Decoder-Only Transformers

Notes on various aspects of Decoder-Only Transformers.

Contents

I Architecture 2

1 Decoder-Only Fundamentals 2
- 1.1 Embedding Layer and Positional Encodings 3
- 1.2 Layer Norm 4
- 1.3 Causal Attention 4
- 1.4 MLP 6
- 1.5 Language Model Head 7
- 1.6 All Together 7
- 1.7 The Loss Function 9

II Training 9

2 Memory 9
- 2.1 No Sharding 10
  - 2.1.1 Parameters, Gradients, Optimizer States, and Mixed Precision 10
  - 2.1.2 Gradients 11
  - 2.1.3 Activations 11
- 2.2 Tensor Parallelism 13
- 2.3 Sequence Parallelism 16
- 2.4 Pipeline Parallelism 16
- 2.5 Case Study: Mixed-Precision GPT3 16

3 Training FLOPs 18
- 3.1 No Recomputation 18

4 Scaling Laws 20

III Inference 20

5 Basics and Problems 20

6 The Minimum Needed and the kv-Cache 20

7 Memory and FLOPs 21

A Conventions and Notation 21

B Collective Communications 23

C Hardware 23

C.1 NVIDIA GPU Stats 23

D TODO 23

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.

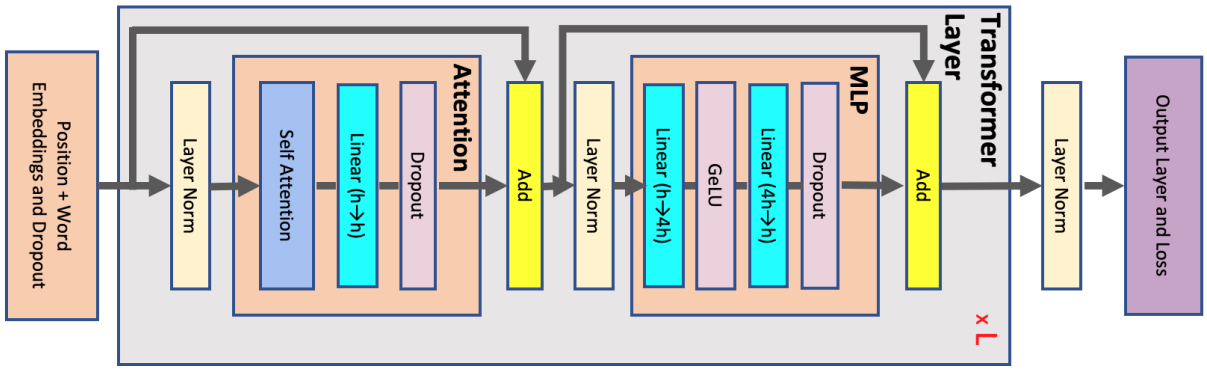

Figure 1. The full transformers architecture. Diagram taken from [5]

At a high level, decoder-only transformers take in a series of word-like objects, called tokens, and are trained to predict the next token in the sequence. An outline of the mechanics:

1. Raw text is **tokenized** and turned into a series of integers[1] whose values lie in `range(V)`,

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

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}.

   (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, i.e. x_{bs+1}, 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.

[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.

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.

1.3 Causal Attention

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

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

These are the **query, key, and value** tensors, respectively [7].

Using the above tensors, we will then build up an **attention map** $w^a_{bss'}$ 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 $s' \leq s$. This corresponds to the condition that $w^a_{bss'} = 0$ if $s' > s$.

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

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

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

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

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

[5] A 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.

forcing $w^a_{bss'd} = 0$ for $s > s'$. In other words, the causal mask ensures that a given tensor, say z_{bsd}, only has dependence on other tensors whose sequence index, say s', obeys $s' \leq s$. This is crucial for inference-time optimizations, in particular the use of the **kv-cache** in which key-value pairs do not need to be re-computed.

The $\sqrt{D/A}$ 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

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

and these A different `(B, S, D/A)`-shaped tensors are then concatenated along the f-direction to re-form a `(B, S, D)`-shaped tensor[9]

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

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:

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

where all of the dot-products are over feature dimensions (those of size D or D/A).

The final `Dropout` layer is often included as part of the `CausalAttention` block , as in the math above and in Fig. 1. Below is pedagogical[10] sample code for a `CausalAttention` layer which separates the last `Dropout` layer out. This will be convenient when we talk about sequence parallelism below in Sec. 2.3.

```
class CausalAttention(nn.Module):
    def __init__(
        self,
        attn_heads=A,
        hidden_dim=D,
        block_size=K,
        dropout=0.1,
    ):
        super().__init__()
        self.head_dim, remainder = divmod(hidden_dim, attn_heads)
```

[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.

[10] The code is written for clarity, not speed. An example optimization missing here: there is no need to form separate Q^a, K^a, V^a `Linear` layers, one large layer which is later chunked is more efficient

```
        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.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
        ]
        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.O(weighted_values)
        return z
```

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]. In code, where we again separate out the last `Dropout` layer as we did in in Sec. 1.3.

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

```python
class MLP(nn.Module):
    def __init__(
        self,
        hidden_dim=D,
        expansion_factor=E,
    ):
        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()
        self.layers = nn.Sequential(linear_1, gelu, linear_2)

    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.

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=A,
        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.attn = CausalAttention(attn_heads, hidden_dim, block_size, dropout)
        self.attn_drop = nn.Dropout(dropout)

        self.mlp_ln = nn.LayerNorm(hidden_dim)
        self.mlp = MLP(hidden_dim, expansion_factor)
        self.mlp_drop = nn.Dropout(dropout)

    def forward(self, inputs):
        z_attn = self.attn_ln(inputs)
        z_attn = self.attn(z_attn)
```

which corresponds to the schematic function

$$z \leftarrow z + \texttt{MLP}\left(\text{LayerNorm}\ \left(z + \texttt{CausalAttention}\left(\text{LayerNorm}\ \left(z\right)\right)\right)\right)\ , \tag{1.9}$$

indices suppressed.

And then the entire architecture:

```
class DecoderOnly(nn.Module):
    def __init__(
        self,
        attn_heads=A,
        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:

$$p(x_{b(s+1)} = v|x_{bs}, x_{b(s-1)}, \ldots, x_{b0}) = \text{Softmax}_v \; z_{bsv} \tag{1.10}$$

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

$$\begin{aligned}\mathcal{L}(x, z) &= \frac{1}{BK} \sum_{b,s} \ln p(x_{b(s+1)} = |x_{bs}, x_{b(s-1)}, \ldots, x_{b0}) \\ &= \frac{1}{BK} \sum_{b,s} \text{Softmax}_v \; z_{bsv}\big|_{v=y_{bs}} \; .\end{aligned} \tag{1.11}$$

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=A,
        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:]
    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[13].

The memory cost is much more than simply the cost of the model parameters. Significant factors include:

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

[13] A nice related blog post is here.

- 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: $\mathcal{O}(N_{\text{params}}) \sim \mathcal{O}(LD^2)$, only counting the dominant contributions from weight matrices. Activation memory scales with the latter, which for a `(B, S, D)`-shaped input gives $\mathcal{O}(LBSD)$ contributions from tensors which preserve the input shape, as well as $\mathcal{O}(LBHS^2)$ 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.

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 `torch.float32`/`torch.float16` ($p = 4$/ $p = 2$) mixed-precision, other, increasingly popular variants to exist, so we keep the precision variable where we can.

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

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

[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.

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

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

$$N_{\text{params}} \approx (4+2E)LD^2 \times \left(1 + \mathcal{O}\left(\frac{V}{DL}\right) + \mathcal{O}\left(\frac{1}{D}\right)\right) \ . \tag{2.2}$$

where the first term comes from the `TransformerBlock` weight matrices, the first omitted subleading correction term is the embedding matrix, and the last comes 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 those of (2.1). However, as we will see, the reduced cost of activation memory can offset these increased costs, and we get the added benefit of increased speed due to specialized hardware. The above also demonstrates why training is so much more expensive than inference.

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.

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

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

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

MLP Activations Eirst 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 Other Contributions 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, but their inputs do not require caching (since they're just additions whose derivatives are independent of inputs). Then, there are additional contributions from pushing the last layer's outputs through the language-model head and computing the loss function, but these do no scale with L and are ultimately $\sim \mathcal{O}\left(\frac{V}{DL}\right)$ suppressed, so we neglect them.

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

$$M_{\text{act}}^{\text{total}} \approx 2BDLS\left(p(E+4) + 1 + \mathcal{O}\left(\frac{V}{DL}\right)\right) + BHLS^2\,(2p+1) \ . \tag{2.7}$$

Evaluating in common limits, we have:

$$\begin{aligned} M_{\text{act}}^{\text{total}}\Big|_{E=4,p=4} &= BLS\,(66D + 15HS) \\ M_{\text{act}}^{\text{total}}\Big|_{E=4,p=2} &= BLS\,(34D + 5HS) \end{aligned} \tag{2.8}$$

When does mixed-precision reduce memory? (Answer: usually.) We saw in Sec. 2.1.1 that mixed precision *increases* the fixed costs of non-activation memory, but from the above we also see that it also *reduces* the activation memory and the saving increase with larger batch sizes and sequence lengths. It is straightforward to find where the tipping point is and for the simplified

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

$E = 4$, vanilla mixed-precision case with no parallelism the minimum batch size which leads to memory savings is

$$B_{\min} = \frac{6D^2}{8DS + AS^2} \ . \tag{2.9}$$

For typical architectures and training sequence lengths, the two terms in the denominator are about equal and so we can approximate the above as $B_{\min} \sim \frac{D}{S} \sim \mathcal{O}(5)$ for typical mid-to-large models in Summer 2023. The constant factors are important, but they turn out to help us: for GPT-2 and GPT-3

2.2 Tensor Parallelism

In **Tensor Parallelism**, sometimes also called **Model 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[18]. 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*A/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

$$z_e \cdot W^1_{ed} = z^{(t)}_e \cdot W^{1(t)}_{ed} \ , \tag{2.10}$$

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.

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

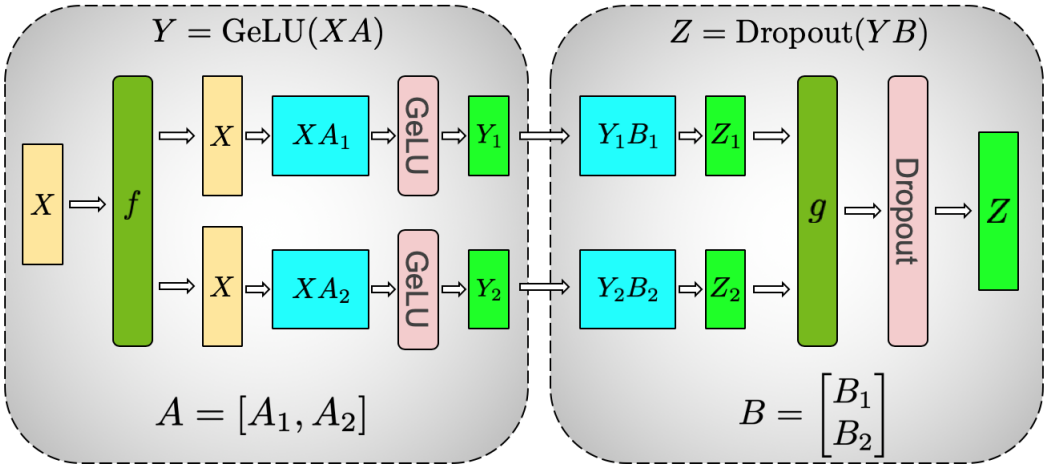

Figure 2. Tensor parallelism for the `MLP` layers. Graphic from [11]. The f/g operations are the collective identity/`AllReduce` operations in the forwards pass and the `AllReduce`/identity operations in the backwards pass.

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

$$\begin{aligned} &\mathrm{Concat}\left(\left[t^0, \ldots, t^{H-1}\right]\right) \cdot O \\ &= \mathrm{Concat}\left(\left[\mathrm{Concat}\left(\left[t^0, \ldots, t^{\frac{A}{T}-1}\right]\right) \cdot O^{(0)}, \ldots, \mathrm{Concat}\left(\left[t^{H-\frac{A}{T}}, \ldots, t^{A}\right]\right) \cdot O^{(T-1)}\right]\right) \ , \end{aligned} \tag{2.11}$$

where matrix products and concatenation both occur along the hidden dimension. That is, each worker in a group has A/T different `(B, S, D/A)`-shaped activations t^a, 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.

Embedding and LM Head Last, we can apply tensor parallelism to the language model head, which will also necessitate sharding the embedding layer, if the two share weights, as typical.

For the LM head, we shard the output dimension as should be now familiar, ending up with T different `(B, S, V/T)`-shaped tensors, one per group member. Rather than communicating these large tensors around and then computing the cross-entropy loss, it is more efficient to have each worker compute their own loss where possible and then communicate the scalar losses around[19].

For a weight-tied embedding layer, the former construction requires `AllReduce` in order for every worker to get the full continuous representation of the input.

[19] In more detail, given the gold-answers y_{bs} for the next-token-targets, a given worker can compute their contribution to the loss whenever their `(B, S, V/T)`-shaped output $z_{bsv'}$ contains the vocabulary dimension v_* specified by y_{bs}, otherwise those tensor components are ignored.

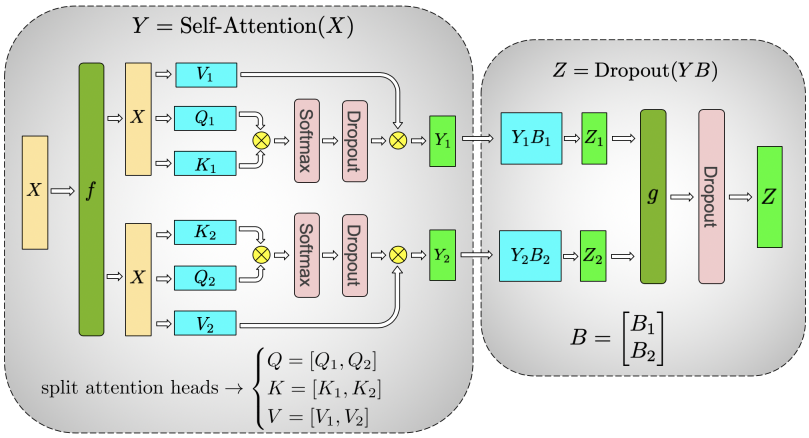

Figure 3. Tensor parallelism for the `CausalAttention` layers. Graphic from [11]. The f/g operators play the same role as in Fig. 2.

LayerNorm and Dropout `LayerNorm` instances are not sharded in pure tensor parallelism both because there is less gain in sharding them parameter-wise, but also sharding `LayerNorm` in particular would require additional cross-worker communication, which we wish to reduce as much as possible. `Dropout` layers are also not sharded in where possible in pure tensor parallelism, but sharding the post-attention `Dropout` layer is unavoidable. It is the goal of sequence parallelism is to shard these layers efficiently; see Sec. 2.3.

Effects on Memory The per-worker memory savings come from the sharding of the weights and the reduced activation memory from sharded intermediate representations.

The gradient and optimizer state memory cost is proportional to the number of parameters local to each worker (later we will also consider sharding these components to reduce redundantly-held information). The number of parameters per worker is reduced to

$$N_{\text{params}} \approx (4 + 2E)\frac{LD^2}{T} \ , \tag{2.12}$$

counting only the dominant contribution from weights which scale with L, since every weight is sharded. Since all non-activation contributions to training memory scale with N_{params}, this is a pure $1/T$ improvement.

The per-layer activation memory costs (2.5) and (2.6) are altered to:

$$\begin{aligned} M_{\text{act}}^{\texttt{Attention}} &= BS\left(\left(p + \frac{4p}{T} + 1\right)D + \left(\frac{2p+1}{T}\right)HS\right) \\ M_{\text{act}}^{\texttt{MLP}} &= \left(\frac{2Ep}{T} + p + 1\right)BSD \ . \end{aligned} \tag{2.13}$$

The derivation is similar to before. Adding in the (unchanged) contributions from `LayerNorm` instances, the total, leading order activation memory sums to

$$M_{\text{act}}^{\text{total}} \approx 2BDLS\left(p\left(2 + \frac{E+2}{T}\right) + 1\right) + BHLS^2\left(\frac{2p+1}{T}\right) \ . \tag{2.14}$$

Again, the terms which did not receive the $1/T$ enhancement correspond to activations from unsharded `LayerNorm` and `Dropout` instances and the $1/T$'s improvements can be enacted by layering sequence parallelism on top (Sec. 2.3).

2.3 Sequence Parallelism

In (2.14), not every factor is reduced by T. **Sequence Parallelism** fixes that by noting that the remaining contributions, which essentially come from `Dropout` and `LayerNorm`, can be parallelized in the sequence dimension (as can the residual connections).

The collective communications change a bit. If we shard the tensors across the sequence dimension before the first `LayerNorm`, then we want the following:

1. The sequence dimension must be restored for the `CausalAttention` layer
2. The sequence should be re-split along the sequence dimension for the next `LayerNorm` instance
3. The sequence dimension should be restored for the `MLP` layer [20]

The easiest way to achieve the above is the following.

1. If the tensor parallelization degree is T, we also use sequence parallelization degree T.
2. The outputs of the first `LayerNorm` are `AllGather`-ed to form the full-dimension inputs to the `CausalAttention` layer
3. The tensor-parallel `CausalAttention` layer functions much like in Fig. 3 *except* that we do not re-form the outputs to full-dimensionality. Instead, before the `Dropout` layer, we `ReduceScatter` them from being hidden-sharded to sequence-sharded and pass them through the subsequent `Dropout`/`LayerNorm` combination, similar to the first step
4. The now-sequence-sharded tensors are reformed with another `AllGather` to be the full-dimensionality inputs to the `MLP` layer whose final outputs are similarly `ReduceScatter`-ed to be sequence-sharded and are recombined with the residual stream

The above allows the `Dropout` mask and `LayerNorm` weights to be sharded T-ways, but if we save the full inputs to the `CausalAttention` and `MLP` layers for the backwards pass, their contributions to the activation memory are not reduced (the p-dependent terms in (2.13)). In [5], they solve this by only saving a $1/T$ shard of these inputs on each device during the forward pass and then performing an extra `AllGather` when needed during the backwards pass. Schematics can be sen in Fig. 4 and Fig. 5 below.

2.4 Pipeline Parallelism

2.5 Case Study: Mixed-Precision GPT3

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

$$L = 96 \ , \quad D = 12288 \ , \quad A = 96 \ , \quad V = 50257 \ . \tag{2.15}$$

[20] This doesn't seem like a hard-requirement, but it's what is done in [5].

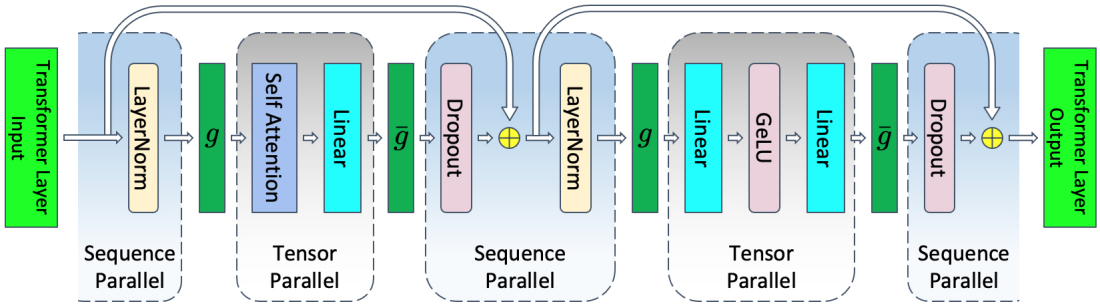

Figure 4. Interleaved sequence and tensor parallel sections. Graphic from [11].

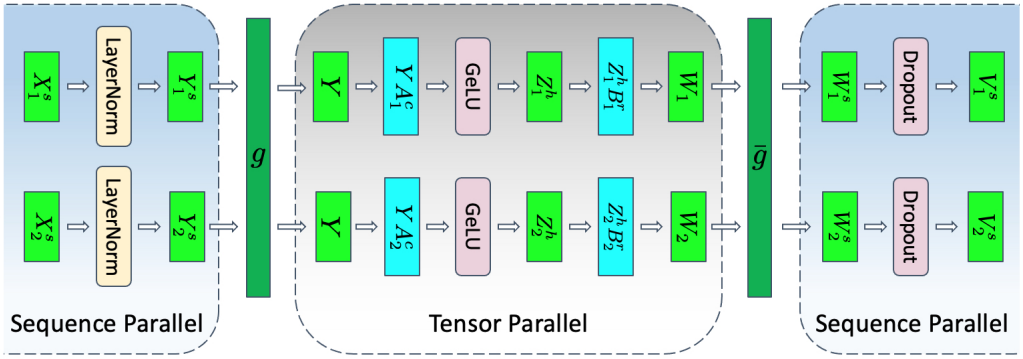

Figure 5. Detail of the sequence-tensor parallel transition for the `MLP` . Graphic from [11].

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

Start by assuming no parallelism at all. The total (not per-layer!) non-activation memory is

$$M_{\text{non-act}}^{\texttt{GPT-3}} \approx 1463 \text{ TiB} \tag{2.16}$$

which can be broken down further as

$$M_{\text{params}}^{\texttt{GPT-3}} \approx 162 \text{ TiB} , \quad M_{\text{grads}}^{\texttt{GPT-3}} \approx 325 \text{ TiB} , \quad M_{\text{optim}}^{\texttt{GPT-3}} \approx 975 \text{ TiB} . \tag{2.17}$$

The embedding matrix only makes up $\approx .3\%$ of the total number of parameters, justifying our neglect of its contribution in preceding expressions.

The activation memory is

$$M_{\text{act}}^{\texttt{GPT-3}} \approx 3 \times 10^{-2} BS \times \left(1 + \frac{S}{10^3}\right) \text{ TiB} . \tag{2.18}$$

Note that the attention matrices, which are responsible for $\mathcal{O}\left(S^2\right)$ 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

$$\frac{M_{\text{act}}^{\texttt{GPT-3}}}{M_{\text{non-act}}^{\texttt{GPT-3}}}\bigg|_{S=2048} \approx .2B . \tag{2.19}$$

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

Using tensor parallelism into the above with the maximal $T = 8$ which can be practically used, the savings are significant. The total memory is now

$$M_{\text{total}}^{\texttt{GPT-3}} \approx 187 \text{ TiB} + 10^{-2} BS + 5 \times 10^{-6} BS^2 \ . \tag{2.20}$$

3 Training FLOPs

The total number of floating point operations (FLOPs)[21] needed to process a given batch of data is effectively determined by the number of matrix multiplies needed.

Recall that a dot-product of the form $v \cdot M$ with $v \in \mathbb{R}^m$ and $M \in \mathbb{R}^{m,n}$ requires $m \times (2n-1) \approx 2mn$ FLOPs. For large language models, $m, n \sim \mathcal{O}\left(10^3\right)$, meaning that even expensive element-wise operations like `GeLU` acting on the same vector v pale in comparison by FLOPs count [22]. It is then a straightforward exercise in counting to estimate the FLOPs for a given architecture. The input tensor is of shape `(B, S, D)` throughout.

3.1 No Recomputation

Start with the case where there is no recomputation activations. These are the **model FLOPs** of [5], as compared to the **hardware FLOPs** which account for gradient checkpointing.

`CausalAttention`: Forwards The FLOPs costs:

- Generating the query, key, and value vectors: $6BSD^2$
- Attention scores: $2BDS^2$
- Re-weighting values: $2BDS^2$
- Final projection: $2BSD^2$

`MLP`: Forwards Passing a through the `MLP` layer, the FLOPs due to the first and second matrix-multiplies are equal, with total matrix-multiply requires $4BSED^2$.

Backwards Pass: Approximate The usual rule of thumb is to estimate the backwards pass as costing twice the flops as the forwards pass. This estimate comes from just counting the number of $\mathcal{O}\left(n^2\right)$ matrix-multiply-like operations and seeing that for every one matrix multiplication that was needed in the forward pass, we have roughly twice as many similar operations in the backwards pass.

The argument: consider a typical sub-computation in a neural network which is of the form $z' = \phi\left(W \cdot z\right)$ where z', a are intermediate representations z, z', ϕ is some non-linearity, and where

[21] 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 which has the lower case "s", FLOPs, but there's little consistency in general. We will use FLOPs and FLOP/s.

[22] Since their FLOPs counts only scales as $\sim \mathcal{O}\left(n\right)$ where the omitted constant may be relatively large, but still negligible when all dimensions are big.

the matrix multiply inside the activation function dominates the forwards-pass FLOPs count, as above. Then, in the backwards pass for this sub-computation, imagine we are handed the upstream derivative $\partial_{z'}\mathcal{L}$. In order to complete backpropagation, we need both to compute $\partial_W\mathcal{L}$ to update W and also $\partial_z\mathcal{L}$ to continue backpropagation to the next layer down. Each of these operations will cost about as many FLOPs as the forwards-pass, hence the estimated factor of two (but, as we will see, this is a very rough estimate).

Being more precise, let z be `(D0, ... , Dn, I)`-shaped and let W be `(I, J)`-shaped such that it acts on the last index of z, making z' `(D0, ... , Dn, J)`-shaped. Denoting $D = \prod_i D_i$ be the number of elements along the D_i directions for brevity, the forward-FLOPs cost of the sub-computation is therefore $2DIJ$.

Adding indices, the two derivatives we need are

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial W_{ij}} &= \frac{\partial\mathcal{L}}{\partial z'_{d_0\ldots d_n j}}\phi'\left((W\cdot z)_{d_0\ldots d_n i}\right) z_{d_0\ldots d_n i}\\ \frac{\partial\mathcal{L}}{\partial z_{d_0\ldots d_n i}} &= \frac{\partial\mathcal{L}}{\partial z'_{d_0\ldots d_n j}}\phi'\left((W\cdot z)_{d_0\ldots d_n i}\right) W_{ij}\ ,\end{aligned} \tag{3.1}$$

which have shapes `(I, J)` and `(D0, ..., Dn, I)`, respectively. On the right side, z and $W\cdot z$ are cached and the element-wise computation of $\phi'(W\cdot z)$ has negligible FLOPs count, as discussed above. The FLOPs count is instead dominated by the broadcast-multiplies, sums, and matrix-products.

Multiplying out the first two factors which are common to each derivative involves a broadcasted multiply of a `(D0, ... , Dn, J)`-shaped and a `(D0, ... , Dn, I, J)`-shaped tensor to produce another `(D0, ... , Dn, I, J)`-shaped result, which requires DIJ FLOPs. The process of multiplying this factor with either $z_{d_0\ldots d_n i}$ or W_{ij} and summing over the appropriate indices requires $2DIJ$ FLOPs for either operation, bringing the total FLOPs to $5DIJ$, which is (slightly more than) double the FLOPs for this same sub-computation in the forward-direction, hence the rough rule of thumb.

Backwards Pass: More Precise TODO

Total Model FLOPs The grand sum is then[23]:

$$F^{\text{model}}_{\text{total}} \approx 12BDLS\left(S + (2+E)\,D\right)\ . \tag{3.2}$$

We can also phrase the FLOPs in terms of the number of parameters as

$$F^{\text{model}}_{\text{total}}\big|_{T=1} = 6BSN_{\text{params}} \times \left(1 + \mathcal{O}\left(S/D\right)\right) \tag{3.3}$$

where we took the $T = 1, D \gg S$ limit for simplicity and we note that BS is the number of total tokens in the processed batches.

Based on the above, it is common to write down the rule of thumb[24]

$$(\text{Achievable FLOPs/sec}) \times (\text{Training time}) \approx 6N_{\text{params}} \times (\text{Total tokens})\ . \tag{3.4}$$

[23] With a large vocabulary, the cost of the final language model head matrix multiply can also be significant, but we have omitted its L-independent, $2BSDV$ contribution here.

[24] See this blog post or the link in Foot. 13.

The above is the total compute needed to train a model with the above data.

Lastly, the **model FLOPs utilization** (MFU) is an often-quoted quantity, which is just ratio of actually achieved FLOP/s (the rate, not the gross sum) to the maximum possible given the hardware

$$\text{MFU} = \frac{F_{\text{total}}^{\text{model}} \times \tau}{(\text{Num. GPUs}) \times (\text{Max. FLOP/s/GPU})} \tag{3.5}$$

where τ is the throughput rate in terms of tokens/second.

4 Scaling Laws

Part III
Inference

5 Basics and Problems

The essentials of decoder-only inference is that a given input sequence x_{bs} is turned into a probability distribution p_{bsv} over the vocabulary for what the next token might be. Text is then generated by sampling from p_{bsv} in some way, appending that value to x_{bs} to create a one-token-longer sequence, and then repeating until desired.

There are various problems that naive implementations of the above face:

- Repeated computation from processing the same tokens in the same order repeatedly, at least for some sub-slice of x_{bs}.
- Inherently sequential computation, rather than parallel
- Sub-optimal sampling strategies. Just choosing the most-probably token at each new step, does not guarantee the most-probable overall sequence, for instance.

6 The Minimum Needed and the kv-Cache

There are two separate stages during generation. First, an original, to-be-continued series of prompts x_{bs} can be processed in parallel to both generate the first prediction and populate any intermediate values we may want to cache for later. We follow [12] and call this the **prefill** stage. For this procedure, we require the entire x_{bs} tensor.

In the second, iterative part of generation (the **decode** stage) we have now appended one-or-more tokens to the sequence and we again want the next prediction, i.e. `z[:, -1, :]` for the last-layer outputs z_{bsd}. In this stage, we can avoid re-processing the entire x_{bs} tensor and get away with only processing the final, newly added token, *if* we are clever and cache old results.

The important pieces occur in the `CausalAttention` layer, as that's the only location in which the sequence index is not completely parallelized across operations. Referring back to Sec. 1.3,

given the input z_{bsd} of the `CausalAttention` layer, the re-weighted value vectors[25] $w^a_{bss'd}v^a_{bs'f}$ are the key objects which determine the next-token-prediction, which only depends on the $s = -1$ index values. Therefore, we can cut out many steps and minimum requirements are:

- Only the attention weights $w^a_{bss'd}$ with $s = -1$ are needed
- The only query values q^a_{bsd} needed to get the above are those with $s = -1$
- Every component of the key and value vectors k^a_{bsd}, v^a_{bsd} is needed, but because of the causal mask, all components except for the last in the sequence dimension ($s \neq -1$) are the same as they were in the last iteration, up to a shift by one position[26]

So, we are led to the concept of the **kv-cache** in which we cache old key and query vectors for generation. The cache represents a tradeoff: fewer FLOPs are needed for inference, but the memory costs are potentially enormous, since the size of the cache grows with batch size and sequence length:

$$M_{\text{kv}-\text{cache}} = 2pBSD \ . \tag{6.1}$$

This can easily be larger than the memory costs of the model parameter: $M^{\text{inference}}_{\text{params}} \sim pN_{\text{params}} \sim pLD^2$ (dropping $\mathcal{O}(1)$ factors), so that the cache takes up more memory when $BS \gtrsim LD$.

7 Memory and FLOPs

The analysis of the memory and FLOPs for inference is much simpler than its corresponding training counterparts (Sec. 3 and Sec. 2), since there is no optimizer state and activations can be thrown away immediately after use (neglecting the kv-cache).

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[27])
- S: input sequence length
- V: vocabulary size
- D: the hidden dimension size
- L: number of transformer layers

[25] Summed over s', but concatenating the different a values over the f dimension.

[26] That is, if we add a label t which indexes the iteration of generation we are on, then $z^{(t+1)}_{bsd} = z^{(t)}_{b(s-1)d}$ for every tensor in the network, as long as s isn't the final index.

[27] In the absence of methods such as ALiBi [13] can be used to extend the sequence length at inference time.

- A: number of attention heads
- P: pipeline parallel size
- T: tensor parallel size
- E: expansion factor for MLP layer (usually $E = 4$)
- τ: throughput rate in tokens/second
- p: the precision of the elements of a tensor in bytes

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

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

For instance, GPT-2 and GPT-3 [2, 3] have $V \sim \mathcal{O}\left(10^4\right)$, $S, L \sim \mathcal{O}\left(10^3\right)$, $L, A \sim \mathcal{O}\left(10^2\right)$.

As indicated above, we use zero-indexing. We also use `python` code throughout[28] 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 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.

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

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[29]).
- `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 Hardware

Basic information about relevant hardware considerations.

C.1 NVIDIA GPU Stats

The most important statistics for GPUs are their memory capacity, their maximum (theoretical) FLOP/s rate, their memory bandwidth, and their *math bandwidth* (which is the ratio of the last two quantities):

GPU	Memory	FLOP/s	Memory Bandwidth (B/s)	Math Bandwidth (FLOPs/B)
A100	40GiB/80GiB	312×10^{12}	1.5×10^{12}	208

D TODO

- Tokenizers
- Generation

[29] The former strategy scales linearly with the number of worker, while the latter strategy underlies "ring" `AllReduce` which is (nearly) independent of the number of workers. See this blog post for a nice visualization or [14] for a relevant paper.

- 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, 4

[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, 22

[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]. 22

[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, 16, 18, 21

[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]. 4

[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]. 5

[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]. 10

[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]. 11

[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]. 13, 14, 15, 17

[12] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," arXiv:2211.05102 [cs.LG]. 20

[13] 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. 21

[14] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing* (2009) . 23