Decoder-Only Transformers

Notes on various aspects of Decoder-Only Transformers.

Contents

Part I

Architecture

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.

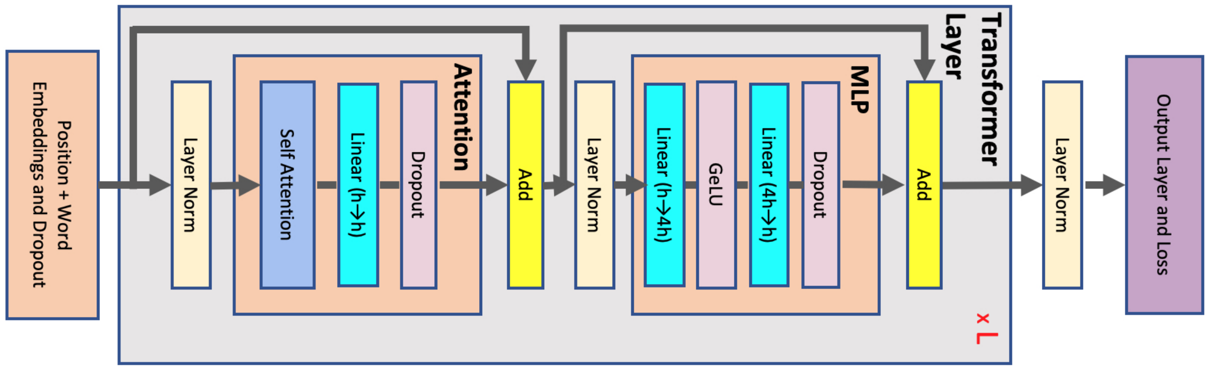

Figure 1. The full transformers architecture. Diagram taken from [5]

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 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}.

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

(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.

We break down the various components below in detail.

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.

1.2 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.

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

[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

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

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 [7].

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 prediction y_{bsv} should only have access to information propagated from positions $x_{bs'v}$ with $s' \le s$. 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

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

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 [8].

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[9]

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

[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.

[7] There are of course many variants of the architecture and one variant which is popular in Summer 2023 is multi-query attention [7] in which all heads share *the same* key and value vectors and only the query changes across heads, as this greatly reduces inference costs.

[8] However, in [8] it is instead argued that no square root should be taken in order to maximize the speed of learning via SGD.

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

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[10], but hopefully pedagogically useful, implementation of the attention layer is below:

```python
class CausalAttention(nn.Module):
    def __init__(
        self,
        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[:, :seq_len, :seq_len] == 0, float("-inf"))
            for a in non_causal_attn_scores
        ]
```

[10] 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_maps = [a.softmax(dim=-1) for a in causal_attn_scores]
        return attn_maps

    def forward(self, inputs):
        seq_len = inputs.shape[1]
        queries, keys, values = self.get_qkv(inputs)
        attn_maps = self.get_attn_maps(queries, keys, values, seq_len)
        weighted_values = torch.cat(
            [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.

1.4 MLP

The feed-forward network is straightforward and corresponds to

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

where W^0 and W^1 are `(D, ED)`- and `(ED, D)`-shaped matrices, respectively (see App. A for notation) and ϕ is a non-linearity[11]. The implementation is straightforward:

```
class MLP(nn.Module):
    def __init__(
        self,
        hidden_dim=D,
        expansion_factor=E,
        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 $2ED^2$ parameters per layer, only counting the contribution from weights.

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.

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

1.6 All Together

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

```python
class TransformerBlock(nn.Module):
    def __init__(
        self,
        attn_heads=H,
        block_size=K,
        dropout=0.1,
        expansion_factor=E,
        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)

    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=E,
        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

```

1.7 The Loss Function

The last necessary component is the loss function. The training loop data is canonically the `(B, K)`-shaped[12] token inputs (x_{bs}) along with their shifted-by-one relatives y_{bs} where `x[:, s + 1] == y[:, s]`. The `(B, K, V)`-shaped outputs (z_{bsv}) of the `DecoderOnly` network are treated as the logits which predict the value of the next token, given the present context:

$$p(x_{b(s+1)} = v|x_{bs}, x_{b(s-1)}, \ldots, x_{b0}) = \text{Softmax}_{\,v}\ z_{bsv} \tag{1.9}$$

and so the model is trained using the usual cross-entropy/maximum-likelihood loss

$$\begin{aligned}\mathcal{L}(x, z) &= \frac{1}{BK}\sum_{b,s} \ln p(x_{b(s+1)} = |x_{bs}, x_{b(s-1)}, \ldots, x_{b0}) \\ &= \frac{1}{BK}\sum_{b,s} \text{Softmax}_{\,v}\ z_{bsv}\big|_{v=y_{bs}}\ .\end{aligned} \tag{1.10}$$

Note that the losses for all possible context lengths are included in the sum.

In `torch` code, the loss computation might look like the following:

```
model = DecoderOnly(
    attn_heads=H,
    block_size=K,
    dropout=0.1,
    expansion_factor=E,
    hidden_dim=D,
    layers=L,
    vocab_size=V,
)
tokens = torch.randint(V, size=(B, K + 1))
inputs, targets = tokens[:, :-1], tokens[:, 1:]
```

[12]`K` is the block size, the maximum sequence-length for the model. See App. A.

```
    outputs = model(inputs)
    outputs_flat, targets_flat = outputs.reshape(-1, outputs.shape[-1]), targets.reshape(-1)
    loss = F.cross_entropy(outputs_flat, targets_flat)
```

Part II
Training

2 Memory

In this section we summarize the train-time memory costs of Transformers under various training strategies and the total number of floating point operations[13] needed to process a given batch of data.

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.
- Gradients
- Activation memory[14] , needed for backpropagation.

Because the activation counting is a little more involved, it is in its own section.

Essentials

Memory costs count the elements of all tensors in some fashion, both from model parameters and intermediate representations. The gradient and optimizer state costs scale with the former quantity: $\mathcal{O}\left(N_{\text{params}}\right) \sim \mathcal{O}\left(LD^2\right)$, only counting the dominant contributions from weight matrices. Activation memory scales with the latter, which for a `(B, S, D)`-shaped input gives $\mathcal{O}\left(LBSD\right)$ contributions from tensors which preserve the input shape, as well as $\mathcal{O}\left(LBHS^2\right)$ factors from attention matrices.

2.1 No Sharding

Start with the simplest case where there is no sharding of the model states. Handling the different parallelism strategies later will be relatively straightforward, as it involves inserting just a few factors here and there.

[13]The notation surrounding floating-point operations is very confusing because another quantity of interest is the number of floating-point operations a given implementation can use *per-second*. Sometimes, people use FLOPs or FLOP/s to indicate the rate, rather than the gross-count, but there's little consistency in general.

[14]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 [9] makes use of this strategy.

2.1.1 Parameters, Gradients, Optimizer States, and Mixed Precision

Memory from the bare parameter cost, gradients, and optimizer states are fixed costs independent of batch size and sequence-length (unlike activation memory), so we discuss them all together here. The parameter and optimizer costs are also sensitive to whether or not mixed-precision is used, hence we also address that topic, briefly. We will assume the use of `Adam`[15] throughout, for simplicity and concreteness. It will sometimes be useful below to let p to denote the precision in bytes that any given element is stored in, so `torch.float32` corresponds to $p = 4$, for instance. Ultimately, we primarily consider vanilla training in $p = 4$ precision and mixed-precision; generalizations to other training formats should be straightforward.

Without mixed precision, the total cost of the `torch.float32` ($p = 4$ bytes) model and optimizer states in bytes is then

$$M^{\text{model}} = 4N_{\text{params}} \ , \quad M^{\text{optim}} = 8N_{\text{params}} \quad (\text{no mixed precision,} p = 4) \tag{2.1}$$

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

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

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. [16] The optimizer states cost double the model itself.

The situation is more complicated when mixed-precision is used [10]. The pertinent components of mixed-precision:

- A half-precision ($p = 2$ bytes) copy of the model is used to perform the forwards and backwards passes
- A second, "master copy" of the model is also kept with weights in full $p = 4$ precision
- The internal `Adam` states are kept in full-precision

Confusingly, the master copy weights are usually accounted for as part of the optimizer state, in which case the above is altered to

$$M^{\text{model}} = 2N_{\text{params}} \ , \quad M^{\text{optim}} = 12N_{\text{params}} \quad (\text{mixed precision}). \tag{2.3}$$

The optimizer state is now six times the cost of the actual model used to process data and the costs of (2.3) are more than double those of (2.1). The point of mixed precision is speed, the ability to take advantage of specialized hardware, not memory savings. The above also demonstrates why training is so much more expensive than inference.

[15]Which stores two different running averages per-model parameter.

[16]For the usual $E = 4$ case, the `MLP` layers are twice as costly as the `CausalAttention` layers.

2.1.2 Gradients

Gradients are pretty simple and always cost the same regardless of whether or not mixed-precision is used:

$$M^{\text{grad}} = 4N_{\text{params}} \ . \tag{2.4}$$

In mixed precision, even though the gradients are initially computed in $p = 2$, they have to be converted to $p = 4$ to be applied to the master weights of the same precision.

2.1.3 Activations

Activations will require a more extended analysis [5]. Unlike the above results, the activation memory will depend on both the batch size and input sequence length, B and S, scaling linearly with both.

Attention Activations We will count the number of input elements which need to be cached. Our `(B, S, D)`-shaped inputs to the attention layer with BSD elements are first converted to $3BSD$ total query, key, value elements, and the query-key dot products produce HBS^2 more, which are softmaxed into HBS^2 normalized scores. The re-weighted inputs to the final linear layer also have BSD elements, bringing the running sum to $BS\,(5D + 2HS)$

Finally, there are also the dropout layers applied to the normalized attention scores and the final output whose masks must be cached in order to backpropagate. In torch, the mask is a `torch.bool` tensor, but surprisingly these use one *byte* of memory per element, rather than one bit [17]. Given this, the total memory cost from activations per attention layer is

$$M_{\text{act}}^{\texttt{Attention}} = BS\left((5p+1)D + (2p+1)HS\right) \tag{2.5}$$

MLP Activations First we cache the `(B, S, D)`-shaped inputs into the first MLP layer. These turn into the `(B, S, ED)` inputs of the non-linearity, which are then passed into the last `Linear` layer, summing to $(2E+1)BSD$ total elements thus far. Adding in the dropout mask, the total memory requirement is

$$M_{\text{act}}^{\texttt{MLP}} = (2Ep + p + 1)BSD \ . \tag{2.6}$$

LayerNorm, Residual Connections, and LM Head Then the last remaining components. The `LayerNorm` instances each have BSD inputs and there are two per transformer block, so $M_{\text{act}}^{\texttt{LayerNorm}} = 2pBSD$ for each layer, and there is an additional instance at the end of the architecture. There are two residual connections per block [18], and we have already cached one of the two inputs, so the per-block cost is $M_{\text{act}}^{\text{residual}} = 2pBSD$. Finally, the inputs to the language-model head are also `(B, S, D)`-shaped, so $M_{\text{act}}^{\text{LM head}} = pBSD$.

Total Activation Memory Summing up the contributions above, the total activation memory cost is

$$M_{\text{act}}^{\text{total}} = BSD\left(L\left(2p(E+5)+2\right)+2p\right) + LHBS^2\left(2p+1\right) \tag{2.7}$$

[17]As you can verify via `4 * torch.tensor([True]).element_size() == torch.tensor([1.]).element_size()`.

[18]These do not appear to be accounted for in [5]

Evaluating in common limits, we have:

$$\begin{aligned} M_{\rm act}^{\rm total}\Big|_{E=4,p=4} &= BS\left((74L+8)\,D + 9HLS\right) \\ M_{\rm act}^{\rm total}\Big|_{E=4,p=2} &= BS\left((38L+4)\,D + 5HLS\right) \end{aligned} \tag{2.8}$$

2.2 Model/Tensor Parallelism

In **Model Parallelism** or **Tensor Parallelism**, individual weight matrices are split across devices [11]. We consider the `MLP` and `CausalAttention` layers in turn. Assume T-way parallelism such that we split some hidden dimension into T-equal parts. The total number of workers is some $N \geq T$ which is evenly divisible by T. With $N > T$ workers, the workers are partitioned into N/T groups and collective communications will be required within, but not across, groups[19]. The members of a given group need to all process the same batch of data; data-parallelism must span different groups.

Essentials

The cost of large weights can be amortized by first sharding its output dimension, resulting in differing activations across group members. Later, the activations are brought back in sync via an `AllReduce`. Weights which act on the sharded-activations can also be sharded in their input dimension.

MLP It is straightforward to find the reasonable ways in which the weights can be partitioned. We suppress all indices apart from those of the hidden dimension for clarity.

The first matrix multiply $z_d W^0_{de}$ is naturally partitioned across the output index, which spans the expanded hidden dimension $e \in \{0, \ldots, EH-1\}$. This functions by splitting the weight matrix across its output indices across T devices: $W^0_{de} \longrightarrow W^{0(t)}_{de}$, where in the split weights $t \in \{0, \ldots, T-1\}$, and $e \in \left\{t\frac{EH}{T}, \ldots, (t+1)\frac{EH}{T}\right\}$. Note that each worker will have to store all components of the input z for their backward pass, and an `AllReduce` operation (see App. B) will be needed to collect gradient shards from other workers.

Let the partial outputs from the previous step again be $z^{(t)}_e$, which are `(B, S, E*H/T)`-shaped. The non-linearity ϕ acts element wise, and using the subsequent $z^{(t)}_e$ to compute the second matrix multiply requires a splitting the weights as in $W^1_{ed} \longrightarrow W^{1(t)}_{ed}$, such that the desired output is computed as in

$$z_e \cdot W^1_{ed} = z^{(t)}_e \cdot W^{1(t)}_{ed} \ , \tag{2.9}$$

sum over t implied, with each device computing one term in the sum. The sum is implemented by an `AllReduce` operation in practice. Note that no `AllReduce` is needed for the backwards pass, however.

[19] Ideally, all the workers in a group reside on the same node.

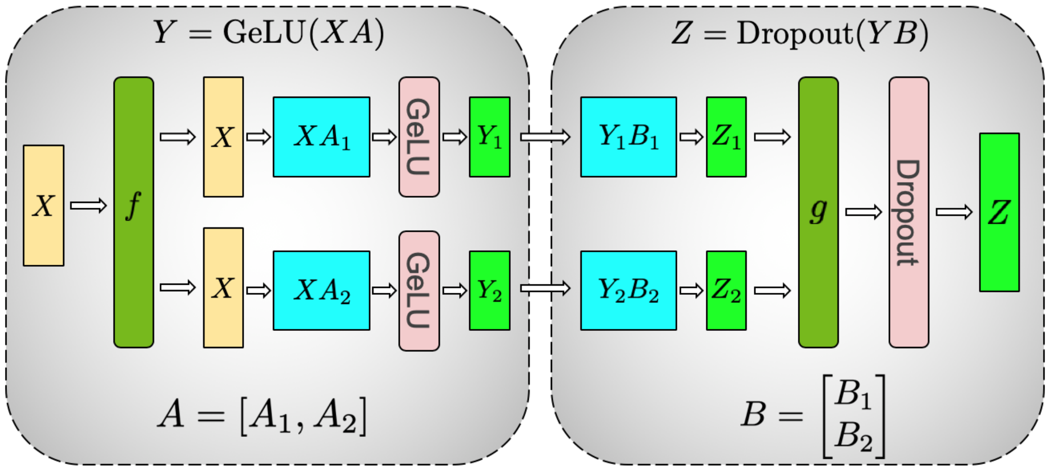

Figure 2. Tensor parallelism for the `MLP` layers. Graphic from [11]. The f/g operations are the collective identity/`AllReduce` operations in the forwards pass and the `AllReduce`/identity operations in the backwards pass.

Attention The computations for each the H individual attention heads, which result in the various re-weighted values t^h_{bsf} (1.4), can be partitioned arbitrarily across workers without incurring any collective communications costs. Each worker then holds some subset of these $h \in \{0, \ldots, H-1\}$ activations and the final output matrix multiply can be schematically broken up as in

$$\begin{aligned}&\text{Concat}\left(\left[t^0, \ldots, t^{H-1}\right]\right) \cdot O \\ &= \text{Concat}\left(\left[\text{Concat}\left(\left[t^0, \ldots, t^{\frac{H}{T}-1}\right]\right) \cdot O^{(0)}, \ldots, \text{Concat}\left(\left[t^{H-\frac{H}{T}}, \ldots, t^H\right]\right) \cdot O^{(T-1)}\right]\right) ,\end{aligned} \tag{2.10}$$

where matrix products and concatenation both occur along the hidden dimension. That is, each worker in a group has H/T different `(B, S, D/H)`-shaped activations t^h, which can be concatenated into a `(B, S, D/T)`-shaped tensor and multiplied into the `(D/T, D)`-shaped shard of O whose dimensions correspond to those in the just-concatenated tensor. Concatenating together each such result from every worker (via an `AllReduce`) gives the desired output. The backwards pass requires similar collective communications to the `MLP` case above.

Effects on Memory The per-worker memory savings come from the sharding of

2.3 Pipeline Parallelism

2.4 Sequence Parallelism

2.5 Case Study: Mixed-Precision GPT3

Let's run through the numbers for mixed-precision GPT3 with parameters:

$$L = 96 \ , D = 12288 \ , H = 96 \ , V = 50257 \ . \tag{2.11}$$

We are leaving the sequence-length unspecified, but the block-size (maximum sequence-length) is $K = 2048$

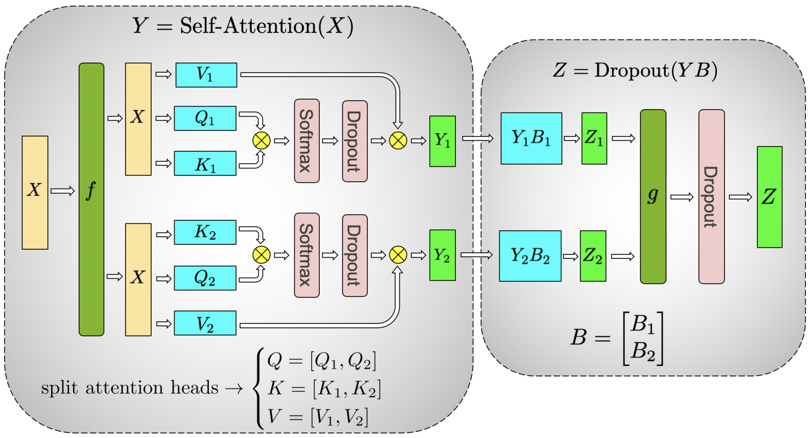
$Y = \text{Self-Attention}(X)$
X
f
X
V_1
Q_1
K_1
Softmax
Dropout
Y_1
X
K_2
Q_2
V_2
Softmax
Dropout
Y_2
split attention heads → $Q = [Q_1, Q_2]$, $K = [K_1, K_2]$, $V = [V_1, V_2]$
$Z = \text{Dropout}(YB)$
Y_1B_1
Z_1
Y_2B_2
Z_2
g
Dropout
Z
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$

Figure 3. Tensor parallelism for the `CausalAttention` layers. Graphic from [11].

Start by assuming no parallelism at all. The total non-activation memory is

$$M^{\texttt{GPT-3}}_{\text{non-act}} \approx 730 \text{ TiB} \tag{2.12}$$

which can be broken down further as

$$M^{\texttt{GPT-3}}_{\text{params}} \approx 80 \text{ TiB} \ , M^{\texttt{GPT-3}}_{\text{grads}} \approx 160 \text{ TiB} \ , M^{\texttt{GPT-3}}_{\text{optim}} \approx 490 \text{ TiB} \ . \tag{2.13}$$

The vocabulary only makes up $\approx .3\%$ of the total number of parameters, so the $\mathcal{O}(VD)$ term in (2.2) is negligible.

The activation memory is

$$M^{\texttt{GPT-3}}_{\text{act}} \approx 3 \times 10^{-2} BS \times \left(1 + \frac{S}{10^3}\right) \text{ TiB} \ . \tag{2.14}$$

Note attention matrices, which are responsible for $\mathcal{O}(S^2)$ term, will provide the dominant contribution to activation memory in the usual $S \gtrsim 10^3$ regime.

In the limit where we process the max block size ($S = K = 2048$), the ratio of activation to non-activation memory is

$$\left.\frac{M^{\texttt{GPT-3}}_{\text{act}}}{M^{\texttt{GPT-3}}_{\text{non-act}}}\right|_{S=2048} \approx .2B \ . \tag{2.15}$$

So, the activation memory is very significant for such models.

3 FLOPS

Part III
Inference

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[20])
- 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
- E: expansion factor for MLP layer (usually $E = 4$)

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[21] and write all ML code using standard `torch` syntax. To avoid needing to come up with new symbols in math 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

[20] In the absence of methods such as ALiBi [12] can be used to extend the sequence length at inference time.

[21] Written in a style conducive to latex, e.g. no type-hints and pegagogy prioritized.

summation symbols are entirely omitted. 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

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

Einstein notation also has implementations available for `torch`: see this blog post on `einsum` or the `einops` package.

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-index 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 Collective Communications

A quick refresher on common distributed communication primitives. Consider N workers with tensor data $x^{(n)}$ of some arbitrary shape `x.shape`, where n labels the worker and any indices on the data are suppressed. The $n = 0$ worker is arbitrarily denoted the *chief*. Then, the primitive operations are:

- `Broadcast`: all workers receive the chief's data, $x^{(0)}$.
- `Gather`: all workers communicate their data x_n to the chief, e.g. in a concatenated array $[x^0, x^1, \ldots, x^{N-1}]$.
- `Reduce`: data is `Gather`-ed to the chief, which then performs some operation (`sum`, `max`, `concatenate`, etc.) producing a new tensor x' on the chief worker.
- `AllGather`: `Gather` followed by `Broadcast`, such that all data $x^{(n)}$ is communicated to all workers.
- `AllReduce`: generalization of `Reduce` where all workers receive the same tensor x' produced by operating on the $x^{(n)}$. Equivalent to a `Reduce` followed by `Broadcast`, or a `ReduceScatter` followed by a `AllGather` (the more efficient choice).
- `ReduceScatter`: a reducing operation is applied to the $x^{(n)}$ to produce a x' of the same shape, but each worker only receives a slice $1/N$ of the result.

C 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]`. 2, 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. 2

[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]`. 2

[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, 11, 15

[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] N. Shazeer, "Fast transformer decoding: One write-head is all you need," `arXiv:1911.02150 [cs.NE]`. 4

[8] 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

[9] 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]`. 9

[10] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," `arXiv:1710.03740 [cs.AI]`. 10

[11] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," `arXiv:1909.08053 [cs.CL]`. 12, 13, 14

[12] 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`. 15