Decoder-Only Transformers

Notes on various aspects of Decoder-Only Transformers.

Contents

1 Decoder-Only Fundamentals

The Transformers architecture [1], which dominates Natural Language Processing (NLP) as of July 2023, is a relatively simple architecture. There are various flavors and variants of Tranformers, but focus here on the decoder-only versions which underlie the GPT models [2–4].

The full decoder-only architecture can be seen in Fig. 1. The parameters which define the network can be found in App. A.

An outline of the mechanics:

1. Raw text is **tokenized** and turned into a series of integers[1] whose values lie in `range(V)`, with V the vocabulary size.
2. The tokenized text is chunked and turned into `(B, S)`-shaped (batch size and sequence length, respectively) integer tensors, x_{bs}.
3. The **embedding layer** converts the integer tensors into continuous representations of shape `(B, S, D)`, z_{bsd}, with D the size of the hidden dimension. **Positional encodings** have

[1] There are about 1.3 tokens per word, on average.

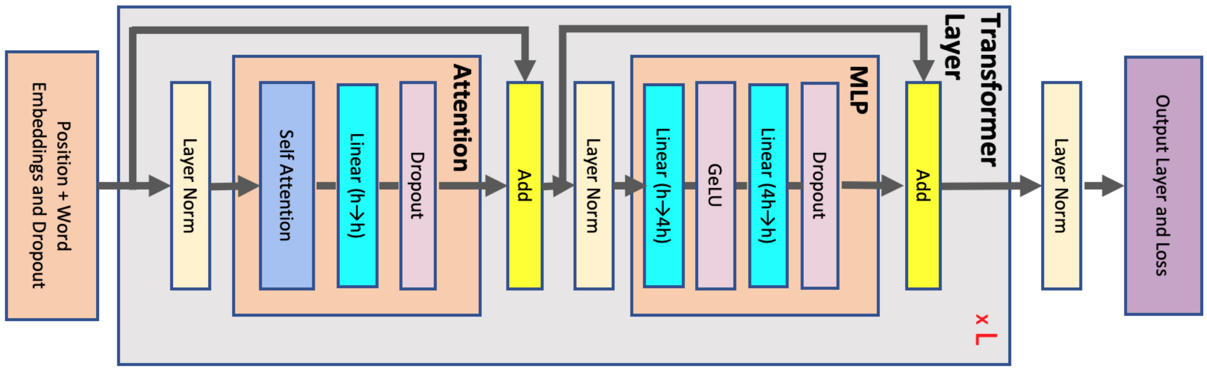

Figure 1. The full transformers architecture. Diagram taken from [5]

also been added to the tensor at this stage to help the architecture understand the relative ordering of the text.

4. The z_{bsd} tensors pass through a series of transformer blocks, each of which has two primary components:

   (a) In the **attention** sub-block, components of z_{bsd} at different positions (s-values) interact with each other, resulting in another `(B, S, D)`-shaped tensor, z'_{bsd}.

   (b) In the **MLP** block, each position in z'_{bsd} is processed independently and in parallel by a two-layer feed-forward network, resulting once more in a `(B, S, D)`-shaped tensor.

   Importantly, there are **residual connections** around each of these[2] (the arrows in Fig. 1).

5. Finally, we convert the `(B, S, D)`-shaped tensors to `(B, S, V)`-shaped ones, y_{bsv}. This is the role of the **language model head** (which is often just the embedding layer used in an inverse manner.)

6. The y_{bsv} predict what the next token will be, having seen the **context** of the first s tokens in the sequence.

Each batch (the b-index) is processed independently. We omitted `LayerNorm` and `Dropout` layers above, as well as the causal mask; these will be covered below as we step through the architecture in more detail.

1.1 Component Details

We break down the various components below.

[2] This gives rise to the concept of the **residual stream** which each transformer block reads from and writes back to repeatedly.

1.1.1 Embedding Layer and Positional Encodings

The **embedding** layer is just a simple look up table: each of the `range(V)` indices in the vocabulary is mapped to a D-dimensional vector via a large `(V, D)`-shaped table/matrix. This layer maps $x_{bs} \longrightarrow z_{bsd}$. In `torch`, this is an `nn.Embedding(V, D)` instance.

To each item in a batch, we add identical **positional encodings** to the vectors above with the goal of adding fixed, position-dependent correlations in the sequence dimension which will hopefully make it easier for the architecture to pick up on the relative positions of the inputs [3] This layer maps $z_{bsd} \leftarrow z_{bsd} + p_{sd}$, with p_{sd} the positional encoding tensor.

The above components require $(V + S)D \approx VD$ parameters per layer.

1.1.2 Layer Norm

The original transformers paper [1] put `LayerNorm` instances after the **attention** and **MLP** blocks, but now it is common [6] to put them before these blocks[4].

The `LayerNorm` operations acts over the sequence dimension. Spelling it out, given the input tensor z_{bsd} whose mean and variance over the s-index are μ_{bd} and σ_{bd}, respectively, the `LayerNorm` output is

$$z_{bsd} \leftarrow \left(\frac{z_{bsd} - \mu_{bd}}{\sigma_{bd}}\right) \times \gamma_d + \beta_d \equiv \text{LayerNorm}_s z_{bsd} \tag{1.1}$$

where γ_d, β_d are the trainable scale and bias parameters. In `torch`, this is a `nn.LayerNorm(D)` instance.

Since there are two `LayerNorm` instances in each transformer block, these components require $2D$ parameters per layer.

1.1.3 Causal Attention

The **causal attention** layer is the most complex layer. It features H triplets[5] of weight matrices[6] $Q^h_{df}, K^h_{df}, V^h_{df}$ where $a \in \{0, \ldots, H-1\}$ and $f \in \{0, \ldots, D/H\}$. From these, we form three different vectors:

$$q^h_{bsf} = z_{bsd} Q^h_{df} \;, \quad k^h_{bsf} = z_{bsd} K^h_{df} \;, \quad v^h_{bsf} = z_{bsd} V^h_{df} \tag{1.2}$$

These are the **query, key, and value** tensors, respectively.

Using the above tensors, we will then build up an **attention map** $w^h_{bss'}$ which corresponds to how much attention the token at position s pays to the token at position s'. Because we have the goal of predicting the next token in the sequence, we need these weights to be causal: the final

[3] Positional encodings and the causal mask are the only components in the transformers architecture which carry weights with a dimension of size S; i.e. they are the only parts that have explicit sequence-length dependence. A related though experiment: you can convince yourself that if the inputs z_{bsd} were just random noise, the transformers architecture would not be able to predict the s-index of each such input in the absence of positional encodings.

[4] Which makes intuitive sense for the purposes of stabilizing the matrix multiplications in the blocks

[5] H must divide the hidden dimension D evenly.

[6] There are also bias terms, but we will often neglect to write them explicitly or account for their (negligible) parameter count.

prediction y_{bsv} should only have access to information propagated from positions $x_{bs'v}$ with $s' \leq s$. This corresponds to the condition that $w^h_{bss'} = 0$ if $s' > s$.

These weights come from `Softmax`-ed attention scores, which are just a normalized dot-product over the hidden dimension:

$$w^h_{bss'd} = \text{Softmax}_{\,s'}\left(m_{ss'} + \; q^h_{bsf} k^h_{bs'f} / \sqrt{D/H}\right) \;, \quad \text{s.t.} \quad \sum_{s'} w^h_{bdss'} = 1 \tag{1.3}$$

The tensor $m_{ss'}$ is the causal mask which zeroes out the relevant attention map components above

$$m_{ss'} = \begin{cases} 0 & s \leq s' \\ -\infty & = s > s' \end{cases}$$

The $\sqrt{D/H}$ normalization is motivated by demanding that the variance of the `Softmax` argument be 1 at initialization, assuming that other components have been configured so that that the query and key components are i.i.d. from a Gaussian normal distribution [7].

The weights above are then passed through a dropout layer and used to re-weigh the **value** vectors and form the tensors

$$t^h_{bsf} = \text{Drop}\left(w^h_{bdss'}\right) v^h_{bs'f} \tag{1.4}$$

and these H different `(B, S, D/H)`-shaped tensors are then concatenated along the f-direction to re-form a `(B, S, D)`-shaped tensor[8]

$$u_{bsd} = \text{Concat}^a_{fd}\left(t^a_{bsf}\right) \;. \tag{1.5}$$

Finally, another weight matrix $O_{d'd}$ and dropout layer transform the output once again to get the final output

$$z_{bsd} = \text{Drop}\left(u_{bsd'} O_{d'd}\right) \;. \tag{1.6}$$

For completeness, the entire operation in condensed notation with indices left implicit is:

$$z \leftarrow \text{Drop}\left(\text{Concat}\left(\text{Drop}\left(\text{Softmax}\left(\frac{\left(z \cdot Q^h\right) \cdot \left(z \cdot K^h\right)}{\sqrt{D/H}}\right)\right) \cdot z \cdot V^h\right) \cdot O\right) \tag{1.7}$$

where all of the dot-products are over feature dimensions (those of size D or D/H). H nice[9], but hopefully pedagogically useful, implementation of the attention layer is below:

```python
class CausalAttention(nn.Module):
    def __init__(
        self,
```

[7] However, in [7] it is instead argued that no square root should be taken in order to maximize the speed of learning via SGD.

[8] It is hard to come up with good index-notation for concatenation.

[9] An example optimization: there is no need to form separate Q^h, K^h, V^h `Linear` layers, one large layer which is later chunked is more efficient

```
        attn_heads=H,
        hidden_dim=D,
        block_size=K,
        dropout=0.1,
    ):
        super().__init__()
        self.head_dim, remainder = divmod(hidden_dim, attn_heads)
        assert not remainder, "attn_heads must divide hidden_dim evenly"

        self.Q = nn.ModuleList([nn.Linear(hidden_dim, self.head_dim) for _ in range(attn_heads)])
        self.K = nn.ModuleList([nn.Linear(hidden_dim, self.head_dim) for _ in range(attn_heads)])
        self.V = nn.ModuleList([nn.Linear(hidden_dim, self.head_dim) for _ in range(attn_heads)])
        self.O = nn.Linear(hidden_dim, hidden_dim)

        self.attn_dropout = nn.Dropout(dropout)
        self.final_dropout = nn.Dropout(dropout)
        self.register_buffer(
            "causal_mask",
            torch.tril(torch.ones(block_size, block_size)[None]),
        )

    def get_qkv(self, inputs):
        queries = [q(inputs) for q in self.Q]
        keys = [k(inputs) for k in self.K]
        values = [v(inputs) for v in self.V]
        return queries, keys, values

    def get_attn_maps(self, queries, keys, values, seq_len):
        norm = math.sqrt(self.head_dim)
        non_causal_attn_scores = [(q @ k.transpose(-2, -1)) / norm for q, k in zip(queries, keys)]
        causal_attn_scores = [
            a.masked_fill(self.causal_mask[:, :S, :S] == 0, float("-inf"))
            for a in non_causal_attn_scores
        ]
        attn_maps = [a.softmax(dim=-1) for a in causal_attn_scores]
        return attn_maps

    def forward(self, inputs):
        S = inputs.shape[1]
        queries, keys, values = self.get_qkv(inputs)
        attn_maps = self.get_attn_maps(queries, keys, values, S)
        weighted_values = torch.concatenate(
            [self.attn_dropout(a) @ v for a, v in zip(attn_maps, values)], dim=-1
        )
        z = self.final_dropout(self.O(weighted_values))
```

The parameter count is dominated by the weight matrices which carry $4D^2$ total parameters per layer.

1.1.4 MLP

The feed-forward network is straightforward and corresponds to

$$z_{bsd} \leftarrow \phi\left(z_{bsd'} W^0_{d'e}\right) W^1_{ed} \tag{1.8}$$

where W^0 and W^1 are `(D, FD)`- and `(FD, D)`-shaped matrices, respectively (see App. A for notation) and ϕ is a non-linearity[10]. The implementation is straightforward:

```
class MLP(nn.Module):
    def __init__(
        self,
        hidden_dim=D,
        expansion_factor=F,
        dropout=0.1,
    ):
        super().__init__()
        linear_1 = nn.Linear(hidden_dim, expansion_factor * hidden_dim)
        linear_2 = nn.Linear(expansion_factor * hidden_dim, hidden_dim)
        gelu = nn.GELU()
        drop = nn.Dropout(dropout)
        self.layers = nn.Sequential(linear_1, gelu, linear_2, drop)

    def forward(self, inputs):
        z = self.layers(inputs)
        return z
```

This bock requires $2FD^2$ parameters per layer, only counting the contribution from weights.

1.1.5 Language Model Head

The layer which converts the `(B, S, D)`-shaped outputs, z_{bsd}, to `(B, S, V)`-shaped predictions over the vocabulary, y_{bsv}, is the **Language Model Head**. It is a linear layer, whose weights are usually tied to be exactly those of the initial embedding layer of Sec. 1.1.1.

1.1.6 All Together

It is then relatively straightforward to tie everything together. In code, we can first create a transformer block like

```
class TransformerBlock(nn.Module):
    def __init__(
        self,
        attn_heads=H,
        block_size=K,
        dropout=0.1,
        expansion_factor=F,
        hidden_dim=D,
        layers=L,
        vocab_size=V,
    ):
        super().__init__()
        self.attn_ln = nn.LayerNorm(hidden_dim)
        self.mlp_ln = nn.LayerNorm(hidden_dim)
        self.attn = CausalAttention(attn_heads, hidden_dim, block_size, dropout)
        self.mlp = MLP(hidden_dim, expansion_factor, dropout)

```

[10]The `GeLU` non-linearity is common.

```python
    def forward(self, inputs):
        z = self.attn(self.attn_ln(inputs)) + inputs
        z = self.mlp(self.mlp_ln(z)) + z
        return z
```

And then the entire architecture:

```python
class DecoderOnly(nn.Module):
    def __init__(
        self,
        attn_heads=H,
        block_size=K,
        dropout=0.1,
        expansion_factor=F,
        hidden_dim=D,
        layers=L,
        vocab_size=V,
    ):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_dim)
        self.pos_encoding = nn.Parameter(torch.randn(1, block_size, hidden_dim))
        self.drop = nn.Dropout(dropout)
        self.trans_blocks = nn.ModuleList(
            [
                TransformerBlock(
                    attn_heads,
                    block_size,
                    dropout,
                    expansion_factor,
                    hidden_dim,
                    layers,
                    vocab_size,
                )
                for _ in range(layers)
            ]
        )
        self.final_ln = nn.LayerNorm(hidden_dim)
        self.lm_head = nn.Linear(hidden_dim, vocab_size, bias=False)
        self.lm_head.weight = self.embedding.weight  # Weight tying.

    def forward(self, inputs):
        S = inputs.shape[1]
        z = self.embedding(inputs) + self.pos_encoding[:, :S]
        z = self.drop(z)
        for block in self.trans_blocks:
            z = block(z)
        z = self.final_ln(z)
        z = self.lm_head(z)
        return z

```

2 Memory and Activations

In this section we summarize the train-time memory costs of Transformers under various training strategies. At a high level, the memory cost is much more than simply the cost of the model parameters. Significant factors include:

- Optimizer states, like those of `Adam`
- Mixed precision training costs, due to keeping multiple model copies.
- Activation memory[11] , needed for backpropagation.
- Gradients

Activations require the most detailed analysis, so we start with their analysis.

2.1 No Sharding/Parallelism

Start with the simplest case where no parallelism are used. The costs of training then come from the model parameters, the optimizer state, the gradients, and the activations. Every parameter is assumed to be stored in `fp32`, i.e. four bytes per parameter [12] .

We will assume the use of `Adam` throughout, for simplicity, which stores two different running averages per-model parameter. Even in this vanilla scenario, the cost of the optimizer states is significant.

Next, we eventually will get gradients out of back propagation, one-per parameter, and so the gradient cost is also equal to the model weight cost.

Finally, activations, which require more analysis [5]. Unlike the above factors, the activation memory will depend on both the batch size and input sequence length, B and S, scaling linearly with both.

2.1.1 MLP Activations

We will count the number of elements which need to be cached. First we cache the `(B, S, D)`-shaped inputs into the first MLP layer. These turn into the `(B, S, FD)` inputs of the non-linearity, which are then passed into the last `Linear` layer. This sums to the following number of cached activations elements:

$$N_{\text{act}}^{\texttt{MLP}} = (2F + 1)BS \ . \tag{2.1}$$

The surprisingly

[11] Activations refers to any intermediate value which needs to be cached in order to compute backpropagation. We will be conservative and assume that the inputs of all operations need to be stored, though in practice gradient checkpointing and recomputation allow one to trade caching for redundant compute. In particular, flash attention [8] makes use of this strategy.

[12] We will usually express memory in terms of GiB, where 1 GiB $\equiv 2^{30}$ bytes.

2.2 Optimizer States and Mixed Precision

From the previous section, the pure parameter-count of the decoder-only Transformers architecture is

$$N_{\text{params}} \approx (4 + 2F)LD^2 + VD + \mathcal{O}(LD) \tag{2.2}$$

where the first term comes from the `TransformerBlock` weight matrices, the next term is the embedding matrix, and the neglected terms come from biases, `LayerNorm` instances, and other negligible factors. [13]

The memory cost of the bare parameters are usually dwarfed

A Conventions and Notation

We loosely follow the conventions of [5] and denote the main Transformers parameters by:

- B: microbatch size
- K: the block size (maximum sequence length[14])
- S: input sequence length
- V: vocabulary size
- D: the hidden dimension size
- L: number of transformer layers
- H: number of attention heads
- P: pipeline parallel size
- T: tensor parallel size
- F: expansion factor for MLP layer (usually $F = 4$)

Where it makes sense, we try to use the lower-case versions of these characters to denote the corresponding indices on various tensors. For instance, an input tensor with the above batch size, sequence length, and vocabulary size would be written as x_{bsv}, with $b \in \{0, \ldots, B-1\}$, $s \in \{0, \ldots, S-1\}$, and $v \in \{0, \ldots, V-1\}$ in math notation, or as `x[b, s, v]` in code. Typical transformers belong to the regime

$$V \gg D, S \gg L, H \gg P, T \ . \tag{A.1}$$

As indicated above, we use zero-indexing. We also use `python` code throughout[15] and write all ML code using standard `torch` syntax. To avoid needing to come up with new symbols in math

[13] For the usual $F = 4$ case, the `MLP` layers are twice as costly as the `CausalAttention` layers.
[14] In the absence of methods such as ALiBi [9] can be used to extend the sequence length at inference time.
[15] Written in a style conducive to latex, e.g. no type-hints and pegagogy prioritized.

expressions we will often use expressions like $x \leftarrow f(x)$ to refer to performing a computation on some argument (x) and assigning the result right back to the variable x again.

Physicists often joke (half-seriously) that Einstein's greatest contribution to physics was his summation notation in which index-sums are implied by the presence of repeated indices and summation symbols are entirely ommitted. For instance, the dot product between two vectors would be written as

$$\vec{x} \cdot \vec{y} = \sum_i x_i y_i \equiv x_i y_i \tag{A.2}$$

We use similar notation which is further adapted to the common element-wise deep-learning operations. The general rule is that if a repeated index appears on one side of an equation, but not the other, then a sum is implied, but if the same index appears on both sides, then it's an element-wise operation. The Hadamard-product between two matrices A and B is just

$$C_{ij} = A_{ij} B_{ij} \ . \tag{A.3}$$

We also put explicit indices on operators such as Softmax to help clarify the relevant dimension, e.g. we would write the softmax operation over the b-ndex of some batched tensor $x_{bvd\ldots}$ as

$$s_{bvd\ldots} = \frac{e^{x_{bvd\ldots}}}{\sum_{v=0}^{v=V-1} e^{x_{bvd\ldots}}} \equiv \text{Softmax}_{\,v}\ x_{bvd\ldots}\ , \tag{A.4}$$

indicating that the sum over the singled-out v-index is gives unity.

B TODO

- Tokenizers
- Generation
- Activations
- Flash attention
- BERT family
- Residual stream
- Scaling laws

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," arXiv:1706.03762 [cs.CL]. 1, 3

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog* **1** (2019) no. 8, 9. 1

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," `arXiv:2005.14165 [cs.CL]`.

[4] OpenAI, "Gpt-4 technical report," `arXiv:2303.08774 [cs.CL]`. 1

[5] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," `arXiv:2205.05198 [cs.LG]`. 2, 8, 9

[6] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu, "On layer normalization in the transformer architecture," `arXiv:2002.04745 [cs.LG]`. 3

[7] G. Yang, E. J. Hu, I. Babuschkin, S. Sidor, X. Liu, D. Farhi, N. Ryder, J. Pachocki, W. Chen, and J. Gao, "Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer," `arXiv:2203.03466 [cs.LG]`. 4

[8] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," `arXiv:2205.14135 [cs.LG]`. 8

[9] O. Press, N. A. Smith, and M. Lewis, "Train short, test long: Attention with linear biases enables input length extrapolation," *CoRR* **abs/2108.12409** (2021) , `2108.12409`. `https://arxiv.org/abs/2108.12409`. 9