^k - \eta^k \sum_{i \in \mathcal{B}} \nabla_m L_i(\mathbf{w}^k)$ .
11:    end for
12:     $k := k + 1$ .
13:  end for
14: end for
```

Remark 1. 1. The partition of the dataset into mini-batches can be either a single partition across all epochs, a random partition in each epoch, or any other partition.

2. The gradient ∇L_i for any i is of the size of \mathbf{w} . If we want to zero a portion of its elements we can use a dropout procedure.

2.2 Optimizers

Several methods exist to enhance the performance of the “vanilla” SG. This is usually done by considering momentum-based variants of SG, adaptive step-size schedules, or a combination of the two. Popular examples are as follows (where we omit layer indices for readability):

- Heavy-ball momentum:

$$\mathbf{w}^{k+1} := \mathbf{w}^k - \eta^k \sum_{i \in \mathcal{B}} \mathbf{m}_i^k \quad \text{where} \quad \mathbf{m}_i^k := \beta \mathbf{m}_i^{k-1} + (1 - \beta) \nabla L_i(\mathbf{w}^k).$$

- AdaGrad: for each coordinate l of \mathbf{w}^k

$$\mathbf{w}_l^{k+1} := \mathbf{w}_l^k - \eta^k \sum_{i \in \mathcal{B}} (\mathbf{m}_i^k)_l \quad \text{where} \quad (\mathbf{m}_i^k)_l = \frac{\nabla L_i(\mathbf{w}^k)_l}{\sqrt{\sum_{j=0}^k \nabla L_i(\mathbf{w}^j)_l^2}},$$

hence each coordinate has an adaptive step-size.

- RMSProp: for each coordinate l of \mathbf{w}^k

$$\mathbf{w}_l^{k+1} := \mathbf{w}_l^k - \eta^k \sum_{i \in \mathcal{B}} (\mathbf{m}_i^k)_l \quad \text{where} \quad (\mathbf{m}_i^k)_l = \frac{\nabla L_i(\mathbf{w}^k)_l}{\sqrt{\mathbf{v}_l^k}},$$

for $\mathbf{v}_l^k = \beta \mathbf{v}_l^{k-1} + (1 - \beta) \nabla L_i(\mathbf{w}^k)_l^2$, hence each coordinate has an adaptive step-size (where the step-size utilizes momentum).

- Adam: for each coordinate l of \mathbf{w}^k

$$\mathbf{w}_l^{k+1} := \mathbf{w}_l^k - \eta^k \sum_{i \in \mathcal{B}} (\mathbf{m}_i^k)_l \quad \text{where} \quad (\mathbf{m}_i^k)_l = \frac{\beta_1 (\mathbf{m}_i^{k-1})_l + (1 - \beta_1) \nabla L_i(\mathbf{w}^k)_l}{\sqrt{\mathbf{v}_l^k}},$$

for $\mathbf{v}_l^k = \beta_2 \mathbf{v}_l^{k-1} + (1 - \beta_2) \nabla L_i(\mathbf{w}^k)_l^2$, hence each coordinate has an adaptive step-size, and both the descent direction and the step-size utilizes momentum.

Part II

Large Language Models

The insight of large language modeling is that many practical tasks (including, but not limited to, NLP tasks) can be cast as word prediction, and that a powerful-enough language model can solve them with a high degree of accuracy. These tasks include *word prediction*, *sentiment analysis*, *question answering* and *text summarization*.

The standard architecture for building large language models is the *transformer*, which is made up of stacks of *transformer blocks*, each is a multi-layer network that maps sequences of input vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N \in \mathbb{R}^d$ to sequences of output vectors $\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_N \in \mathbb{R}^d$. The size of the sequence N , is the *context window*. These blocks are made by combining feed-forward networks, normalization layers, and *self-attention layers*, the key innovation of transformer architecture.

3 Self-Attention Process

The intuition of attention is the idea of comparing an item to a collection of other items to reveal their relevance in the current context. In the case of self-attention for language, comparisons of words (or tokens) to other words within a given sequence. The result of these comparisons is then used to compute an output sequence for the current input sequence.

Formally, for each current input token \mathbf{x}_i , the *proportional relevance* vector $\boldsymbol{\alpha}_i \in \mathbb{R}^i$ with respect to all previous input tokens \mathbf{x}_j from the sequence, $j \leq i$, is

$$\boldsymbol{\alpha}_i \equiv \text{SOFTMAX} \left(\begin{bmatrix} \mathbf{x}_i^T \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_i^T \mathbf{x}_i \end{bmatrix} \right).$$

The attention vector $\mathbf{a}_i \in \mathbb{R}^d$ for the token \mathbf{x}_i is the weighted sum of comparisons for inputs seen so far

$$\mathbf{a}_i \equiv \sum_{j=1}^i \boldsymbol{\alpha}_{ij} \mathbf{x}_j.$$

Notice that each token \mathbf{x}_i has three different roles in the attention process. As the current focus of attention (*query*), as a preceding input (*key*), and as an argument for computing the weighted sum (*value*). These roles motivate the introduction of variable weight matrices $\mathbf{Q} \in \mathbb{R}^{d_K \times d}$, $\mathbf{K} \in \mathbb{R}^{d_K \times d}$ and $\mathbf{V} \in \mathbb{R}^{d_V \times d}$, which yield the modified definitions

$$\boldsymbol{\alpha}_i \equiv \text{SOFTMAX} \left(\begin{bmatrix} \mathbf{x}_i^T \mathbf{Q}^T \mathbf{K} \mathbf{x}_1 / d_K \\ \vdots \\ \mathbf{x}_i^T \mathbf{Q}^T \mathbf{K} \mathbf{x}_i / d_K \end{bmatrix} \right) \in \mathbb{R}^i \quad \text{and} \quad \mathbf{a}_i \equiv \sum_{j=1}^i \boldsymbol{\alpha}_{ij} \mathbf{V} \mathbf{x}_j \in \mathbb{R}^{d_V}. \quad (1)$$

The division of $\mathbf{Q} \mathbf{x}_i$ and $\mathbf{K} \mathbf{x}_j$, $j \leq i$, by $\sqrt{d_K}$ in the definition of $\boldsymbol{\alpha}_i$ is done to prevent exponentiation of large values.

The self-attention process, as formulated mathematically in (1), is depicted in Figure 1.

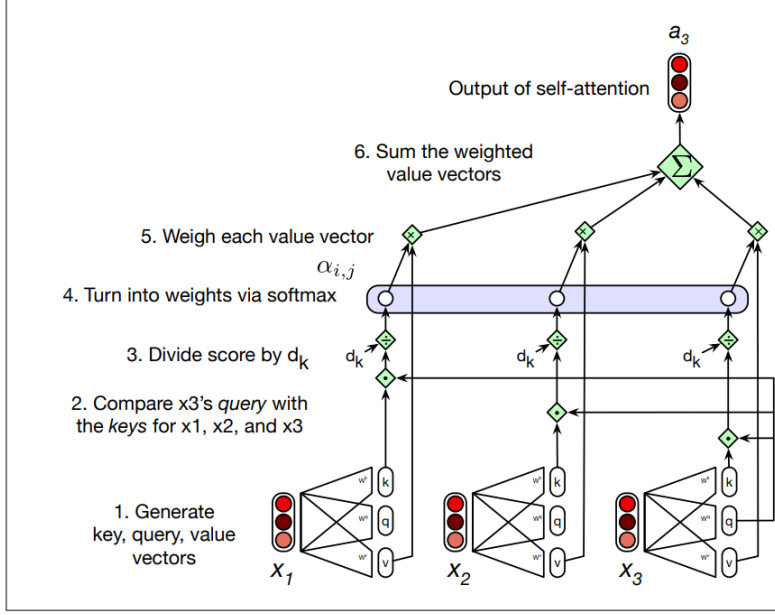


Figure 1: Calculating the self-attention vector $\mathbf{a}_3 \in \mathbb{R}^{d_v}$ using causal (left-to-right) self-attention process, in a sequence of $N = 3$ tokens.

Parallelization. A key observation is that the calculation of each attention vector is independent of other attention vectors, thus the entire attention process can be parallelized in matrix form. To this end, we denote by $\mathbf{X} \in \mathbb{R}^{N \times d}$ the matrix whose rows are all the N tokens in the sequence.

Notice that each attention vector only considers previous tokens in the sequence, and not future tokens. In order to mask future tokens, we define the future masking matrix operator, that keeps the upper-right triangular of a matrix and replaces all other entries with $-\infty$. For example, for a 4×4 matrix \mathbf{B} we get

$$\text{MASKFUTURE}(\mathbf{B}) \equiv \begin{pmatrix} \mathbf{b}_{11} & -\infty & -\infty & -\infty \\ \mathbf{b}_{21} & \mathbf{b}_{22} & -\infty & -\infty \\ \mathbf{b}_{31} & \mathbf{b}_{32} & \mathbf{b}_{33} & -\infty \\ \mathbf{b}_{41} & \mathbf{b}_{42} & \mathbf{b}_{43} & \mathbf{b}_{44} \end{pmatrix}.$$

Now we can write the self-attention process in matrix form as

$$\mathbf{A} \equiv \text{ROWSOFTMAX}(\text{MASKFUTURE}(\mathbf{X}\mathbf{Q}^T\mathbf{K}\mathbf{X}^T/d_K))\mathbf{X}\mathbf{V}^T \in \mathbb{R}^{N \times d_v},$$

where ROWSOFTMAX is the softmax operator performed on each row of a matrix. Notice that the N rows of the matrix \mathbf{A} are exactly the N attention vectors described above. We write the entire self-attention operation compactly as

$$\mathbf{A} = \text{SELFATTENTION}(\mathbf{Q}, \mathbf{K}, \mathbf{V}; \mathbf{X}).$$

Multi-Head Self-Attention. To allow more complicated attention relations between tokens in a sequence, the self-attention process is performed d_H times, all of which processes occur in parallel. Each such process is called *head*, and the entire process is called *multi-head self-attention*. Formally, we perform

$$\begin{aligned}\mathbf{H}_l &= \text{SELFATTENTION}(\mathbf{Q}_l, \mathbf{K}_l, \mathbf{V}_l; \mathbf{X}) \in \mathbb{R}^{N \times d_V}, \quad l = 1, 2, \dots, d_H, \\ \mathbf{A} &= \text{MULTIHEADS}(\mathbf{X}) = [\mathbf{H}_1; \mathbf{H}_2; \dots; \mathbf{H}_{d_H}] \mathbf{O} \in \mathbb{R}^{N \times d_O},\end{aligned}\quad (2)$$

where $\mathbf{O} \in \mathbb{R}^{d_H d_V \times d_O}$ is a variable matrix for transforming the N attention outputs into the dimension of our choice d_O (notice that the concatenated matrix of heads is of size $N \times d_H d_V$). Of course, we can take the inner dimensions d_K and d_V to be dependent on the head index $l = 1, 2, \dots, d_H$.

Again, the entire multi-head self-attention process is performed in parallel over the tokens, and the N output attention vectors reside in the rows of the matrix \mathbf{A} . The multi-head self-attention process, as formulated in (2), is depicted in Figure 2.

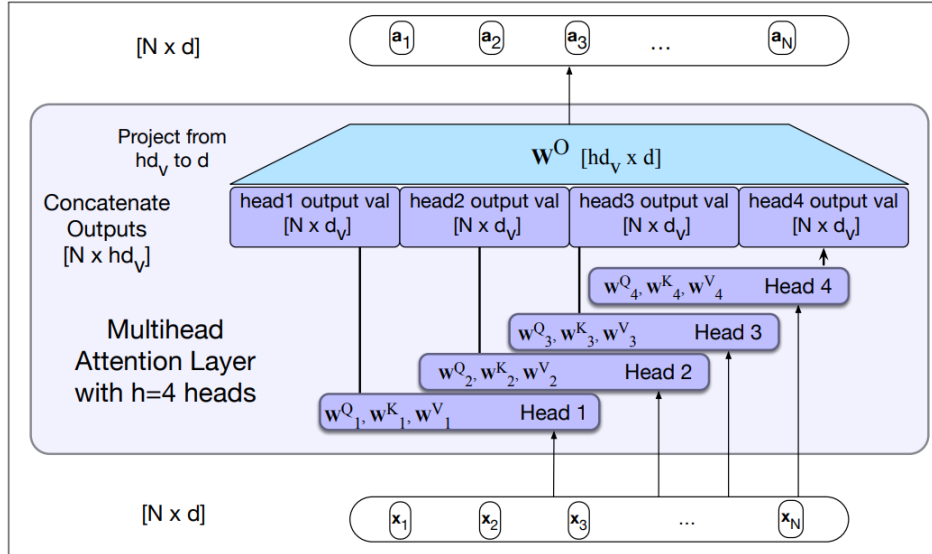


Figure 2: Calculating the self-attention matrix $\mathbf{A} \in \mathbb{R}^{N \times d_O}$ (the N vectors at the top represent the rows of this matrix), with a multi-head self-attention process with $d_H = 4$ heads. Notice that each head has its own query, key and value matrices, while the projection matrix $\mathbf{O} \in \mathbb{R}^{d_H d_V \times d_O}$ do not depend on the heads.

4 Transformer Block

The self-attention process lies at the core of the *transformer block*, which is composed of a multi-head self-attention, layer normalization, and N feed-forward networks, one for each attention vector. Also, we add residual connections that bypass the multi-head and the feed-forward layers. These blocks are also called *decoder-only*.

We assume that $d_K = d_V = d_O = d$ to enable easy stacking of blocks and for residual connectivity. LLMs with transformers stack many transformer blocks, from 12 layers (used

for the T5 or GPT-3-small language models) to 96 layers (used for GPT-3 large), to even more for more recent models.

Layer Normalization. The layer normalization operator takes an input vector $\mathbf{z} \in \mathbb{R}^d$ and returns the normalized vector $(\mathbf{z} - \mu) / \sigma$, where

$$\mu = \frac{1}{d} \sum_{j=1}^d \mathbf{z}_j \quad \text{and} \quad \sigma = \sqrt{\frac{1}{d} \sum_{j=1}^d (\mathbf{z}_j - \mu)^2}.$$

We point out that μ is a scalar and $\mathbf{z} - \mu$ is element-wise subtraction. The layer normalization operator is written as

$$\text{LN}(\mathbf{z}) = \gamma \left(\frac{\mathbf{z} - \mu}{\sigma} \right) + \beta,$$

for γ and β learnable scalar parameters. Notice that γ and β depend on the layer, and can be different for each phase of the layer normalization operator.

Feed-Forward Networks. The N feed-forward networks in the block can be taken to be, for example, fully-connected networks with a single hidden layer. That is, two weight matrices for each network. The input and output of each network is a d -dimensional vector. Usually, the dimension of the hidden layer is taken to be greater than d . The weight matrices can be taken to be, for example, the same for all N networks, but different across each block. Notice that the N networks can be computed in parallel.

The Complete Block. Thus, for an input matrix $\mathbf{Z} \in \mathbb{R}^{N \times d}$ – which can be either the token matrix \mathbf{X} or the output of the previous block – the *post-norm* transformer architecture can be written as

$$\begin{aligned} \tilde{\mathbf{A}} &= \text{ROWLN}(\text{MULTIHEADS}(\mathbf{Z}) + \mathbf{Z}) \in \mathbb{R}^{N \times d}, \\ \mathbf{A} &= \text{ROWLN}(\text{ROWFF}(\tilde{\mathbf{A}}) + \tilde{\mathbf{A}}) \in \mathbb{R}^{N \times d}, \end{aligned}$$

where ROWLN is the layer normalization operator applied to each row, and where ROWFF is a feed-forward network applied to each row of its input matrix, as described above. This can abbreviated as $\mathbf{A} = \text{TRANSFORMER}(\mathbf{Z})$.

The *pre-norm* transformer architecture, which is commonly used, can be written as

$$\begin{aligned} \tilde{\mathbf{A}} &= \text{MULTIHEADS}(\text{ROWLN}(\mathbf{Z})) + \mathbf{Z} \in \mathbb{R}^{N \times d}, \\ \mathbf{A} &= \text{ROWFF}(\text{ROWLN}(\tilde{\mathbf{A}})) + \tilde{\mathbf{A}} \in \mathbb{R}^{N \times d}. \end{aligned}$$

Again, the computations in a transformer block can be performed in parallel over the input vectors (either token vectors or output vectors from the previous block), and the N output attention vectors reside in the rows of \mathbf{A} .

A scheme depicting the relations between the layers in a single transformer block is given in Figure 3.

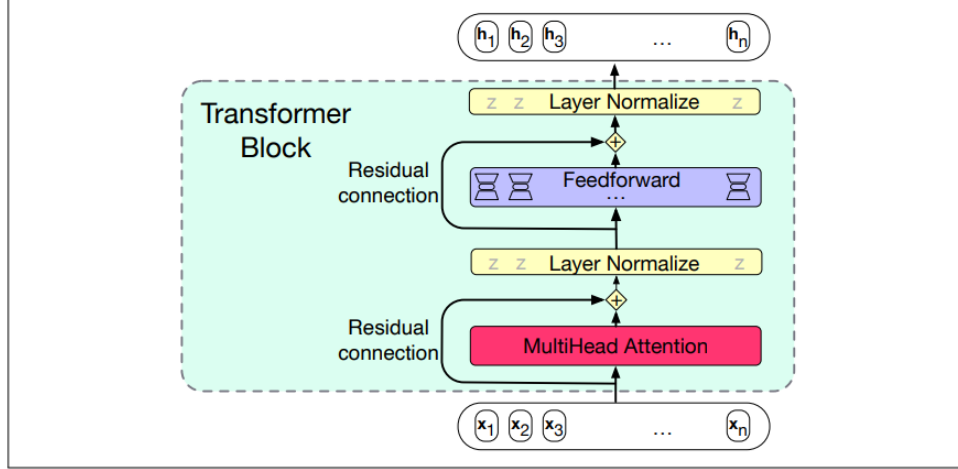


Figure 3: A post-norm transformer block architecture, taking input token embeddings $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N \in \mathbb{R}^d$ and returns attention vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_N \in \mathbb{R}^{d_v}$ (top row in the figure).

4.1 Token and Position Embeddings

The input matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ of tokens is obtained by taking the rows of the learnable *token embedding matrix* $\mathbf{E} \in \mathbb{R}^{|V| \times d}$. The i -th row of the matrix \mathbf{E} is an embedding of size d of the i -th token in the vocabulary V . Formally, let $\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \dots, \tilde{\mathbf{x}}_N \in \mathbb{R}^{|V|}$ be one-hot vectors corresponding to the N tokens taken from the vocabulary. Then,

$$\mathbf{X} \equiv [\tilde{\mathbf{x}}_1; \tilde{\mathbf{x}}_2; \dots; \tilde{\mathbf{x}}_N]^T \mathbf{E}.$$

The representation of each token as a vocabulary index (thus a row index in \mathbf{E}) is first created using byte pair encoding (BPE) algorithm, for example, described in a box below.

BPE algorithm. EXPLAIN

Notice that the vocabulary indices do not represent the position of each token in the given sequence. Hence, we should also consider *positional embeddings*, which are d -dimensional vector representations that encapsulate the relative positions of tokens within the target sequence. As such, it provides the transformer with information about where the tokens are in the input sequence.

A popular *relative* positional embedding function $\mathbf{p}: \mathbb{R} \rightarrow \mathbb{R}^d$ is the sinusoidal positional embedding, which takes a position t of a token and returns

$$(\mathbf{p}(t)_{2k}, \mathbf{p}(t)_{2k+1}) = (\sin(t \cdot C^{-2/d}), \cos(t \cdot C^{-2/d})), \quad \forall k = 0, 1, \dots, d/2 - 1,$$

where $C \gg d$ is a hyper-parameter.

A useful property of the sinusoidal embedding is that $\mathbf{p}(t + \Delta t) = \text{diag}(\mathbf{p}(\Delta t)) \mathbf{p}(t)$, hence allowing to calculate position shifts as a linear transformation of the current position. This allows the transformer to take any encoded position, and find the encoding of the position at $\pm \Delta t$ steps away by a matrix multiplication.

There are simpler *absolute* positional embeddings, which do not allow calculation of shifts by matrix multiplication.

Finally, considering both token and positional embeddings, the input matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ is taken as

$$\mathbf{X} \equiv [\tilde{\mathbf{x}}_1; \tilde{\mathbf{x}}_2; \dots; \tilde{\mathbf{x}}_N]^T \mathbf{E} + \mathbf{P},$$

where $\mathbf{P} \in \mathbb{R}^{N \times d}$ is the row-wise positional embedding matrix.

4.2 Language Modeling Head

Language models are token predictors, that is given a context of tokens, they assign a probability to each possible next token. The job of the language modeling head is to take the output of the final transformer from the last token N , and use it to predict the upcoming token at position $N + 1$.

To this end, we take the N -th (last) row of the last attention matrix \mathbf{A}^{last} , that is we take the vector $\mathbf{a} \equiv \mathbf{A}_N^{\text{last}} \in \mathbb{R}^d$, and unembed it to the vocabulary size by matrix multiplication. Using the *weight tying* approach, this learnable *unembedding matrix* can be taken as the embedding matrix $\mathbf{E} \in \mathbb{R}^{|V| \times d}$, but with multiplication from the right (in contrary to multiplication from the left as in the embedding process, which can also be viewed equivalently as matrix transposition). That is, the unembedding process is simply $\mathbf{u} = \mathbf{E}\mathbf{a} \in \mathbb{R}^{|V|}$, where the vector \mathbf{u} is called the vector of *logits*.

Finally, we compute the probabilities $\hat{\mathbf{y}} = \text{softmax}(\mathbf{u}) \in \mathbb{R}^{|V|}$, before plugging $\hat{\mathbf{y}}$ into CE loss. The vector $\hat{\mathbf{y}}$ contains the probabilities for each token in the vocabulary to be the next token in the sequence. The LLM head is depicted in Figure 4, while the entire decoder-only LLM architecture is given in Figure 5.

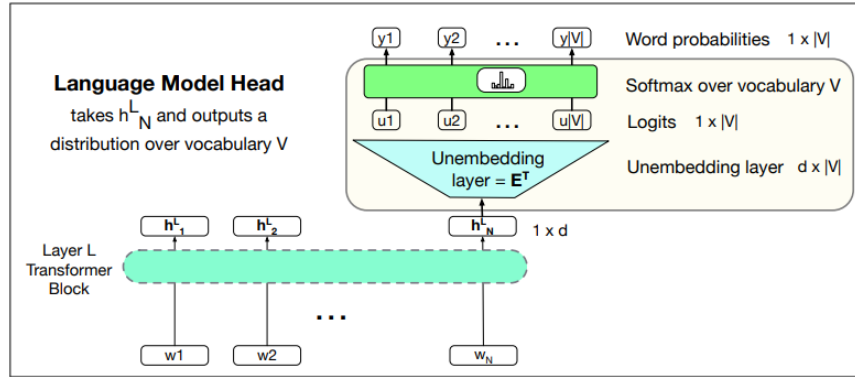


Figure 4: A language modeling head on top of the last attention vector $\mathbf{A}_N^{\text{last}} \in \mathbb{R}^d$ corresponding to the last N token in the sequence. This vector is unembedded using the weight tying approach to produce the vector of logits, which is transformed into token probabilities $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_{|V|} \in \mathbb{R}$ using softmax.

Loss Function. During training at time $t = 1, 2, \dots, N$, the next predicted $t + 1$ token is the one that minimizes the function

$$\mathcal{L}^t(\hat{\mathbf{y}}) = - \sum_{w \in V} \mathbf{y}_w^t \log(\hat{\mathbf{y}}_w^t), \quad (3)$$

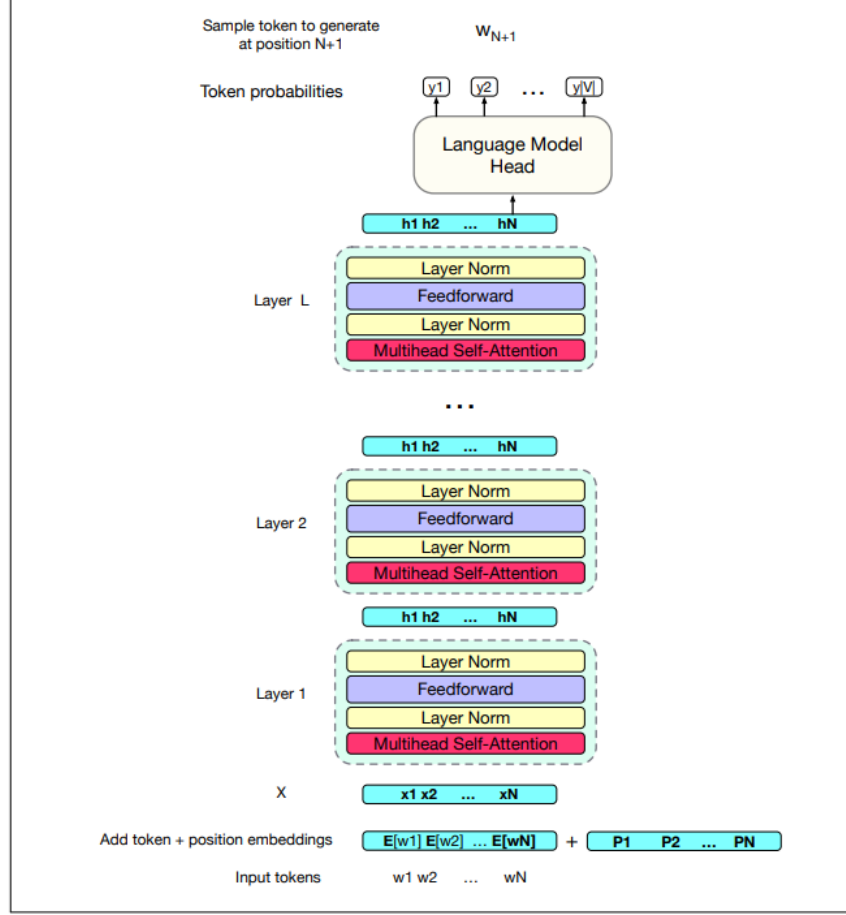


Figure 5: Decoder-only LLM architecture. The input tokens are transformed into the input matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ using token and position embeddings. Then, several transformer blocks are stacked to generate the attention vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_N \in \mathbb{R}^d$. Finally, the last \mathbf{a}_N vector is transferred to the language modeling head that generate the token probabilities $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_{|V|} \in \mathbb{R}$ that are used to sample the next $N + 1$ token.

where $\mathbf{y}^t \in \{0, 1\}^{|V|}$ is the true one-hot next $t + 1$ token. Since \mathbf{y}^t contains 1 for the true next w^{t+1} token and 0 otherwise, then (3) simplifies to

$$\mathcal{L}^t(\hat{\mathbf{y}}) = -\log(\hat{\mathbf{y}}_{w^{t+1}}^t).$$

The loss is then defined as

$$\mathcal{L}(\hat{\mathbf{y}}) = \frac{1}{N} \sum_{t=1}^N \mathcal{L}^t(\hat{\mathbf{y}}).$$

The batch size for gradient descent for minimizing this loss is usually quite large for LLMs (the largest GPT-3 model uses a batch size of 3.2 million tokens grouped into sequence contexts of size N).

5 LLM for Word Generation

Once the model is trained, the task of choosing a word to generate based on the probabilities the model provides is called *decoding*. Repeatedly choosing the next word conditioned on previous choices is called *autoregressive generation* or *causal LM generation*. Alternatives that predict words based on both past and future words are *non-causal*.

NLP tasks are cases of *conditional generation*, the task of generating text conditioned on an input piece of text – a prompt. Give the prompt, the LLM is asked to generate a possible completion. Note that as the generation process proceeds, the model has direct access to the primal prompt, as well as to all of its own subsequently generated outputs (which are concatenated to the prompt), at least as much as fits in the context window. For LLMs with transformers, the context window (for looking at previous tokens), is of size 1024 or even 4096 tokens, which makes them very powerful for conditional generation. Formally, the size of the context window N is what limits the size (number of rows) of the attention matrices as previously discussed.

As for the size of the vocabulary, the GPT-3 models, for example, are trained mostly on the web (429 billion tokens), some text from books (67 billion tokens) and Wikipedia (3 billion tokens).

Word Generation with Sampling. Once the model is trained, which words do we generate at each step? A simple way, called *greedy decoding*, is taking the most likely word given the context. A major problem with greedy decoding is that because the words it chooses are (by definition) extremely predictable, the resulting text is generic and quite repetitive (though a modification of this technique works well for machine translation).

Sampling is the most common approach for decoding. With a simple random sampling we sample words according to their probability conditioned on our previous choices, while the transformer LLM is the probability model that tells us this probability. Still, this sampling approach lacks quality and diversity.

Top- k sampling is a generalization of greedy decoding. We first truncate the distribution to the top k most likely words, normalize to produce a legitimate probability distribution, and then randomly sample from within these k words according to their normalized probabilities.

One problem with top- k sampling is that k is fixed, but the shape of the probability distribution over words differs in different contexts. An alternative approach is *top- p or nucleus sampling*. Here, the process is the same, but we choose the top- p percent of the words instead.

In *temperature sampling*, we reshape the the distribution instead of truncating. In low-temperature sampling, we smoothly increase the probability of the most probable words and decrease the probability of the rare words. Formally, this is simply done by dividing the vector of logits by a temperature hyper-parameter τ . That is, $\hat{\mathbf{y}} = \text{softmax}(\mathbf{u}/\tau)$.

Scaling Laws. The performance of LLMS is mainly determined by three factors: model size $\#H$ (the number of parameters not counting embeddings), dataset size $\#D$ (the amount of training data), and the amount of computer/iterations budget C used for training. The relationships between these factors and performance (value of loss), if the

other two properties are held constant, are known as *scaling laws*:

$$\mathcal{L}(\#H) = \left(\frac{\bar{H}}{\#H} \right)^{\alpha_H}, \quad \mathcal{L}(\#D) = \left(\frac{\bar{D}}{\#D} \right)^{\alpha_D} \quad \text{and} \quad \mathcal{L}(C) = \left(\frac{\bar{C}}{C} \right)^{\alpha_C},$$

where the values of \bar{H} , \bar{D} , \bar{C} , α_H , α_D and α_C depend on the exact transformer architecture, tokenization, and vocabulary size.

The number of (non-embedding) parameters $\#H$ for LLMs with transformers with one hidden layer in each FF network, can be roughly computed as follows

$$\#H \approx 2d \cdot \#\text{layers} \cdot (2d_V + d_{\text{FF}}),$$

where $\#\text{layers}$ is the number of transformer blocks and d_{FF} is the size of the hidden layer in the FF network. The specification of GPT-3, for example, are $\#\text{layers} = 96$ and $d = d_V = d_{\text{FF}}/4 = 12288$, thus it roughly has $\#H \approx 175$ billion parameters.

6 Bidirectional Transformer Encoders

Map sequences of input vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N \in \mathbb{R}^d$ to output vectors $\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_N \in \mathbb{R}^d$ that are contextualized using information from the entire input sequence. These blocks are also called *encoder-only*, because they produce an encoding for each input token.

The self-attention operator for encoder-only operates similarly to the decoder-only, with the exception that it does not mask the future. Hence, the operator is defined as

$$\mathbf{A} \equiv \text{RowSoftmax}(\mathbf{X}\mathbf{Q}^T\mathbf{K}\mathbf{X}^T/d_K)\mathbf{X}\mathbf{V}^T \in \mathbb{R}^{N \times d_V}.$$

For example, the original English-only encoder-only BERT consists of 30k tokens generated using WordPiece algorithm. It comprises of hidden layers of size 786, and 12 bidirectional transformer blocks with 12 multi-head attention layers each. The resulting model has about 100M parameters.

The larger multilingual XLM-RoBERTa, trained on 100 languages, has a multilingual 250k token vocabulary using SentencePiece Unigram LM algorithm. It comprises of hidden layers of size 1024, and 24 bidirectional transformer blocks with 16 multi-head attention layers each. The resulting model has about 550M parameters.

As with causal transformers, the size of the input layer dictates the complexity of the model. Both the time and memory requirements in a transformer grow quadratically with the length of the input N . It is necessary to set a fixed input length that is long enough to provide sufficient context for the model to function and yet still be computationally tractable. For BERT and XLM-RoBERTa, $N = 512$ was set.

Masking Words. For training, eliminating the masking process makes the guess-the-next-word task trivial since the answer is now directly available from the context. Instead, the model learns to fill-in-the-blank task, technically called the *cloze task*.

The original approach to training bidirectional encoders is called *Masked Language Modeling* (MLM), where a subset of $M \leq N$ tokens in the sequence is sampled, and each is either

replaced with [MASK], with another randomly sampled token from the vocabulary, or it is left unchanged. In BERT, 15% of the input tokens are sampled for learning, of which 80% are replaced with [MASK], 10% are replaced with random tokens, and the remaining 10% are left unchanged. Note that only M out of N tokens play a role in learning, so in that sense BERT is inefficient as less than 15% of the input is used for training.

Contextual Embeddings. We can think of the sequence of outputs as *contextual embeddings* for each token in the input. These embeddings are vectors representing some aspect of the meaning of a token in context, and can be used for any task requiring the meaning of tokens or words. It is common to compute a representation for each token by averaging the output tokens from each of the last four layers of the model. Thus, while thesauruses like WordNet give discrete lists of senses, embeddings offer a continuous high-dimensional model of meaning that, although it can be clustered, does not divide up into fully discrete senses.

During training, for each sense s of any word in the corpus (e.g., the token mouse as an animal or a cursor), for each of the n tokens of that sense, we average their n contextual representations \mathbf{v}_i to produce a contextual sense embedding \mathbf{v}_s . At test time, given a token t in the context, we compute its contextual embedding vector \mathbf{t} and choose its nearest neighbor sense from the training set, i.e., the sense whose embedding has the highest cosine with t :

$$\text{sense}(\mathbf{t}) = \underset{s \in \text{senses}(\mathbf{t})}{\operatorname{argmax}} \cos(\mathbf{t}, \mathbf{v}_s),$$

where the set $\text{senses}(\mathbf{t})$ is taken from some sense-labeled dataset like SemCore or SenseEval.

7 Enhancing LLMs

Is there room for enhancement with a trained LLM? Can we improve the predictions it generates? Two approaches exist, differing in whether we adjust a portion of the model’s weights further or not.

7.1 In-Context Learning

Prompting involves providing LLMs with a sequence of tokens during inference, aiming to predict the next tokens (for decoder-only) or generate contextual embeddings (for encoder-only). Prompts that do not contain additional labeled examples are termed *zero-shot* prompts. On the other hand, prompts that incorporate additional labeled data are termed *in-context learning* (ICL) or *few-shot* prompts.

In all prompting scenarios (including the ICL approach), it is important to highlight that the task is solely accomplished by conditioning on the given prompt, without any additional parameter optimization.

We recall that LLMs such as GPT-3 are trained on internet-scale text and as a result are very flexible in a sense that they can “read” any text input and condition on it to “write” text that could plausibly come after the input. While the training procedure is both simple and general, the GPT-3 paper found that the large-scale leads to the particularly interesting emergent behavior of ICL.

During ICL, we give the LLM a prompt that consists of a list of input-output pairs that demonstrate a task. At the end of the prompt, we append a test input and allow the LLM to make a prediction just by conditioning on the prompt and predicting the next tokens (the regular way). For example, in order to translate the word “chien” (dog) from French to English, we can simply use the zero-shot prompt “Chien in English is:”, which is a non-ICL prompt. With ICL, we give a few-shot prompt like “Maison in English is house. Chat in English is cat. Chien in English is:”. It turns out, that ICL gives better results compared to zero-shot prompts for a given LLM.

It is important to know that in contrast to training and fine-tuning for each specific task (see below), which are not temporary, what has been learnt during ICL is of a temporary nature. It does not carry the temporary contexts, except the ones already present in the training set, from one conversation to the other.

What accounts for the superior performance of ICL over zero-shot prompts is an active area of research, with various hypotheses proposed despite the lack of definitive conclusions.

7.2 Fine-Tuning

When solving a task, one can either use a general LLM without further weight updates using any type of prompting, or update a portion of the model weights in a process called *fine-tuning*. In this process, we create applications on top of the models by adding a small set of application-specific parameters. The fine-tuning process consists of using labeled data about the application to train these additional application-specific parameters. Typically, this training will either freeze or make only minimal adjustments to the language model parameters, a technique called *transfer learning*.

Let’s describe two examples of classification tasks typically accomplished through the fine-tuning approach of LLMs:

1. Sequence classification: such applications often represent an input sequence with a single representation. An additional *sentence embedding* vector is added to the model to stand for the entire sequence. In BERT, the [CLS] token plays the role of this embedding. This unique token is added to the vocabulary and is appended to the start of all input sequences. The output vector in the final layer of the model for [CLS] represents the entire input sequence and serves as the input to a classifier head – a logistic regression or neural network classifier that makes the relevant decision.

An example of fine-tuning is sentiment classification. We consider the output vector $\mathbf{z}_{[\text{CLS}]}$ for the [CLS] token, generated by a trained LLM. Then, the classification of the set into n possible sentiments proceeds by introducing a learnable weight matrix $\mathbf{C} \in \mathbb{R}^{n \times d_h}$, we calculate $\hat{\mathbf{y}} = \text{softmax}(\mathbf{C}\mathbf{z}_{[\text{CLS}]})$. Then, we only learn the matrix \mathbf{C} (fine-tune) using supervised training data consisting of input sequences labeled with the appropriate class with CE loss.

2. Sequence labeling: such tasks, for example part-of-speech tagging, can follow the same basic classification approach. Here, the final output vector corresponding to each input token is passed to a classifier that produces a softmax distribution over the possible set of tags. Again, assuming a simple classifier consisting of a single feed-forward layer

followed by a softmax, the set of weights to be learned for this additional layer is $\mathbf{K} \in \mathbb{R}^{k \times d_h}$, where k is the number of possible tags.

Instruct Tuning. An issue with LLMs is the mismatch between the training objective and users’ objective: LLMs are trained on minimizing the token prediction error, while users want the model to “follow their instructions”. To address this mismatch, *instruct tuning* (IT) is proposed. It is a fine-tuning process that uses the ICL approach that involves further training LLMs using (INSTRUCTION, OUTPUT) pairs, where INSTRUCTION denotes the human instruction for the model, and OUTPUT denotes the desired output that follows the INSTRUCTION. We point out that both INSTRUCTION and OUTPUT in the IT dataset are given by the user.

Based on the collected IT dataset, a model can be directly fine-tuned in a fully supervised manner, where given INSTRUCTION as the input, the model is trained by sequentially predicting each token in OUTPUT.

8 Encoder-Decoder Architecture

This architecture is mainly used for machine translation (MT). The encoder takes the source language input tokens $\mathbf{X} \in \mathbb{R}^{N \times d}$ and maps them to an output representation $\mathcal{E} \in \mathbb{R}^{N \times d}$, usually via six stacked self-attention transformer blocks (as previously introduced).

The decoder has a modified transformer block architecture, that attends both the the output of its previous layer (in the output language), and the output of the encoder (in the source language). Decoding into output translated tokens can use any of the decoding methods discussed earlier, like greedy, temperature or nucleus sampling (though the most common decoding algorithm for MT is the beam search algorithm discussed later).

In order to attend to the source language, the transformer blocks in the decoder have an extra *cross-attention* layer (also called *encoder-decoder attention* or *source attention*). This layer has the same form as the multi-head self-attention layer in a normal transformer block, except that while the queries come as usual from the previous layer of the decoder, the keys and values come from the output of the encoder $\mathcal{E} \in \mathbb{R}^{N \times d}$.

Formally, the cross-attention architecture for the decoder, for some block, is as follows

$$\mathbf{A}(\mathbf{Z}) \equiv \text{ROWSOFTMAX}(\text{MASKFUTURE}(\mathbf{Z}\mathbf{Q}^T\mathbf{K}\mathcal{E}^T/d_K))\mathcal{E}\mathbf{V}^T \in \mathbb{R}^{M \times d_V},$$

where M is the length of the translated sequence, $\mathbf{Z} \in \mathbb{R}^{M \times d}$ is the input from the last decoder layer, and the matrices \mathbf{Q} , \mathbf{K} and \mathbf{V} are of suitable dimensions. This can be written compactly as

$$\mathbf{A} = \text{CROSSATTENTION}(\mathbf{Q}, \mathbf{K}, \mathbf{V}; \mathbf{Z}).$$

Similarly to self-attention, in order to allow more complicated attention relations, we use multi-head cross-attention for the decoder as

$$\begin{aligned} \mathbf{H}_l &= \text{CROSSATTENTION}(\mathbf{Q}_l, \mathbf{K}_l, \mathbf{V}_l; \mathbf{Z}) \in \mathbb{R}^{M \times d_V}, \quad l = 1, 2, \dots, d_H, \\ \mathbf{A} &= \text{MULTIHEADC}(\mathbf{Z}) = [\mathbf{H}_1; \mathbf{H}_1; \dots; \mathbf{H}_{d_H}] \mathbf{O} \in \mathbb{R}^{M \times d_O} \end{aligned}$$

where the matrix \mathbf{O} is of suitable dimensions.

Finally, the decoder transformer block architecture is given as

$$\begin{aligned}\tilde{\mathbf{A}} &= \text{RowLN}(\text{MultiHeadS}(\mathbf{Z}) + \mathbf{Z}) \in \mathbb{R}^{M \times d}, \\ \tilde{\mathbf{A}} &= \text{RowLN}\left(\text{MultiHeadC}(\tilde{\mathbf{A}}) + \tilde{\mathbf{A}}\right) \in \mathbb{R}^{M \times d}, \\ \mathbf{A} &= \text{RowLN}\left(\text{RowFF}(\tilde{\mathbf{A}}) + \tilde{\mathbf{A}}\right) \in \mathbb{R}^{M \times d}.\end{aligned}$$

Notice that first we use the MULTIPLEHEADS operator, and then the MULTIPLEHEADC operator.

Beyond MT, the encoder-decoder architecture can also be used for training chatbots. In this case, the entire conversation up to the last turn (as much as fits in the context) is presented to the encoder, and the decoder generates the next turn.

Beam Search. In greedy decoding discussed above, at each time step t in generation, the output \hat{y}_t is chosen by computing the probability for each token in the vocabulary and then choosing the highest probability token. A problem with greedy decoding is that what the argmax might turn out to have been the wrong choice once we get to token $t+1$. Instead, we model decoding as searching the space of possible generations, represented as a *search tree* whose *branches* represent actions (generating a token), and *nodes* represent states (having generated a particular prefix). We search for the the highest probability sequence, contrary to just choosing the argmax at each time.

Computing the probability of every potential sequence is time-consuming. Instead, we utilize *beam search* algorithms, where we maintain k (referred to as the *beam width*) potential tokens at each stage. Initially, we calculate a softmax across the entire vocabulary to select the top k options, known as *hypotheses*. In subsequent stages, for each of the top k hypotheses, we compute the scores for all possible extensions so far, and once again select the best k hypotheses, ensuring a consistent count of k hypotheses at each stage.

The scoring of each hypotheses is done by computing the product of the log of the probability associated with the current token choice, multiplied by the product of the log of the probabilities of the path leading to this token.

Alternatives to this version of beam search also exists, like *minimum Bayes risk* decoding, that instead of choosing the translation which is most probable, it chooses the one that is likely to have the least error.

9 Evaluation of LLMs

LLMs are evaluated on different aspects depending on the specific task they are designed for. Main evaluation aspects typically assess:

1. **Correctness:** the factual correctness of the information provided by the model.
2. **Fluency and Coherence:** how well the text reads and whether it logically flows from one part to another.
3. **Hallucination:** instances where the model generates information that is not present in the input data and may be incorrect or fabricated.

4. Relevance: how well the generated content matches the given prompt or query.

These aspects are generic, and some task-specific aspects should also be considered. For example, for text summarization task we will need to assess whether the summary contains enough information from the original text.

9.1 Statistical Scorers

Before describing popular scorers, we mention that statistical methods performs poorly whenever reasoning is required, making it too inaccurate for most LLM evaluation criteria.

BLEU (BiLingual Evaluation Understudy). Measures the precision of n -grams (sequences of n symbols or words) in the generated text relative to reference texts. Precision here means the proportion of n -grams in the generated text that appear in the reference texts. Usually, $n = 1, 2, 3, 4$ (unigrams, bigrams, trigrams and 4-grams).

BLEU includes a brevity penalty to prevent generating overly short translations that might have high precision but fail to capture the full meaning. The brevity penalty is calculated as:

$$\text{BP} = \begin{cases} 1, & c > r, \\ \exp\left(1 - \frac{r}{c}\right), & c \leq r, \end{cases}$$

where c is the length of the generated text and r is the length of the reference text (or the closest reference length).

The BLEU score combines the n -gram precisions and the brevity penalty. The formula for the BLEU score is:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^N w_n \log(p_n)\right),$$

for p_n is the precision for n -grams and w_n are weights (usually $w_n = 1/N$).

For calculating the precisions p_n , we follow the next example. Assume reference texts ["there is a cat on the mat", "a cat is on the mat"] together with a candidate generated text "the the the the the the the". We assume that $p = 1$ and that unigrams are words (which is not optimal). For each word in the candidate translation, the algorithm takes its maximum total count m_{\max} in any of the reference translations. In this example, the word "the" appears twice in reference 1, and once in reference 2, thus $m_{\max} = 2$. For the candidate translation, the count of each word m_w is clipped to a maximum of m_{\max} for that word. In this case, "the" has $m_w = 7$ and $m_{\max} = 2$, thus m_w is clipped to 2. These clipped counts m_w are then summed over all distinct words in the candidate. This sum is then divided by the total number of unigrams in the candidate translation. In the above example, the unigram precision score would be $p_1 = 2/7$. Continuing in this fashion, we can calculate p_n for any $n \leq N$. Last, notice that $c = 7$ (length of candidate) and $r = 7$ (closest reference length) and hence $\text{BP} = 1$ (that is, we do not punish the generated text due to its length).

ROUGE (Recall-Oriented Understudy for Gisting Evaluation). A set of metrics and a software package used for evaluating automatic summarization and machine translation. The metrics compare a summary or translation against a reference or a set of human-produced references. ROUGE metrics range between 0 and 1, with higher scores indicating higher similarity between the automatically produced summary and the reference. Main metrics are available:

1. ROUGE- n : measures the overlap of n -grams between the generated summary and the reference summary.
2. ROUGE-L: measures the longest common sub-sequence between the generated summary and the reference summary. It takes into account sentence-level structure similarity and identifies the longest matching sequence of words.
3. ROUGE-W: a weighted version of ROUGE-L, giving higher scores to longer matches.
4. ROUGE-S: measures the overlap of skip-bigrams, which are pairs of words in their sentence order, allowing for arbitrary gaps.
5. ROUGE-SU: extends ROUGE-S to also includes unigram overlap in the evaluation.

METEOR (Metric for Evaluation of Translation with Explicit Ordering). A metric for the evaluation of machine translation, is based on the harmonic mean of unigram precision and recall, with recall weighted higher than precision. It also has several features that are not found in other metrics, such as stemming and synonymy matching, along with the standard exact word matching. The metric was designed to fix some of the problems found in BLEU, and also produce good correlation with human judgment at the sentence or segment level. This differs from the BLEU metric in that BLEU seeks correlation at the corpus level.

9.2 Model-based Scoring

Scoring that is purely statistical is reliable but inaccurate, as it struggles to take semantics into account. Creating a metric based on machine learning poses a fundamental challenge: the metric should do well consistently on a wide range of tasks and domains, and over time. However, there is only a limited amount of training data. Several model-based and LLM-based scorers exist.

BLEURT. Uses the BERT model to score LLM outputs on some expected outputs. First, the contextual word representations of BERT are taken. Second, a pre-training scheme is introduced: experiments reveal that training a model directly over publicly available human ratings is a risky approach, since we cannot control in what domain and across what time span the metric will be used. The accuracy is likely to drop in the presence of *domain drift* (when the text used comes from a different domain than the training sentence pairs) and it may also drop when there is a *quality drift* (when the ratings to be predicted are higher than those used during training).

To overcome this, BLEURT relies on “warming-up” the model using synthetic sentence pairs before fine-tuning on human ratings. The synthetic training data is generated by applying random perturbations to sentences from Wikipedia. Instead of collecting human ratings for these synthetic sentences, a collection of metrics (including BLEU and ROUGE) are calculated. The model is then fine-tuned on the WMT Metrics dataset, on a set of ratings provided by the user, or a combination of both.

G-Eval. A general evaluation scheme that uses LLMs to evaluate LLM outputs, and is one of the best ways to create task-specific metrics. G-Eval first generates a series of evaluation steps using chain of thoughts (CoTs) before using the generated steps to determine the final score. For example, evaluating LLM output coherence using G-Eval involves constructing a prompt that contains the criteria and text to be evaluated to generate evaluation steps, before using an LLM to output a score. The G-Eval scheme is as follows:

1. Introduce the required task to an LLM of choice. For example, “rate this summary from 1 to 5 based on coherence”.
2. Give a definition to your criteria. For example, “Coherence is the collective quality of all sentences in the actual output”.
3. Tell the LLM to generate a sequence of evaluation steps (CoTs) based on the two previous sections. For example, one can prompt “Evaluation steps:”.
4. Create a prompt concatenating the evaluation steps (CoTs), together with its necessary arguments. For example, for text summarization evaluation, concatenate the CoTs together with the original text and the out summary from a certain LLM.

G-Eval is great because as an LLM evaluator it is able to take full semantics of LLM outputs into account, making it much more accurate. Another major advantage of using an LLM evaluator is that LLMs are able to generate a reason for its evaluation score. However, although G-Eval correlates much more with human judgment when compared to its counterparts, it can still be unreliable, as asking an LLM to come up with a score is arbitrary.

Prometheus. A fully open-source LLM fine-tuned for evaluation that is comparable to GPT-4’s evaluation capabilities when the appropriate reference materials (reference answer and evaluation steps) are provided. It uses Llama-2-Chat as a base model and fine-tuned on 100K feedbacks generated by GPT-4. While G-Eval generates the evaluation steps via CoTs, the steps for Prometheus are provided in the prompt instead.

BERTScore. Relies on pre-trained models like BERT and computes the cosine similarity between the contextual embeddings of words in the reference and the generated texts. These similarities are then aggregated to produce a final score. A higher BERTScore indicates a greater degree of semantic overlap between the LLM output and the reference text.

SelfCheckGPT. A simple sampling-based approach that is used to fact-check LLM outputs. It assumes that hallucinated outputs are not reproducible, whereas if an LLM has knowledge of a given concept, sampled responses are likely to be similar and contain consistent facts. SelfCheckGPT makes detecting hallucination a reference-less process, which is useful in a production setting. However, SelfCheckGPT is only suitable for hallucination detection, and not for evaluating other use cases such as summarization, coherence, etc.

QAG (Question Answer Generation) Score. A scheme that leverages LLMs to reliably evaluate outputs of LLMs. It uses answers (usually either a ‘yes’ or ‘no’) to close-ended questions (which can be generated or preset) to compute a final metric score. It is reliable because it does not use LLMs to directly generate scores.

For example, if you want to compute a score for hallucination detection, you would first use an LLM to extract all claims made in an LLM output. Then, for each claim, ask the ground truth whether it agrees (‘yes’) or not (‘no’) with the claim made. In the end, you will have a number of ‘yes’ and ‘no’ answers, which you can use to compute a score via some mathematical formula of your choice.

LLM Evaluators for Text Summarization. Traditional methods for text summarization evaluation include ROUGE, BERTScore and others.

Word overlap metrics (like ROUGE) compare the overlap of words or phrases between the generated summary and a reference summary. If both summaries are of similar length (regardless of the quality of the summary), the likelihood of a higher overlap increases, potentially leading to higher scores.

Semantic similarity metrics (like BERTScore) evaluate the semantic similarity between the generated summary and a reference summary. Longer summaries might cover more content from the reference, which could result in a higher similarity score, even if the summary is not great in terms of quality.

LLM evaluators, although better than other evaluators, present challenges such as arbitrariness of the CoTs and scores, and can also be biased towards summaries that reflect the style and content present in their training data.

Research shows that the above limitations in LLM evaluators can hopefully be handled using the QAG general scheme discussed above. To this end, notice that developing LLM evaluators for the text summarization can be reduced to asking closed-ended questions to:

1. Coverage of details from the original text.
2. Factual alignment between the original text and summary.

Therefore, a text summarization task can be evaluated by calculating coverage and alignment scores, and combining the two (by taking their average, minimum, etc.) to yield a final summarization score.

Calculating coverage score: recall that coverage measures the amount of detail included in the summary from the original text, and that QAG requires a reference text to generate a set of close-ended questions. Hence, an algorithm for the coverage score is:

1. Generate n questions from the original text.
2. For each question, generate either a ‘yes’, ‘no’, or ‘idk’ answer using information from the original text and summary individually.

Since the questions are generated from the original text, answers from the original text are all ‘yes’, hence the higher the number of ‘yes’ in the summary, the greater the coverage score. A ‘no’ from the summary indicates a contradiction, whereas an ‘idk’ indicates omission of information from the summary.

Calculating alignment score: the algorithm is identical to coverage score, with the difference that we utilize the summary as the reference text to generate close-ended questions instead, and now all answers from the summary are ‘yes’. The ‘idk’ or ‘no’ answer from the original text indicates either a hallucination or contradiction.