Formal Algorithms for Transformers

Mary Phuong[1] **and Marcus Hutter**[1]
[1]DeepMind

This document aims to be a self-contained, mathematically precise overview of transformer architectures and algorithms (*not* results). It covers what transformers are, how they are trained, what they are used for, their key architectural components, and a preview of the most prominent models. The reader is assumed to be familiar with basic ML terminology and simpler neural network architectures such as MLPs.

Keywords: formal algorithms, pseudocode, transformers, attention, encoder, decoder, BERT, GPT, Gopher, tokenization, training, inference.

Contents

A famous colleague once sent an actually very well-written paper he was quite proud of to a famous complexity theorist. His answer: "I can't find a theorem in the paper. I have no idea what this paper is about."

1. Introduction

Transformers are deep feed-forward artificial neural networks with a (self)attention mechanism. They have been tremendously successful in natural language processing tasks and other domains. Since their inception 5 years ago [VSP+17], many variants have been suggested [LWLQ21]. Descriptions are usually graphical, verbal, partial, or incremental. Despite their popularity, it seems no pseudocode has ever been published for any variant. Contrast this to other fields of computer science, even to "cousin" discipline reinforcement learning [MKS+13, SBB18, EMK+21].

This report intends to rectify the situation for Transformers. It aims to be a self-contained, complete, precise and compact overview of transformer architectures and formal algorithms (but *not* results). It covers what Transformers are (Section 6), how they are trained (Section 7), what they're used for (Section 3), their key architectural components (Section 5), tokenization (Section 4), and a preview of practical considerations (Section 8) and the most prominent models.

The essentially complete pseudocode is about 50 lines, compared to thousands of lines of actual real source code. We believe these formal algorithms will be useful for theoreticians who require compact, complete, and precise formulations, experimental researchers interested in implementing a Transformer from scratch, and encourage authors to augment their paper or text book with formal Transformer algorithms (Section 2).

The reader is assumed to be familiar with basic ML terminology and simpler neural network architectures such as MLPs.

In short, a (formally inclined) reader, upon understanding the contents of this document, will have a solid grasp of transformers: they will be ready to read and contribute to the literature on the topic as well as implement their own Transformer using the pseudocode as templates.

2. Motivation

The true story above the introduction describes quite well the feeling we have when browsing

Corresponding author(s): {buiphuong,mhutter}@deepmind.com

many Deep Learning (DL) papers; unable to figure out the algorithmic suggestions exactly. For practitioners, the papers may be sufficiently detailed, but the precision needed by theoreticians is usually higher. For reasons not entirely clear to us, the DL community seems shy of providing pseudocode for their neural network models. Below we describe the SOTA in DL paper writing and argue for the value of formal algorithms. The reader already convinced about their merit can without loss skip this section.

The lack of scientific precision and detail in DL publications. Deep Learning has been tremendously successful in the last 5 to 10 years with thousands of papers published every year. Many describe only informally how they change a previous model, Some 100+ page papers contain only a few lines of prose informally describing the model [RBC+21]. At best there are some high-level diagrams. No pseudocode. No equations. No reference to a precise explanation of the model. One may argue that most DL models are minor variations of a few core architectures, such as the Transformer [VSP+17], so a reference augmented by a description of the changes should suffice. This would be true if (a) the changes were described precisely, (b) the reference architecture has been described precisely elsewhere, and (c) a reference is given to this description. Some if not all three are lacking in most DL papers. To the best of our knowledge no-one has even provided pseudocode for the famous Transformer and its encoder/decoder-only variations.

Interfacing algorithms. Equally important are proper explanations of how these networks are trained and used, but sometimes it is not even clear what the inputs and outputs and potential side-effects of the described model are. Of course someone experienced would usually be able to correctly guess, but this is not a particularly scientific approach. The experimental section in publications often does not explain what is actually fed into the algorithms and how. If there is some explanation in the methods section, it is often disconnected from what is described in the experimental section, possibly due to different authors writing the different sections. The core algorithms for the models should be accompanied by the wrapper algorithms that call them, e.g. (pre)training, fine-tuning, prompting, inference, deployment. Sometimes this is simple, but sometimes the magic happens there. In any case, if these things are not formally stated they remain unclear. Again, if the setup is standard and has been formally explained elsewhere, a simple reference will do.

Source code vs pseudocode. Providing open source code is very useful, but not a proper substitute for formal algorithms. There is a massive difference between a (partial) Python dump and well-crafted pseudocode. A lot of abstraction and clean-up is necessary: remove boiler plate code, use mostly single-letter variable names, replace code by math expressions wherever possible, e.g. replace loops by sums, remove (some) optimizations, etc. A well-crafted pseudocode is often less than a page and still essentially complete, compared to often thousands of lines of real source code. This is hard work no-one seems to be willing to do. Of course a forward process of first designing algorithms and write up pseudocode on paper, and then implementing it is fine too, but few DL practitioners seem to work that way.

Examples of good neural network pseudocode and mathematics and explanations. Multi-Layer Perceptrons (MLPs) are usually well-described in many papers, e.g. [MPCB14, BFT17, JGH18], though also without pseudocode. For a rare text-book example of pseudocode for a non-trivial neural network architecture, see Algorithm S2 of [SGBK+21], which constitutes a *complete*, i.e. *essentially executable*, pseudocode of just 25 lines based on a 350-line Python Colab toy implementation, which itself is based on a proper 1000+ line implementation.

This work aims to do the same for Transformers: The whole decoder-only Transformer GPT Algorithm 10 based on attention Algorithms 4 and 5 and normalization Algorithm 6 including training Algorithm 13 and prompting and inference Algorithm 14 all-together is less than 50 lines of pseudocode, compared to e.g. the 2000-

line self-contained C-implementation [Bel21].

[Ala19] is a great blog-post explaining Transformers and [EGKZ21] describes the attention mechanism to sufficient mathematical precision to allow proving properties about it, but neither provides pseudocode. [Elh21] is an attempt to understand Transformers by reverse-engineering the computations they perform and interpreting them as circuits.

Motivation. But does anyone actually need pseudocode and what would they be useful for (we sometimes get asked)? We find the absence of pseudocode in DL and this question quite perplexing, but apparently it requires answering. Providing such pseudocode can be useful for many purposes:

- They can be used as templates and adapted to precisely describe future variations, and therewith set a new standard in DL publishing. We explicitly encourage the reader to copy and adapt them to their needs and cite the original as "adapted from [PH22]".
- Having all that matters on one page in front of you makes it easier to develop new variations compared to reading prose or scrolling through 1000s of lines of actual code.
- They can be used as a basis for new implementations from scratch, e.g. in different programming languages, without having to wade through and reverse-engineer existing idiosyncratic real source code.
- They may establish notational convention, which eases communication and reading future variations.
- The process of converting source code into pseudocode can exhibit implementation errors (as it e.g. did in [SGBK+21]).
- Theoreticians need compact, complete, and precise representations for reasoning and ultimately proving properties about algorithms. They are often unwilling or unable to reverse engineer code, or guess the meaning of words or fancy diagrams.

With this motivation in mind, the following five sections formally describe all aspects of transformer architectures, training, and inference.

3. Transformers and Typical Tasks

Transformers are neural network models that excel at natural language processing, or more generally at modelling sequential data. Two common types of tasks they are used for are *sequence modelling* and *sequence-to-sequence prediction*.

Notation. Let V denote a finite set, called a *vocabulary*, often identified with $[N_\mathrm{V}] := \{1, ..., N_\mathrm{V}\}$. This could be words or letters, but typically are sub-words, called tokens. Let $\boldsymbol{x} \equiv x[1:\ell] \equiv x[1]x[2]...x[\ell] \in V^*$ be a sequence of tokens, e.g. a sentence or a paragraph or a document. Unlike in Python, we use arrays starting from 1, and $x[1:\ell]$ includes $x[\ell]$. For a matrix $M \in \mathbb{R}^{d\times d'}$, we write $M[i,:] \in \mathbb{R}^{d'}$ for the ith row and $M[:,j] \in \mathbb{R}^{d}$ for the j-th column. We use matrix $\times$ column vector convention more common in mathematics, compared to the default row vector $\times$ matrix in the transformer literature, i.e. our matrices are transposed. See Appendix B for a complete list of notation.

Chunking. The predominant paradigm in machine learning is (still) learning from independent and identically distributed (i.i.d.) data. Even for sequence modelling for practical reasons this tradition is upheld. The training data may naturally be a collection of (independent) articles, but even then, some may exceed the maximal context length ℓ_max transformers can handle. In this case, an article is crudely broken into shorter chunks of length $\leq \ell_\mathrm{max}$.

Sequence modelling (DTransformer). Given a vocabulary V, let $\boldsymbol{x}_n \in V^*$ for $n \in [N_\mathrm{data}]$ be a dataset of sequences (imagined to be) sampled i.i.d. from some distribution P over V^*. The goal is to learn an estimate $\hat{P}$ of the distribution $P(\boldsymbol{x})$. In practice, the distribution estimate is often decomposed via the chain rule as $\hat{P}(\boldsymbol{x}) = \hat{P}_{\boldsymbol{\theta}}(x[1]) \cdot \hat{P}_{\boldsymbol{\theta}}(x[2]\,|\,x[1]) \cdots \hat{P}_{\boldsymbol{\theta}}(x[\ell]\,|\,\boldsymbol{x}[1:\ell-1])$, where $\boldsymbol{\theta}$ consists of all neural network parameters to be learned. The goal is to learn a distribution over a single token $x[t]$ given its preceding tokens $x[1:t-1]$ as context.

Examples include e.g. language modelling, RL policy distillation, or music generation.

Sequence-to-sequence (seq2seq) prediction (EDTransformer). Given a vocabulary V and an i.i.d. dataset of sequence pairs $(\boldsymbol{z}_n, \boldsymbol{x}_n) \sim P$, where P is a distribution over $V^* \times V^*$, learn an estimate of the conditional distribution $P(\boldsymbol{x}|\boldsymbol{z})$. In practice, the conditional distribution estimate is often decomposed as $\hat{P}(\boldsymbol{x}|\boldsymbol{z}) = \hat{P}_{\boldsymbol{\theta}}(x[1] \,|\, \boldsymbol{z}) \cdot \hat{P}_{\boldsymbol{\theta}}(x[2] \,|\, x[1], \boldsymbol{z}) \cdots \hat{P}_{\boldsymbol{\theta}}(x[\ell] \,|\, \boldsymbol{x}[1:\ell-1], \boldsymbol{z})$.

Examples include translation ($\boldsymbol{z}$ = a sentence in English, $\boldsymbol{x}$ = the same sentence in German), question answering ($\boldsymbol{z}$ = question, $\boldsymbol{x}$ = the corresponding answer), text-to-speech ($\boldsymbol{z}$ = a piece of text, $\boldsymbol{x}$ = a voice recording of someone reading the text).

Classification (ETransformer). Given a vocabulary V and a set of classes $[N_{\text{C}}]$, let $(\boldsymbol{x}_n, c_n) \in V^* \times [N_{\text{C}}]$ for $n \in [N_{\text{data}}]$ be an i.i.d. dataset of sequence-class pairs sampled from $P(\boldsymbol{x}, c)$. The goal in classification is to learn an estimate of the conditional distribution $P(c|\boldsymbol{x})$.

Examples include e.g. sentiment classification, spam filtering, toxicity classification.

4. Tokenization: How Text is Represented

In the context of natural language tasks, *tokenization* refers to how a piece of text such as "My grandma makes the best apple pie." is represented as a sequence of vocabulary elements (called *tokens*).

Character-level tokenization. One possible choice is to let V be the English alphabet (plus punctuation). In the example above, we'd get a sequence of length 36: ['M', 'y', ' ', ...]. Character-level tokenization tends to yield very long sequences.

Word-level tokenization. Another choice would be to let V consist of all English words (plus punctuation). In the example above, we'd get a sequence of length 7: ['My ', 'grandma ', 'makes ', ...]. Word-level tokenization tends to require a very large vocabulary and cannot deal with new words at test time.

Subword tokenization. This is the method used in practice nowadays: V is a set of commonly occurring word segments like 'cious', 'ing', 'pre'. Common words like 'is ' are often a separate token, and single characters are also included in V to ensure all words are expressible.

There are in fact many ways to do subword tokenization. One of the simplest and most successful ones is Byte Pair Encoding [Gag94, SHB16] used in GPT-2 [RWC+19].

Final vocabulary and text representation. Given a choice of tokenization / vocabulary, each vocabulary element is assigned a unique index in $\{1, 2, \ldots, N_{\text{V}} - 3\}$. A number of special tokens are then added to the vocabulary. The number of special tokens varies, and here we will consider three: `mask_token` $:= N_{\text{V}} - 2$, used in masked language modelling (see Algorithm 12); `bos_token` $:= N_{\text{V}} - 1$, used for representing the beginning of sequence; and `eos_token` $:= N_{\text{V}}$, used for representing the end of sequence. The complete vocabulary has $N_{\text{V}} = |V|$ elements.

A piece of text is represented as a sequence of indices (called *token IDs*) corresponding to its (sub)words, preceded by `bos_token` and followed by `eos_token`.

5. Architectural Components

The following are the neural network building blocks (functions with learnable parameters) from which transformers are made. Full architectures featuring these building blocks are presented in the next section. (By a slight abuse of notation, we identify V with the set $\{1, 2, \ldots, N_{\text{V}}\}$.)

Token embedding. The token embedding learns to represent each vocabulary element as a vector in $\mathbb{R}^{d_{\text{e}}}$; see Algorithm 1.

Algorithm 1: Token embedding.

Input: $v \in V \cong [N_{\mathrm{V}}]$, a token ID.
Output: $\boldsymbol{e} \in \mathbb{R}^{d_{\mathrm{e}}}$, the vector representation of the token.
Parameters: $\boldsymbol{W_e} \in \mathbb{R}^{d_{\mathrm{e}} \times N_{\mathrm{V}}}$, the token embedding matrix.

return $\boldsymbol{e} = \boldsymbol{W_e}[:, v]$

Algorithm 2: Positional embedding.

Input: $\ell \in [\ell_{\max}]$, position of a token in the sequence.
Output: $\boldsymbol{e_p} \in \mathbb{R}^{d_{\mathrm{e}}}$, the vector representation of the position.
Parameters: $\boldsymbol{W_p} \in \mathbb{R}^{d_{\mathrm{e}} \times \ell_{\max}}$, the positional embedding matrix.

return $\boldsymbol{e_p} = \boldsymbol{W_p}[:, \ell]$

Positional embedding. The positional embedding learns to represent a token's position in a sequence as a vector in $\mathbb{R}^{d_{\mathrm{e}}}$. For example, the position of the first token in a sequence is represented by a (learned) vector $\boldsymbol{W_p}[:, 1]$, the position of the second token is represented by another (learned) vector $\boldsymbol{W_p}[:, 2]$, etc. The purpose of the positional embedding is to allow a Transformer to make sense of word ordering; in its absence the representation would be permutation invariant and the model would perceive sequences as "bags of words" instead.

Learned positional embeddings require that input sequence length is at most some fixed number $\ell_{\max}$ (the size of the learned positional embedding matrix must be finite and fixed in advance of training). An intuitive explanation of how this works can be found at [Ala18]. For pseudocode, see Algorithm 2.

Not all transformers make use of *learned* positional embeddings, some use a hard-coded mapping $\boldsymbol{W_p} : \mathbb{N} \to \mathbb{R}^{d_{\mathrm{e}}}$ instead [Ker21]. Such hard-coded positional embeddings can (theoretically) handle arbitrarily long sequences. The original Transformer [VSP+17] uses

$$\begin{aligned} \boldsymbol{W_p}[2i-1, t] &= \sin(t/\ell_{\max}^{2i/d_{\mathrm{e}}}), \\ \boldsymbol{W_p}[2i, t] &= \cos(t/\ell_{\max}^{2i/d_{\mathrm{e}}}). \end{aligned}$$

for $0 < i \leq d_{\mathrm{e}}/2$.

The positional embedding of a token is usually added to the token embedding to form a token's initial embedding. For the t-th token of a sequence $\boldsymbol{x}$, the embedding is

$$\boldsymbol{e} = \boldsymbol{W_e}[:, x[t]] + \boldsymbol{W_p}[:, t]. \tag{1}$$

Attention. Attention is the main architectural component of transformers. It enables a neural network to make use of contextual information (e.g. preceding text or the surrounding text) for predicting the current token.

On a high level, attention works as follows: the token currently being predicted is mapped to a *query* vector $\boldsymbol{q} \in \mathbb{R}^{d_{\mathrm{attn}}}$, and the tokens in the context are mapped to *key* vectors $\boldsymbol{k}_t \in \mathbb{R}^{d_{\mathrm{attn}}}$ and *value* vectors $\boldsymbol{v}_t \in \mathbb{R}^{d_{\mathrm{value}}}$. The inner products $\boldsymbol{q}^\intercal \boldsymbol{k}_t$ are interpreted as the degree to which token $t \in V$ is important for predicting the current token q – they are used to derive a distribution over the context tokens, which is then used to combine the value vectors. An intuitive explanation how this achieves attention can be found at [Ala18, Ala19]. The precise algorithm is given in Algorithm 3.

Algorithm 3: Basic single-query attention.

Input: $\boldsymbol{e} \in \mathbb{R}^{d_{\mathrm{in}}}$, vector representation of the current token
Input: $\boldsymbol{e}_t \in \mathbb{R}^{d_{\mathrm{in}}}$, vector representations of context tokens $t \in [T]$.
Output: $\tilde{\boldsymbol{v}} \in \mathbb{R}^{d_{\mathrm{out}}}$, vector representation of the token and context combined.
Parameters: $\boldsymbol{W_q}, \boldsymbol{W_k} \in \mathbb{R}^{d_{\mathrm{attn}} \times d_{\mathrm{in}}}$, $\boldsymbol{b_q}, \boldsymbol{b_k} \in \mathbb{R}^{d_{\mathrm{attn}}}$, the query and key linear projections.
Parameters: $\boldsymbol{W_v} \in \mathbb{R}^{d_{\mathrm{out}} \times d_{\mathrm{in}}}$, $\boldsymbol{b_v} \in \mathbb{R}^{d_{\mathrm{out}}}$, the value linear projection.

$\boldsymbol{q} \leftarrow \boldsymbol{W_q e} + \boldsymbol{b_q}$
$\forall t: \ \boldsymbol{k}_t \leftarrow \boldsymbol{W_k e}_t + \boldsymbol{b_k}$
$\forall t: \ \boldsymbol{v}_t \leftarrow \boldsymbol{W_v e}_t + \boldsymbol{b_v}$
$\forall t: \ \alpha_t = \frac{\exp(\boldsymbol{q}^\intercal \boldsymbol{k}_t / \sqrt{d_{\mathrm{attn}}})}{\sum_u \exp(\boldsymbol{q}^\intercal \boldsymbol{k}_u / \sqrt{d_{\mathrm{attn}}})}$
return $\tilde{\boldsymbol{v}} = \sum_{t=1}^{T} \alpha_t \boldsymbol{v}_t$

There are many ways the basic attention mechanism is used in transformers. We list some of the most common variants below.

It will be useful to define the softmax function for matrix arguments, as well as a Mask matrix:

$$\text{softmax}(\boldsymbol{A})[t_z, t_x] := \frac{\exp A[t_z, t_x]}{\sum_t \exp A[t, t_x]}, \quad (2)$$

$$\text{Mask}[t_z, t_x] = \begin{cases} 1 & \text{for bidirectional attention} \\ [\![t_z \le t_x]\!] & \text{for unidirectional att.} \end{cases} \quad (3)$$

Algorithm 4: $\tilde{\boldsymbol{V}} \leftarrow \texttt{Attention}(\boldsymbol{X}, \boldsymbol{Z} | \boldsymbol{\curlyvee W_{qkv}}, \text{Mask})$

```
/* Computes a single (masked) self- or
   cross- attention head.                */
```

Input: $\boldsymbol{X} \in \mathbb{R}^{d_x \times \ell_x}, \boldsymbol{Z} \in \mathbb{R}^{d_z \times \ell_z}$, vector representations of primary and context sequence.

Output: $\tilde{\boldsymbol{V}} \in \mathbb{R}^{d_{\text{out}} \times \ell_x}$, updated representations of tokens in $\boldsymbol{X}$, folding in information from tokens in $\boldsymbol{Z}$.

Parameters: $\boldsymbol{\curlyvee W_{qkv}}$ consisting of:
$\boldsymbol{W_q} \in \mathbb{R}^{d_{\text{attn}} \times d_x}$, $\boldsymbol{b_q} \in \mathbb{R}^{d_{\text{attn}}}$
$\boldsymbol{W_k} \in \mathbb{R}^{d_{\text{attn}} \times d_z}$, $\boldsymbol{b_k} \in \mathbb{R}^{d_{\text{attn}}}$
$\boldsymbol{W_v} \in \mathbb{R}^{d_{\text{out}} \times d_z}$, $\boldsymbol{b_v} \in \mathbb{R}^{d_{\text{out}}}$.

Hyperparameters: $\text{Mask} \in \{0,1\}^{\ell_z \times \ell_x}$, $\uparrow$(3)

$\boldsymbol{Q} \leftarrow \boldsymbol{W_q X} + \boldsymbol{b_q} \boldsymbol{1}^\top$ $\quad [\![\boldsymbol{Q}\text{uery} \in \mathbb{R}^{d_{\text{attn}} \times \ell_x}]\!]$
$\boldsymbol{K} \leftarrow \boldsymbol{W_k Z} + \boldsymbol{b_k} \boldsymbol{1}^\top$ $\quad [\![\boldsymbol{K}\text{ey} \in \mathbb{R}^{d_{\text{attn}} \times \ell_z}]\!]$
$\boldsymbol{V} \leftarrow \boldsymbol{W_v Z} + \boldsymbol{b_v} \boldsymbol{1}^\top$ $\quad [\![\boldsymbol{V}\text{alue} \in \mathbb{R}^{d_{\text{out}} \times \ell_z}]\!]$
$\boldsymbol{S} \leftarrow \boldsymbol{K}^\top \boldsymbol{Q}$ $\quad [\![\boldsymbol{S}\text{core} \in \mathbb{R}^{\ell_z \times \ell_x}]\!]$
$\forall t_z, t_x$, if $\neg \text{Mask}[t_z, t_x]$ then $S[t_z, t_x] \leftarrow -\infty$
return $\tilde{\boldsymbol{V}} = \boldsymbol{V} \cdot \text{softmax}\left(\boldsymbol{S}/\sqrt{d_{\text{attn}}}\right)$

Bidirectional / unmasked self-attention. Given a sequence of token representations, this variant applies attention to each token, treating all tokens in the sequence as the context. See Algorithm 4, called with token sequence $\boldsymbol{Z} = \boldsymbol{X}$ and no masking (Mask$\equiv$1).

Unidirectional / masked self-attention. Given a sequence of token representations, this variant applies attention to each token, treating all preceding tokens (including itself) as the context. Future tokens are masked out, so this causal auto-regressive version can be used for online prediction. See Algorithm 4, called with token sequence $\boldsymbol{Z} = \boldsymbol{X}$ and $\text{Mask}[t_z, t_x] := [\![t_z \le t_x]\!]$. For this Mask, the output $\tilde{\boldsymbol{V}}[:, 1:t]$ only depends on $\boldsymbol{X}[:, 1:t]$, hence can be used to predict $\boldsymbol{X}[:, t+1]$.

Cross-attention. Given two sequences of token representations (often in the context of a sequence-to-sequence task), this variant applies attention to each token of the primary token sequence $\boldsymbol{X}$, treating the second token sequence $\boldsymbol{Z}$ as the context. See Algorithm 4, called with Mask$\equiv$1. While the output $\tilde{\boldsymbol{V}}$ and input sequences $\boldsymbol{X}$ have the same length ℓ_x, the context sequence $\boldsymbol{Z}$ can have different length ℓ_z.

Multi-head attention. The attention algorithm presented so far (Algorithm 4) describes the operation of a single *attention head*. In practice, transformers run multiple attention heads (with separate learnable parameters) in parallel and combine their outputs; this is called *multi-head attention*; see Algorithm 5

Algorithm 5: $\tilde{\boldsymbol{V}} \leftarrow \texttt{MHAttention}(\boldsymbol{X}, \boldsymbol{Z} | \boldsymbol{\curlyvee W}, \text{Mask})$

```
/* Computes Multi-Head (masked) self-
   or cross- attention layer.          */
```

Input: $\boldsymbol{X} \in \mathbb{R}^{d_x \times \ell_x}, \boldsymbol{Z} \in \mathbb{R}^{d_z \times \ell_z}$, vector representations of primary and context sequence.

Output: $\tilde{\boldsymbol{V}} \in \mathbb{R}^{d_{\text{out}} \times \ell_x}$, updated representations of tokens in $\boldsymbol{X}$, folding in information from tokens in $\boldsymbol{Z}$.

Hyperparameters: H, number of attention heads

Hyperparameters: $\text{Mask} \in \{0,1\}^{\ell_z \times \ell_x}$, $\uparrow$(3)

Parameters: $\boldsymbol{\curlyvee W}$ consisting of
For $h \in [H]$, $\boldsymbol{\curlyvee W^h_{qkv}}$ consisting of:
$\quad \boldsymbol{W^h_q} \in \mathbb{R}^{d_{\text{attn}} \times d_x}$, $\boldsymbol{b^h_q} \in \mathbb{R}^{d_{\text{attn}}}$,
$\quad \boldsymbol{W^h_k} \in \mathbb{R}^{d_{\text{attn}} \times d_z}$, $\boldsymbol{b^h_k} \in \mathbb{R}^{d_{\text{attn}}}$,
$\quad \boldsymbol{W^h_v} \in \mathbb{R}^{d_{\text{mid}} \times d_z}$, $\boldsymbol{b^h_v} \in \mathbb{R}^{d_{\text{mid}}}$.
$\boldsymbol{W_o} \in \mathbb{R}^{d_{\text{out}} \times H d_{\text{mid}}}$, $\boldsymbol{b_o} \in \mathbb{R}^{d_{\text{out}}}$.

For $h \in [H]$:
$\quad \boldsymbol{Y}^h \leftarrow \texttt{Attention}(\boldsymbol{X}, \boldsymbol{Z} | \boldsymbol{\curlyvee W^h_{qkv}}, \text{Mask})$
$\boldsymbol{Y} \leftarrow [\boldsymbol{Y}^1; \boldsymbol{Y}^2; \ldots; \boldsymbol{Y}^H]$
return $\tilde{\boldsymbol{V}} = \boldsymbol{W_o Y} + \boldsymbol{b_o} \boldsymbol{1}^\top$

Algorithm 6: $\hat{e} \leftarrow$ `layer_norm`$(e|\gamma,\beta)$

`/* Normalizes layer activations` e. `*/`

Input: $e \in \mathbb{R}^{d_e}$, neural network activations.

Output: $\widehat{e} \in \mathbb{R}^{d_e}$, normalized activations.

Parameters: $\gamma, \beta \in \mathbb{R}^{d_e}$, element-wise scale and offset.

$m \leftarrow \sum_{i=1}^{d_e} e[i]/d_e$

$v \leftarrow \sum_{i=1}^{d_e} (e[i]-m)^2/d_e$

return $\widehat{e} = \frac{e-m}{\sqrt{v}} \odot \gamma + \beta$, where $\odot$ denotes element-wise multiplication.

Algorithm 7: Unembedding.

Input: $e \in \mathbb{R}^{d_e}$, a token encoding.

Output: $p \in \Delta(V)$, a probability distribution over the vocabulary.

Parameters: $W_u \in \mathbb{R}^{N_V \times d_e}$, the unembedding matrix.

return $p = \text{softmax}(W_u e)$

Layer normalisation. Layer normalisation explicitly controls the mean and variance of individual neural network activations; the pseudocode is given in Algorithm 6. Some transformers use a simpler and more computationally efficient version of layer normalization setting $m = \beta = 0$, called root mean square layer normalisation, or RMSnorm.

Unembedding. The unembedding learns to convert a vector representation of a token and its context into a distribution over the vocabulary elements; see Algorithm 7. The algorithm describes an independently learned unembedding matrix, but note that sometimes the unembedding matrix is instead fixed to be the transpose of the embedding matrix.

6. Transformer Architectures

This section presents a few prominent transformer architectures, based on attention Algorithms 4 and 5 and using normalization Algorithm 6, in roughly historical order:

- EDT [VSP+17] The original sequence-to-sequence / Encoder-Decoder Transformer, Algorithms 8, 11 and 15.
- BERT [DCLT19], which is an instance of an encoder-only transformer (encoder-only means that it is derived from the encoder-decoder architecture by dropping the decoder part), Algorithms 9 and 12.
- GPT [RWC+19, BMR+20], which is an instance of a decoder-only transformer, Algorithms 10, 13 and 14.

While the main architectural difference between BERT and GPT is in attention masking, they also differ in a number of less important ways: e.g. they use different activation functions and the layer-norms are positioned differently. We included these differences in the pseudocode to stay faithful to the original algorithms, but note that different transformer architectures may adopt these selectively.

To simplify notation, we denote by $\mathcal{W}$ the entire set of parameters (query, key, value and output linear projections) required by a multi-head attention layer:

$$\mathcal{W} := \begin{pmatrix} W_q^h \in \mathbb{R}^{d_{\text{attn}} \times d_x}, & b_q^h \in \mathbb{R}^{d_{\text{attn}}}, & h \in [H] \\ W_k^h \in \mathbb{R}^{d_{\text{attn}} \times d_z}, & b_k^h \in \mathbb{R}^{d_{\text{attn}}}, & h \in [H] \\ W_v^h \in \mathbb{R}^{d_{\text{mid}} \times d_z}, & b_v^h \in \mathbb{R}^{d_{\text{mid}}}, & h \in [H] \\ W_o \in \mathbb{R}^{d_{\text{out}} \times H d_{\text{mid}}}, & b_o \in \mathbb{R}^{d_{\text{out}}} & \end{pmatrix} \tag{4}$$

Encoder-decoder / sequence-to-sequence transformer [VSP+17]. This is the very first transformer. It was originally used for sequence-to-sequence tasks (machine translation), which is why it is more complicated than its successors.

The idea behind the architecture is as follows: First, the context sequence is encoded using bidirectional multi-head attention. The output of this 'encoder' part of the network is a vector representation of each context token, taking into account the entire context sequence. Second, the primary sequence is encoded. Each token in the primary sequence is allowed to use information from the encoded context sequence, as well as primary sequence tokens that precede it. See Algorithm 8 for more details.

Encoder-only transformer: BERT [DCLT19]. BERT is a bidirectional transformer trained on the task of masked language modelling. Given a piece of text with some tokens masked out, the goal is to correctly recover the masked-out tokens. The original use of BERT was to learn generally useful text representations, which could then be adapted for various downstream NLP tasks. The masking is not performed via the Mask parameter but differently: During training each input token is replaced with probability p_{mask} by a dummy token `mask_token`, and evaluation is based on the reconstruction probability of these knocked-out tokens (see Algorithm 12).

The BERT architecture resembles the encoder part of the seq2seq transformer (hence 'encoder-only'). It is described in detail in Algorithm 9. It uses the GELU nonlinearity instead of ReLU:

$$\text{GELU}(x) = x \cdot \mathbb{P}_{X\sim\mathcal{N}(0,1)}[X < x]. \tag{5}$$

(When called with vector or matrix arguments, GELU is applied element-wise.)

Decoder-only transformers: GPT-2 [RWC+19], GPT-3 [BMR+20], Gopher [RBC+21]. GPT-2 and GPT-3 are large language models developed by OpenAI, and Gopher is a large language model developed by DeepMind. They all have similar architectures and are trained by autoregressive language modelling: Given an incomplete sentence or paragraph, the goal is to predict the next token.

The main difference from BERT is that GPT/Gopher use unidirectional attention instead of bidirectional attention; they also apply layernorms in slightly different order.

See Algorithm 10 for the pseudocode of GPT-2. GPT-3 is identical except larger, and replaces dense attention in Line 6 by sparse attention, i.e. each token only uses a subset of the full context.

Gopher also deviates only slightly from the GPT-2 architecture: it replaces layer norm in lines 5, 7 and 10 by RMSnorm ($m = \beta = 0$), and it uses different positional embeddings.

Multi-domain decoder-only transformer: Gato [RZP+22]. Gato is a multi-modal multi-task transformer built by DeepMind. It is a single neural network that can play Atari, navigate 3D environments, control a robotic arm, caption images, have conversations, and more.

Under the hood, each modality is converted into a sequence prediction problem by a separate tokenization and embedding method; for example images are divided into non-overlapping 16×16 patches, ordered in raster order (left-to-right, top-to-bottom) and processed by a ResNet block to obtain a vector representation.

The actual Gato architecture is then a decoder-only transformer like the one in Algorithm 10, but where Line 2 is replaced with modality-specific embedding code.

7. Transformer Training and Inference

This section lists the pseudocode for various algorithms for training and using transformers:

- **EDTraining()** Algorithm 11 shows how to train a sequence-to-sequence transformer (the original Transformer [VSP+17]).
- **ETraining()** Algorithm 12 shows how to train a transformer on the task of masked language modelling (like BERT [DCLT19]).
- **DTraining()** Algorithm 13 shows how to train a transformer on the task of next token prediction (like CPT-x [BMR+20] and Gopher [RBC+21]).
- **DInference()** Algorithm 14 shows how to prompt a transformer trained on next token prediction (like GPT-x [BMR+20]). The temperature parameter τ interpolates between most likely continuation ($\tau = 0$), faithful sampling ($\tau = 1$), and uniform sampling ($\tau = \infty$).
- **EDInference()** Algorithm 15 shows how to use a sequence-to-sequence transformer for prediction.

Gradient descent. The described training Algorithms 11 to 13 use Stochastic Gradient Descent

(SGD) $\theta \leftarrow \theta - \eta \cdot \nabla \text{loss}(\theta)$ to minimize the log loss (aka cross entropy) as the update rule. Computation of the gradient is done via automatic differentiation tools; see [BPRS18, Table 5]. In practice, vanilla SGD is usually replaced by some more refined variation such as RMSProp or AdaGrad or others [Rud16]. Adam [KB15] is used most often these days.

8. Practical Considerations

While the vanilla transformers provided here may work in practice, a variety of "tricks" have been developed over the years to improve the performance of deep neural networks in general and transformers in particular [LWLQ21]:

- **Data preprocessing:** cleaning, augmentation [FGW+21], adding noise, shuffling [Lem21] (besides tokenization and chunking).
- **Architecture:** sparse layers, weight sharing (besides attention).
- **Training:** improved optimizers, minibatches, batch normalization, learning rate scheduling, weight initialization, pretraining, ensembling, multi-task, adversarial (besides layer normalization) [Sut15].
- **Regularization:** weight decay, early stopping, cross-validation, dropout, adding noise [MBM20, TZ22].
- **Inference:** scratchpad prompting, few-shot prompting, chain of thought, majority voting [LAD+22].
- **Others.**

A. References

[Ala18] Jay Alammar. The Illustrated Transformer. http://jalammar.github.io/illustrated-transformer/, 2018.

[Ala19] Jay Alammar. The Illustrated GPT-2 (Visualizing Transformer Language Models). http://jalammar.github.io/illustrated-gpt2/, 2019.

[Bel21] Fabrice Bellard. NNCP v2: Lossless Data Compression with Transformer. https://bellard.org/libnc/gpt2tc.html, 2021.

[BFT17] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. *NeurIPS*, 2017.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. *NeurIPS*, 2020.

[BPRS18] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: A Survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *ACL*, 2019.

[EGKZ21] Benjamin L. Edelman, Surbhi Goel, Sham Kakade, and Cyril Zhang. Inductive Biases and Variable Creation in Self-Attention Mechanisms. *arXiv:2110.10090 [cs, stat]*, October 2021.

[Elh21] Nelson Elhage. A Mathematical Framework for Transformer Circuits. https://transformer-circuits.pub/2021/framework/index.html, 2021.

[EMK+21] Yonathan Efroni, Dipendra Misra, Akshay Krishnamurthy, Alekh Agarwal, and John Langford. Provable RL with Exogenous Distractors via Multistep Inverse Dynamics, March 2021.

[FGW+21] Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988, 2021.

[Gag94] Philip Gage. A new algorithm for data compression. *Dr. Dobbs / C Users Journal*, 12(2):23–38, 1994.

[JGH18] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *NeurIPS*, 2018.

[KB15] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

[Ker21] Jonathan Kernes. Master Positional Encoding: Part I. https://towardsdatascience.com/master-positional-encoding-part-i-63c05d90a0c3, March 2021.

[LAD+22] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving Quantitative Reasoning Problems with Language Models. *arXiv:2206.14858 [cs]*, June 2022.

[Lem21] Chris Lemke. Data preprocessing in NLP. https://towardsdatascience.com/data-preprocessing-in-nlp-c371d53ba3e0, July 2021.

[LWLQ21] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A Survey of Transformers, June 2021.

[MBM20] Reza Moradi, Reza Berangi, and Behrouz Minaei. A survey of regularization strategies for deep models. *Artificial Intelligence Review*, 53(6):3947–3986, August 2020.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning, December 2013.

[MPCB14] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *NeurIPS*, 2014.

[PH22] M. Phuong and M. Hutter. Formal algorithms for transformers. Technical report, DeepMind, London, UK, 2022. LaTeX source available at http://arXiv.org

[RBC+21] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv:2112.11446*, 2021.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/, January 2016.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[RZP+22] Scott Reed, Konrad Żołna, Emilio Parisotto, et al. A generalist agent. *arXiv:2205.06175*, 2022.

[SBB18] Richard S. Sutton, Andrew G. Barto, and Francis Bach. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts, second edition edition edition, November 2018.

[SGBK+21] Eren Sezener, Agnieszka Grabska-Barwińska, Dimitar Kostadinov, Maxime Beau, Sanjukta Krishnagopal, David Budden, Marcus Hutter, Joel Veness, Matthew Botvinick, Claudia Clopath, Michael Häusser, and Peter E. Latham. A rapid and efficient learning rule for biological neural circuits. Technical report, DeepMind, London, UK, 2021.

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725. Association for Computational Linguistics (ACL), 2016.

[Sut15] Ilya Sutskever. A Brief Overview of Deep Learning. http://yyue.blogspot.com/2015/01/a-brief-overview-of-deep-learning.html, January 2015.

[TZ22] Yingjie Tian and Yuqi Zhang. A comprehensive survey on regularization strategies in machine learning. *Information Fusion*, 80:146–166, April 2022.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NeurIPS*, 2017.

Algorithm 8: $\boldsymbol{P} \leftarrow \texttt{EDTransformer}(\boldsymbol{z}, \boldsymbol{x} | \boldsymbol{\theta})$

/* Encoder-decoder transformer forward pass */

Input: $\boldsymbol{z}, \boldsymbol{x} \in V^*$, two sequences of token IDs.

Output: $\boldsymbol{P} \in (0,1)^{N_\text{V} \times \text{length}(\boldsymbol{x})}$, where the t-th column of $\boldsymbol{P}$ represents $\hat{P}_{\boldsymbol{\theta}}(x[t+1] | \boldsymbol{x}[1:t], \boldsymbol{z})$.

Hyperparameters: $\ell_\text{max}, L_\text{enc}, L_\text{dec}, H, d_\text{e}, d_\text{mlp} \in \mathbb{N}$

Parameters: $\boldsymbol{\theta}$ includes all of the following parameters:

$\boldsymbol{W_e} \in \mathbb{R}^{d_\text{e} \times N_\text{V}}$, $\boldsymbol{W_p} \in \mathbb{R}^{d_\text{e} \times \ell_\text{max}}$, the token and positional embedding matrices.

For $l \in [L_\text{enc}]$:

| $\boldsymbol{\curlyvee W}_l^\text{enc}$, multi-head encoder attention parameters for layer l, see (4),

| $\boldsymbol{\gamma}_l^1, \boldsymbol{\beta}_l^1, \boldsymbol{\gamma}_l^2, \boldsymbol{\beta}_l^2 \in \mathbb{R}^{d_\text{e}}$, two sets of layer-norm parameters,

| $\boldsymbol{W}_{\text{mlp1}}^l \in \mathbb{R}^{d_\text{mlp} \times d_\text{e}}$, $\boldsymbol{b}_{\text{mlp1}}^l \in \mathbb{R}^{d_\text{mlp}}$, $\boldsymbol{W}_{\text{mlp2}}^l \in \mathbb{R}^{d_\text{e} \times d_\text{mlp}}$, $\boldsymbol{b}_{\text{mlp2}}^l \in \mathbb{R}^{d_\text{e}}$, MLP parameters.

For $l \in [L_\text{dec}]$:

| $\boldsymbol{\curlyvee W}_l^\text{dec}$, multi-head decoder attention parameters for layer l, see (4),

| $\boldsymbol{\curlyvee W}_l^\text{e/d}$, multi-head cross-attention parameters for layer l, see (4),

| $\boldsymbol{\gamma}_l^3, \boldsymbol{\beta}_l^3, \boldsymbol{\gamma}_l^4, \boldsymbol{\beta}_l^4, \boldsymbol{\gamma}_l^5, \boldsymbol{\beta}_l^5 \in \mathbb{R}^{d_\text{e}}$, three sets of layer-norm parameters,

| $\boldsymbol{W}_{\text{mlp3}}^l \in \mathbb{R}^{d_\text{mlp} \times d_\text{e}}$, $\boldsymbol{b}_{\text{mlp3}}^l \in \mathbb{R}^{d_\text{mlp}}$, $\boldsymbol{W}_{\text{mlp4}}^l \in \mathbb{R}^{d_\text{e} \times d_\text{mlp}}$, $\boldsymbol{b}_{\text{mlp4}}^l \in \mathbb{R}^{d_\text{e}}$, MLP parameters.

$\boldsymbol{W_u} \in \mathbb{R}^{N_\text{V} \times d_\text{e}}$, the unembedding matrix.

/* Encode the context sequence: */

$\ell_\text{z} \leftarrow \text{length}(\boldsymbol{z})$

for $t \in [\ell_\text{z}]$: $\boldsymbol{e}_t \leftarrow \boldsymbol{W_e}[:, z[t]] + \boldsymbol{W_p}[:, t]$

$\boldsymbol{Z} \leftarrow [\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots \boldsymbol{e}_{\ell_\text{z}}]$

for $l = 1, 2, \ldots, L_\text{enc}$ **do**

  $\boldsymbol{Z} \leftarrow \boldsymbol{Z} + \texttt{MHAttention}(\boldsymbol{Z} | \boldsymbol{\curlyvee W}_l^\text{enc}, \text{Mask} \equiv 1)$

  for $t \in [\ell_\text{z}]$: $\boldsymbol{Z}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{Z}[:, t] | \boldsymbol{\gamma}_l^1, \boldsymbol{\beta}_l^1)$

  $\boldsymbol{Z} \leftarrow \boldsymbol{Z} + \boldsymbol{W}_{\text{mlp2}}^l \text{ReLU}(\boldsymbol{W}_{\text{mlp1}}^l \boldsymbol{Z} + \boldsymbol{b}_{\text{mlp1}}^l \boldsymbol{1}^\intercal) + \boldsymbol{b}_{\text{mlp2}}^l \boldsymbol{1}^\intercal$

  for $t \in [\ell_\text{z}]$: $\boldsymbol{Z}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{Z}[:, t] | \boldsymbol{\gamma}_l^2, \boldsymbol{\beta}_l^2)$

end

/* Decode the primary sequence, conditioning on the context: */

$\ell_\text{x} \leftarrow \text{length}(\boldsymbol{x})$

for $t \in [\ell_\text{x}]$: $\boldsymbol{e}_t \leftarrow \boldsymbol{W_e}[:, x[t]] + \boldsymbol{W_p}[:, t]$

$\boldsymbol{X} \leftarrow [\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots \boldsymbol{e}_{\ell_\text{x}}]$

for $i = 1, 2, \ldots, L_\text{dec}$ **do**

  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \texttt{MHAttention}(\boldsymbol{X} | \boldsymbol{\curlyvee W}_l^\text{dec}, \text{Mask}[t, t'] \equiv [[t \leq t']])$

  for $t \in [\ell_\text{x}]$: $\boldsymbol{X}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t] | \boldsymbol{\gamma}_l^3, \boldsymbol{\beta}_l^3)$

  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \texttt{MHAttention}(\boldsymbol{X}, \boldsymbol{Z} | \boldsymbol{\curlyvee W}_l^\text{e/d}, \text{Mask} \equiv 1)$

  for $t \in [\ell_\text{x}]$: $\boldsymbol{X}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t] | \boldsymbol{\gamma}_l^4, \boldsymbol{\beta}_l^4)$

  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \boldsymbol{W}_{\text{mlp4}}^l \text{ReLU}(\boldsymbol{W}_{\text{mlp3}}^l \boldsymbol{X} + \boldsymbol{b}_{\text{mlp3}}^l \boldsymbol{1}^\intercal) + \boldsymbol{b}_{\text{mlp4}}^l \boldsymbol{1}^\intercal$

  for $t \in [\ell_\text{x}]$: $\boldsymbol{X}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t] | \boldsymbol{\gamma}_l^5, \boldsymbol{\beta}_l^5)$

end

/* Derive conditional probabilities and return: */

return $\boldsymbol{P} = \text{softmax}(\boldsymbol{W_u X})$

Algorithm 9: $\boldsymbol{P} \leftarrow \texttt{ETransformer}(\boldsymbol{x}|\boldsymbol{\theta})$

```
/* BERT, an encoder-only transformer, forward pass */
```

Input: $\boldsymbol{x} \in V^*$, a sequence of token IDs.

Output: $\boldsymbol{P} \in (0,1)^{N_\mathrm{V} \times \mathrm{length}(\boldsymbol{x})}$, where each column of $\boldsymbol{P}$ is a distribution over the vocabulary.

Hyperparameters: $\ell_{\max}, L, H, d_\mathrm{e}, d_\mathrm{mlp}, d_\mathrm{f} \in \mathbb{N}$

Parameters: $\boldsymbol{\theta}$ includes all of the following parameters:

$\boldsymbol{W_e} \in \mathbb{R}^{d_\mathrm{e} \times N_\mathrm{V}}$, $\boldsymbol{W_p} \in \mathbb{R}^{d_\mathrm{e} \times \ell_{\max}}$, the token and positional embedding matrices.

For $l \in [L]$:

| $\boldsymbol{\curlyvee W}_l$, multi-head attention parameters for layer l, see (4),

| $\boldsymbol{\gamma}_l^1, \boldsymbol{\beta}_l^1, \boldsymbol{\gamma}_l^2, \boldsymbol{\beta}_l^2 \in \mathbb{R}^{d_\mathrm{e}}$, two sets of layer-norm parameters,

| $\boldsymbol{W}^l_{\mathrm{mlp1}} \in \mathbb{R}^{d_\mathrm{mlp} \times d_\mathrm{e}}$, $\boldsymbol{b}^l_{\mathrm{mlp1}} \in \mathbb{R}^{d_\mathrm{mlp}}$, $\boldsymbol{W}^l_{\mathrm{mlp2}} \in \mathbb{R}^{d_\mathrm{e} \times d_\mathrm{mlp}}$, $\boldsymbol{b}^l_{\mathrm{mlp2}} \in \mathbb{R}^{d_\mathrm{e}}$, MLP parameters.

$\boldsymbol{W_f} \in \mathbb{R}^{d_\mathrm{f} \times d_\mathrm{e}}$, $\boldsymbol{b_f} \in \mathbb{R}^{d_\mathrm{f}}$, $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{d_\mathrm{f}}$, the final linear projection and layer-norm parameters.

$\boldsymbol{W_u} \in \mathbb{R}^{N_\mathrm{V} \times d_\mathrm{e}}$, the unembedding matrix.

$\ell \leftarrow \mathrm{length}(\boldsymbol{x})$

for $t \in [\ell]$: $\boldsymbol{e}_t \leftarrow \boldsymbol{W_e}[:, x[t]] + \boldsymbol{W_p}[:, t]$

$\boldsymbol{X} \leftarrow [\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots \boldsymbol{e}_\ell]$

for $l = 1, 2, \ldots, L$ **do**

  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \texttt{MHAttention}(\boldsymbol{X}\,|\boldsymbol{\curlyvee W}_l, \mathrm{Mask} \equiv 1)$

  for $t \in [\ell]$: $\boldsymbol{X}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t]\,|\,\boldsymbol{\gamma}_l^1, \boldsymbol{\beta}_l^1)$

  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \boldsymbol{W}^l_{\mathrm{mlp2}} \texttt{GELU}(\boldsymbol{W}^l_{\mathrm{mlp1}} \boldsymbol{X} + \boldsymbol{b}^l_{\mathrm{mlp1}} \boldsymbol{1}^\intercal) + \boldsymbol{b}^l_{\mathrm{mlp2}} \boldsymbol{1}^\intercal$

  for $t \in [\ell]$: $\boldsymbol{X}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t]\,|\,\boldsymbol{\gamma}_l^2, \boldsymbol{\beta}_l^2)$

end

$\boldsymbol{X} \leftarrow \texttt{GELU}(\boldsymbol{W_f X} + \boldsymbol{b_f} \boldsymbol{1}^\intercal)$

for $t \in [\ell]$: $\boldsymbol{X}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t]\,|\,\boldsymbol{\gamma}, \boldsymbol{\beta})$

return $\boldsymbol{P} = \mathrm{softmax}(\boldsymbol{W_u X})$

Algorithm 10: $\boldsymbol{P} \leftarrow \texttt{DTransformer}(\boldsymbol{x}|\boldsymbol{\theta})$

/* GPT, a decoder-only transformer, forward pass */

Input: $\boldsymbol{x} \in V^*$, a sequence of token IDs.

Output: $\boldsymbol{P} \in (0,1)^{N_\text{V} \times \text{length}(\boldsymbol{x})}$, where the t-th column of $\boldsymbol{P}$ represents $\hat{P}_{\boldsymbol{\theta}}(x[t+1]|\boldsymbol{x}[1:t])$.

Hyperparameters: $\ell_\text{max}, L, H, d_\text{e}, d_\text{mlp} \in \mathbb{N}$

Parameters: $\boldsymbol{\theta}$ includes all of the following parameters:

$\boldsymbol{W_e} \in \mathbb{R}^{d_\text{e} \times N_\text{V}}$, $\boldsymbol{W_p} \in \mathbb{R}^{d_\text{e} \times \ell_\text{max}}$, the token and positional embedding matrices.

For $l \in [L]$:

| $\boldsymbol{\curlyvee W}_l$, multi-head attention parameters for layer l, see (4),

| $\boldsymbol{\gamma}_l^1, \boldsymbol{\beta}_l^1, \boldsymbol{\gamma}_l^2, \boldsymbol{\beta}_l^2 \in \mathbb{R}^{d_\text{e}}$, two sets of layer-norm parameters,

| $\boldsymbol{W}^l_{\text{mlp1}} \in \mathbb{R}^{d_\text{mlp} \times d_\text{e}}$, $\boldsymbol{b}^l_{\text{mlp1}} \in \mathbb{R}^{d_\text{mlp}}$, $\boldsymbol{W}^l_{\text{mlp2}} \in \mathbb{R}^{d_\text{e} \times d_\text{mlp}}$, $\boldsymbol{b}^l_{\text{mlp2}} \in \mathbb{R}^{d_\text{e}}$, MLP parameters.

$\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{d_\text{e}}$, final layer-norm parameters.

$\boldsymbol{W_u} \in \mathbb{R}^{N_\text{V} \times d_\text{e}}$, the unembedding matrix.

$\ell \leftarrow \text{length}(\boldsymbol{x})$

for $t \in [\ell]$: $\boldsymbol{e}_t \leftarrow \boldsymbol{W_e}[:, x[t]] + \boldsymbol{W_p}[:, t]$

$\boldsymbol{X} \leftarrow [\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots \boldsymbol{e}_\ell]$

for $l = 1, 2, \ldots, L$ **do**

  for $t \in [\ell]$: $\tilde{\boldsymbol{X}}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t] \,|\, \boldsymbol{\gamma}_l^1, \boldsymbol{\beta}_l^1)$

  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \texttt{MHAttention}(\tilde{\boldsymbol{X}} \,|\, \boldsymbol{\curlyvee W}_l, \text{Mask}[t, t'] = [[t \le t']])$

  for $t \in [\ell]$: $\tilde{\boldsymbol{X}}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t] \,|\, \boldsymbol{\gamma}_l^2, \boldsymbol{\beta}_l^2)$

  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \boldsymbol{W}^l_{\text{mlp2}} \texttt{GELU}(\boldsymbol{W}^l_{\text{mlp1}} \tilde{\boldsymbol{X}} + \boldsymbol{b}^l_{\text{mlp1}} \boldsymbol{1}^\intercal) + \boldsymbol{b}^l_{\text{mlp2}} \boldsymbol{1}^\intercal$

end

for $t \in [\ell]$: $\boldsymbol{X}[:, t] \leftarrow \texttt{layer\_norm}(\boldsymbol{X}[:, t] \,|\, \boldsymbol{\gamma}, \boldsymbol{\beta})$

return $\boldsymbol{P} = \text{softmax}(\boldsymbol{W_u X})$

Algorithm 11: $\hat{\theta} \leftarrow \texttt{EDTraining}(z_{1:N_{\text{data}}}, x_{1:N_{\text{data}}}, \theta)$

```
/* Training a seq2seq model */
Input: {(z_n, x_n)}_{n=1}^{N_data}, a dataset of sequence pairs.
Input: θ, initial transformer parameters.
Output: θ̂, the trained parameters.
Hyperparameters: N_epochs ∈ ℕ, η ∈ (0, ∞)
for i = 1, 2, ..., N_epochs do
    for n = 1, 2, ... N_data do
        ℓ ← length(x_n)
        P(θ) ← EDTransformer(z_n, x_n|θ)
        loss(θ) = −Σ_{t=1}^{ℓ−1} log P(θ)[x_n[t+1], t]
        θ ← θ − η · ∇loss(θ)
    end
end
return θ̂ = θ
```

Algorithm 12: $\hat{\theta} \leftarrow \texttt{ETraining}(x_{1:N_{\text{data}}}, \theta)$

```
/* Training by masked language modelling */
Input: {x_n}_{n=1}^{N_data}, a dataset of sequences.
Input: θ, initial encoder-only transformer parameters.
Output: θ̂, the trained parameters.
Hyperparameters: N_epochs ∈ ℕ, η ∈ (0, ∞), p_mask ∈ (0, 1)
for i = 1, 2, ..., N_epochs do
    for n = 1, 2, ..., N_data do
        ℓ ← length(x_n)
        for t = 1, 2, ..., ℓ do
            x̃_n[t] ← mask_token or x_n[t] randomly with probability p_mask or 1 − p_mask
        end
        T̃ ← {t ∈ [ℓ] : x̃_n[t] = mask_token}
        P(θ) ← ETransformer(x̃_n | θ)
        loss(θ) = −Σ_{t∈T̃} log P(θ)[x_n[t], t]
        θ ← θ − η · ∇loss(θ)
    end
end
return θ̂ = θ
```

Algorithm 13: $\hat{\theta} \leftarrow \texttt{DTraining}(x_{1:N_{\text{data}}}, \theta)$

```
/* Training next token prediction */
Input: {x_n}_{n=1}^{N_data}, a dataset of sequences.
Input: θ, initial decoder-only transformer parameters.
Output: θ̂, the trained parameters.
Hyperparameters: N_epochs ∈ ℕ, η ∈ (0, ∞)
for i = 1, 2, ..., N_epochs do
    for n = 1, 2, ... N_data do
        ℓ ← length(x_n)
        P(θ) ← DTransformer(x_n | θ)
        loss(θ) = −Σ_{t=1}^{ℓ−1} log P(θ)[x_n[t+1], t]
        θ ← θ − η · ∇loss(θ)
    end
end
return θ̂ = θ
```

Algorithm 14: $y \leftarrow \texttt{DInference}(x, \hat{\theta})$

```
/* Prompting a trained model and using it for prediction. */
Input: Trained transformer parameters θ̂.
Input: x ∈ V*, a prompt.
Output: y ∈ V*, the transformer's continuation of the prompt.
Hyperparameters: ℓ_gen ∈ ℕ, τ ∈ (0, ∞)
ℓ ← length(x)
for i = 1, 2, ... ℓ_gen do
    P ← DTransformer(x | θ̂)
    p ← P[:, ℓ + i − 1]
    sample a token y from q ∝ p^{1/τ}
    x ← [x, y]
end
return y = x[ℓ + 1 : ℓ + ℓ_gen]
```

Algorithm 15: $\hat{\boldsymbol{x}} \leftarrow \texttt{EDInference}(\boldsymbol{z}, \hat{\boldsymbol{\theta}})$

```
/* Using a trained seq2seq model for
   prediction.                        */
```

Input: A seq2seq transformer and trained parameters $\hat{\boldsymbol{\theta}}$ of the transformer.

Input: $\boldsymbol{z} \in V^*$, input sequence, e.g. a sentence in English.

Output: $\hat{\boldsymbol{x}} \in V^*$, output sequence, e.g. the sentence in German.

Hyperparameters: $\tau \in (0, \infty)$

$\hat{\boldsymbol{x}} \leftarrow [\texttt{bos\_token}]$
$y \leftarrow 0$
while $y \neq \textit{eos\_token}$ **do**
  $\boldsymbol{P} \leftarrow \texttt{EDTransformer}(\boldsymbol{z}, \hat{\boldsymbol{x}} \,|\, \hat{\boldsymbol{\theta}})$
  $\boldsymbol{p} \leftarrow \boldsymbol{P}[:, \text{length}(\hat{\boldsymbol{x}})]$
  sample a token y from $\boldsymbol{q} \propto \boldsymbol{p}^{1/\tau}$
  $\hat{\boldsymbol{x}} \leftarrow [\hat{\boldsymbol{x}}, y]$
end
return $\hat{\boldsymbol{x}}$

B. List of Notation

Symbol	Type	Explanation
$[N]$	$:= \{1, ..., N\}$	set of integers $1, 2, ..., N-1, N$
i, j	$\in \mathbb{N}$	generic integer indices
V	$\cong [N_\text{V}]$	vocabulary
N_V	$\in \mathbb{N}$	vocabulary size
V^*	$= \bigcup_{\ell=0}^{\infty} V^\ell$	set of token sequences; elements include e.g. sentences or documents
ℓ_max	$\in \mathbb{N}$	maximum sequence length
ℓ	$\in [\ell_\text{max}]$	length of token sequence
t	$\in [\ell]$	index of token in a sequence
$d_\text{...}$	$\in \mathbb{N}$	dimension of various vectors
$\boldsymbol{x}$	$\equiv x[1:\ell]$	$\equiv x[1]x[2]...x[\ell] \in V^\ell$ primary token sequence
$\boldsymbol{z}$	$\equiv z[1:\ell]$	$\equiv z[1]z[2]...z[\ell] \in V^\ell$ context token sequence
$M[i,j]$	$\in \mathbb{R}$	entry M_{ij} of matrix $M \in \mathbb{R}^{d \times d'}$
$M[i,:] \equiv M[i]$	$\in \mathbb{R}^{d'}$	i-th row of matrix $M \in \mathbb{R}^{d \times d'}$
$M[:,j]$	$\in \mathbb{R}^{d}$	j-th column of matrix $M \in \mathbb{R}^{d \times d'}$
$\boldsymbol{e}$	$\in \mathbb{R}^{d_\text{e}}$	vector representation / embedding of a token
$\boldsymbol{X}$	$\in \mathbb{R}^{d_\text{e} \times \ell_x}$	encoded primary token sequence
$\boldsymbol{Z}$	$\in \mathbb{R}^{d_\text{e} \times \ell_z}$	encoded context token sequence
Mask	$\in \mathbb{R}^{\ell_z \times \ell_x}$	masking matrix, it determines the attention context for each token
$L, L_\text{enc}, L_\text{dec}$	$\in \mathbb{N}$	number of network (encoder, decoder) layers
l	$\in [L]$	index of network layer
H	$\in \mathbb{N}$	number of attention heads
h	$\in [H]$	index of attention head
N_data	$\in \mathbb{N}$	(i.i.d.) sample size
n	$\in [N_\text{data}]$	index of sample sequence
η	$\in (0, \infty)$	learning rate
τ	$\in (0, \infty)$	temperature; it controls the diversity-plausibility trade-off at inference
$\boldsymbol{W_e}$	$\in \mathbb{R}^{d_\text{e} \times N_\text{V}}$	token embedding matrix
$\boldsymbol{W_p}$	$\in \mathbb{R}^{d_\text{e} \times \ell_\text{max}}$	positional embedding matrix
$\boldsymbol{W_u}$	$\in \mathbb{R}^{N_\text{V} \times d_\text{e}}$	unembedding matrix
$\boldsymbol{W_q}$	$\in \mathbb{R}^{d_\text{attn} \times d_\text{x}}$	query weight matrix
$\boldsymbol{b_q}$	$\in \mathbb{R}^{d_\text{attn}}$	query bias
$\boldsymbol{W_k}$	$\in \mathbb{R}^{d_\text{attn} \times d_\text{z}}$	key weight matrix
$\boldsymbol{b_k}$	$\in \mathbb{R}^{d_\text{attn}}$	key bias
$\boldsymbol{W_v}$	$\in \mathbb{R}^{d_\text{out} \times d_\text{z}}$	value weight matrix
$\boldsymbol{b_v}$	$\in \mathbb{R}^{d_\text{out}}$	value bias
$\boldsymbol{'W_{qkv}}$		collection of above parameters of a single-head attention layer
$\boldsymbol{W_o}$	$\in \mathbb{R}^{d_\text{out} \times H d_\text{mid}}$	output weight matrix
$\boldsymbol{b_o}$	$\in \mathbb{R}^{d_\text{out}}$	output bias
$\boldsymbol{'W}$		collection of above parameters of a multi-head attention layer, see eq. (4)
$\boldsymbol{W_\text{mlp}}$	$\in \mathbb{R}^{d_1 \times d_2}$	weight matrix corresponding to an MLP layer in a Transformer
$\boldsymbol{b_\text{mlp}}$	$\in \mathbb{R}^{d_1}$	bias corresponding to an MLP layer in a Transformer
$\boldsymbol{\gamma}$	$\in \mathbb{R}^{d_\text{e}}$	layer-norm learnable scale parameter
$\boldsymbol{\beta}$	$\in \mathbb{R}^{d_\text{e}}$	layer-norm learnable offset parameter
$\boldsymbol{\theta}, \hat{\boldsymbol{\theta}}$	$\in \mathbb{R}^{d}$	collection of all learnable / learned Transformer parameters