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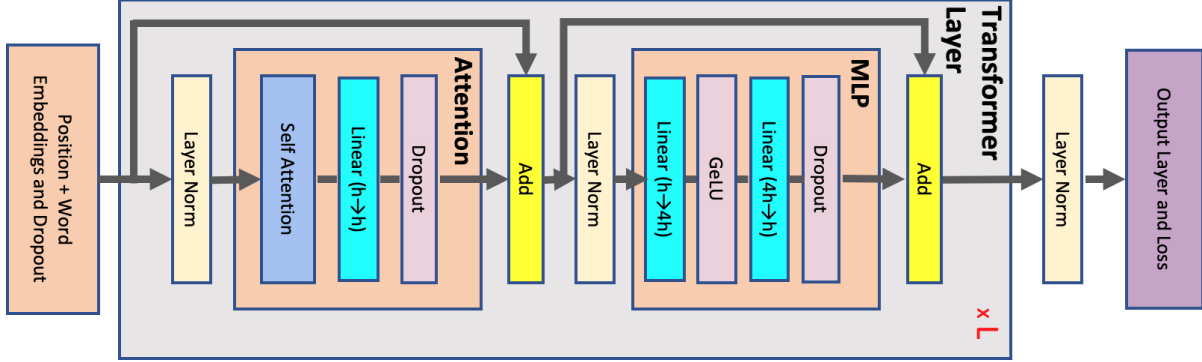
Figure 1. The full transformers architecture. Diagram taken from [5]

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)`, 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.)

---

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

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

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.

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.

We will continue discussing `LayerNorm` instances in what follows in order to adhere to the usual construction and to discuss methods like sequence-parallelism in their original form (see Sec. 3.3), but note: the γ_d, β_d parameters are *totally redundant* if `LayerNorm` is immediately followed by a `Linear` layer, as it is in the standard transformers architecture. The scaling the biases performed by these parameters can be equivalently performed by the weight matrix and bias (if included) in `Linear` layer.

---

[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

1.3 Causal Attention

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

$$q_{bsf}^a = z_{bsd} Q_{df}^a \ , \quad k_{bsf}^a = z_{bsd} K_{df}^a \ , \quad v_{bsf}^a = z_{bsd} V_{df}^a \tag{1.2}$$

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

Using the above tensors, we will then build up an **attention map** $w_{bss'}^a$ 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 $w_{bss'}^a = 0$ if $s' > s$.

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

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

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

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

forcing $w_{bss'd}^a = 0$ 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

$$t_{bsf}^a = \ \text{Drop} \left(w_{bdss'}^a \right) v_{bs'f}^a \tag{1.4}$$

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]

$$u_{bsd} = \text{Concat}_{fd}^a \left(t_{bsf}^a \right) \ . \tag{1.5}$$

---

[5]We assume throughout that A divides 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:

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

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

Below is pedagogical[10] sample code for such a `CausalAttention` layer[11]:

```
8    class CausalAttention(nn.Module):
9        def __init__(
10           self,
11           block_size=K,
12           dropout=0.1,
13           hidden_dim=D,
14           num_attn_heads=A,
15       ):
16           super().__init__()
17           self.block_size = block_size
18           self.dropout = dropout
19           self.hidden_dim = hidden_dim
20           self.num_attn_heads = num_attn_heads
21
22           self.head_dim, remainder = divmod(hidden_dim, num_attn_heads)
23           assert not remainder, "num_attn_heads must divide hidden_dim evenly"
24
25           self.Q = nn.ModuleList(
26               [nn.Linear(hidden_dim, self.head_dim) for _ in range(num_attn_heads)]
27           )
28           self.K = nn.ModuleList(
29               [nn.Linear(hidden_dim, self.head_dim) for _ in range(num_attn_heads)]
30           )
31           self.V = nn.ModuleList(
32               [nn.Linear(hidden_dim, self.head_dim) for _ in range(num_attn_heads)]
33           )
34           self.O = nn.Linear(hidden_dim, hidden_dim)
35
36           self.attn_dropout = nn.Dropout(dropout)
37           self.out_dropout = nn.Dropout(dropout)
38           self.register_buffer(
39               "causal_mask",
40               torch.tril(torch.ones(block_size, block_size)[None]),
41           )
42
```

---

[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

[11]When using sequence-parallelism, it will be more natural to separate out the final `Dropout` layer and combine it with the subsequent `LayerNorm`, as they are sharded together; see Sec. 3.3. The same is true for the `MLP` layer below.

```
43      def get_qkv(self, inputs):
44          queries = [q(inputs) for q in self.Q]
45          keys = [k(inputs) for k in self.K]
46          values = [v(inputs) for v in self.V]
47          return queries, keys, values
48
49      def get_attn_maps(self, queries, keys):
50          S = queries[0].shape[1]
51          norm = math.sqrt(self.head_dim)
52          non_causal_attn_scores = [(q @ k.transpose(-2, -1)) / norm for q, k in zip(queries, keys)]
53          causal_attn_scores = [
54              a.masked_fill(self.causal_mask[:, :S, :S] == 0, float("-inf"))
55              for a in non_causal_attn_scores
56          ]
57          attn_maps = [a.softmax(dim=-1) for a in causal_attn_scores]
58          return attn_maps
59
60      def forward(self, inputs):
61          queries, keys, values = self.get_qkv(inputs)
62          attn_maps = self.get_attn_maps(queries, keys)
63          weighted_values = torch.cat(
64              [self.attn_dropout(a) @ v for a, v in zip(attn_maps, values)], dim=-1
65          )
66          z = self.O(weighted_values)
67          z = self.out_dropout(z)
68          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[12]. In code, where we again separate out the last Dropout layer as we did in in Sec. 1.3.

```
6   class MLP(nn.Module):
7       def __init__(
8           self,
9           hidden_dim=D,
10          expansion_factor=E,
11          dropout=0.1,
12      ):
13          super().__init__()
14          self.hidden_dim = hidden_dim
15          self.expansion_factor = expansion_factor
16          self.dropout = dropout
```

---

[12]The GeLU non-linearity is common.

```
17
18              linear_1 = nn.Linear(hidden_dim, expansion_factor * hidden_dim)
19              linear_2 = nn.Linear(expansion_factor * hidden_dim, hidden_dim)
20              gelu = nn.GELU()
21              self.num_layers = nn.Sequential(linear_1, gelu, linear_2)
22              self.dropout = nn.Dropout(dropout)
23
24          def forward(self, inputs):
25              z = self.num_layers(inputs)
26              z = self.dropout(z)
27              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

```
8   class TransformerBlock(nn.Module):
9       def __init__(
10          self,
11          block_size=K,
12          dropout=0.1,
13          expansion_factor=E,
14          hidden_dim=D,
15          num_attn_heads=A,
16          num_layers=L,
17          vocab_size=V,
18      ):
19          super().__init__()
20          self.block_size = block_size
21          self.dropout = dropout
22          self.expansion_factor = expansion_factor
23          self.hidden_dim = hidden_dim
24          self.num_attn_heads = num_attn_heads
25          self.num_layers = num_layers
26          self.vocab_size = vocab_size
27
28          self.attn_ln = nn.LayerNorm(hidden_dim)
29          self.attn = CausalAttention(
30              block_size=block_size,
31              dropout=dropout,
32              hidden_dim=hidden_dim,
33              num_attn_heads=num_attn_heads,
34          )
35
```

```
36            self.mlp_ln = nn.LayerNorm(hidden_dim)
37            self.mlp = MLP(hidden_dim, expansion_factor, dropout)
38
39        def forward(self, inputs):
40            z_attn = self.attn_ln(inputs)
41            z_attn = self.attn(z_attn) + inputs
42
43            z_mlp = self.mlp_ln(z_attn)
44            z_mlp = self.mlp(z_mlp) + z_attn
45            return z_mlp
```

which corresponds to the schematic function

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

indices suppressed.

And then the entire architecture:

```
 7  class DecoderOnly(nn.Module):
 8      def __init__(
 9          self,
10          block_size=K,
11          dropout=0.1,
12          expansion_factor=E,
13          hidden_dim=D,
14          num_attn_heads=A,
15          num_layers=L,
16          vocab_size=V,
17      ):
18          super().__init__()
19          self.block_size = block_size
20          self.dropout = dropout
21          self.expansion_factor = expansion_factor
22          self.hidden_dim = hidden_dim
23          self.num_attn_heads = num_attn_heads
24          self.num_layers = num_layers
25          self.vocab_size = vocab_size
26
27          self.embedding = nn.Embedding(vocab_size, hidden_dim)
28          self.pos_encoding = nn.Parameter(torch.randn(1, block_size, hidden_dim))
29          self.drop = nn.Dropout(dropout)
30          self.trans_blocks = nn.ModuleList(
31              [
32                  TransformerBlock(
33                      block_size=block_size,
34                      dropout=dropout,
35                      expansion_factor=expansion_factor,
36                      hidden_dim=hidden_dim,
37                      num_attn_heads=num_attn_heads,
38                      num_layers=num_layers,
39                      vocab_size=vocab_size,
40                  )
41                  for _ in range(num_layers)
42              ]
```

```
43              )
44              self.final_ln = nn.LayerNorm(hidden_dim)
45              self.lm_head = nn.Linear(hidden_dim, vocab_size, bias=False)
46              self.lm_head.weight = self.embedding.weight  # Weight tying.
47
48          def forward(self, inputs):
49              S = inputs.shape[1]
50              z = self.embedding(inputs) + self.pos_encoding[:, :S]
51              z = self.drop(z)
52              for block in self.trans_blocks:
53                  z = block(z)
54              z = self.final_ln(z)
55              z = self.lm_head(z)
56              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[13] 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

$$\mathcal{L} = -\frac{1}{BK} \sum_{b,s} \ln p(x_{b(s+1)} = | x_{bs}, x_{b(s-1)}, \ldots, x_{b0})$$

$$= \frac{-1}{BK} \sum_{b,s} \text{Softmax}_v \, z_{bsv} \big|_{v=y_{bs}} . \tag{1.11}$$

Note that the losses for all possible context lengths are included in the sum[14].

In `torch` code, the loss computation might look like the following (using fake data):

```
7   def test_loss():
8       model = DecoderOnly(
9           num_attn_heads=A,
10          block_size=K,
11          dropout=0.1,
12          expansion_factor=E,
13          hidden_dim=D,
14          num_layers=L,
15          vocab_size=V,
16      )
```

---

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

[14]In Natural Language Processing (NLP), the `perplexity` is often reported instead of the loss, which is just the exponential of the loss, a geometric-mean over the gold-answer probabilities: perplexity $= e^{\mathcal{L}} = \left(\prod_{b,s} p(x_{b(s+1)} = | x_{bs}, x_{b(s-1)}, \ldots, x_{b0}) \right)^{\frac{-1}{BK}}$.

```
17      tokens = torch.randint(model.vocab_size, size=(B, model.block_size + 1))
18      inputs, targets = tokens[:, :-1], tokens[:, 1:]
19      outputs = model(inputs)
20      outputs_flat, targets_flat = outputs.reshape(-1, outputs.shape[-1]), targets.reshape(-1)
21      loss = F.cross_entropy(outputs_flat, targets_flat)
22      assert loss
```

2   Architecture Variants

There are, of course, many variants on the basic architecture. Some particularly important ones are summarized here.

2.1   Multi-Query Attention

In [7], the A different key and value matrices are replaced by a single matrix each, while A different query-heads remain. The mechanisms are otherwise unchanged: where there were previously distinct key and value tensors used across different heads, we just use the same tensors everywhere[15].

The primary reason for multi-query attention is that it vastly reduces the size of the kv-cache (see Sec. 9) during inference time, decreasing the memory-burden of the cache by a factor of A. This strategy also reduces activation memory during training, but that is more of a side-effect.

2.2   Parallel `MLP` and `CausalAttention` Layers

Rather than first pass inputs into the `CausalAttention` layer of each block, and then pass those outputs on to `MLP` in series, GPT-J-6B instead processes the `LayerNorm` outputs in *parallel*. That is, instead of something like

$$z \leftarrow z + \texttt{MLP}\left(\texttt{LayerNorm}\left(z + \texttt{CausalAttention}\left(z\right)\right)\right) \tag{2.1}$$

we instead have[16]

$$z \leftarrow z + \texttt{MLP}\left(z\right) + \texttt{CausalAttention}\left(z\right) . \tag{2.2}$$

Note that a `LayerNorm` instance is also removed.

Part II
Training

3   Memory

In this section we summarize the train-time memory costs of Transformers under various training strategies[17].

---

[15]TODO: understand how this changes tensor-parallelism, since it naively introduces A-fold redundancy

[16]This alternative layer was also used in PaLM [9] where it was claimed that this formulation is $\sim 15\%$ faster due to the ability to fuse the `MLP` and `CausalAttention` matrix multiplies together (though this is not done in the GPT-J-6B repo above).

[17]A nice related blog post is here.

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[18] , 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 $\mathcal{O}\left(BDLS\right)$ contributions from tensors which preserve the input shape, as well as $\mathcal{O}\left(BHLS^2\right)$ factors from attention matrices.

3.1 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.

3.1.1 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`[19] 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

$$M_{\text{model}} = 4N_{\text{params}} , \quad M_{\text{optim}} = 8N_{\text{params}} \quad (\text{no mixed precision}, p = 4) \tag{3.1}$$

---

[18]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 [10] makes use of this strategy.

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

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

$$N_{\text{params}} \approx (4 + 2E)LD^2 \times \left(1 + \mathcal{O}\left(\frac{V}{DL}\right) + \mathcal{O}\left(\frac{1}{D}\right)\right) . \tag{3.2}$$

where the first term comes from the `TransformerBlock` weight matrices[20], the first omitted subleading correction term is the embedding matrix, and the last comes from biases, `LayerNorm` instances, and other negligible factors. The optimizer states cost double the model itself.

The situation is more complicated when mixed-precision is used [11]. 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

$$M_{\text{model}} = 2N_{\text{params}} , \quad M_{\text{optim}} = 12N_{\text{params}} \quad \text{(mixed precision)}. \tag{3.3}$$

The optimizer state is now six times the cost of the actual model used to process data and the costs of (3.3) are more than those of (3.1). 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.

3.1.2 Gradients

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

$$M_{\text{grad}} = 4N_{\text{params}} . \tag{3.4}$$

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.

3.1.3 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.

---

[20]So, in 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 BDS elements are first converted to $3BDS$ total query, key, value elements, and the query-key dot products produce ABS^2 more, which are softmaxed into ABS^2 normalized scores. The re-weighted inputs to the final linear layer also have BDS elements, bringing the running sum to $BS(5D + 2AS)$

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 [21]. Given this, the total memory cost from activations is

$$M_{\text{act}}^{\texttt{Attention}} = BLS\left((5p+1)D + (2p+1)AS\right) \ . \tag{3.5}$$

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 $(2E+1)BDS$ total elements thus far. Adding in the dropout mask, the total memory requirement across all `MLP` layers is:

$$M_{\text{act}}^{\texttt{MLP}} = (2Ep + p + 1)BDLS \ . \tag{3.6}$$

LayerNorm, Residual Connections, and Other Contributions Then the last remaining components. The `LayerNorm` instances each have BDS inputs and there are two per transformer block, so $M_{\text{act}}^{\texttt{LayerNorm}} = 2pBDLS$, 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 their 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

$$M_{\text{act}}^{\text{total}} \approx 2BDLS\left(p(E+4) + 1 + \mathcal{O}\left(\frac{V}{DL}\right)\right) + ABLS^2(2p+1) \ . \tag{3.7}$$

Evaluating in common limits, we have:

$$M_{\text{act}}^{\text{total}}\Big|_{E=4, p=4} = BLS(66D + 15AS)$$

$$M_{\text{act}}^{\text{total}}\Big|_{E=4, p=2} = BLS(34D + 5AS) \tag{3.8}$$

When does mixed-precision reduce memory? (Answer: usually.) We saw in Sec. 3.1.1 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. Specializing to the case

---

[21] 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[22], the minimum batch size which leads to memory savings is

$$B_{\min} = \frac{6D^2}{8DS + AS^2} \ . \tag{3.9}$$

Plugging in numbers for the typical $\mathcal{O}\left(40 \text{ GiB}\right)$ model in the Summer of 2023 gives $B_{\min} \sim \mathcal{O}\left(1\right)$, so mixed-precision is indeed an overall savings at such typical scales.

3.2 Tensor Parallelism

In **Tensor Parallelism**, sometimes also called **Model Parallelism**, individual weight matrices are split across devices [12]. 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[23]. 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 $z_d W_{de}^0$ 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: $W_{de}^0 \longrightarrow W_{de}^{0(t)}$, where in the split weights $t \in \{0, \ldots, T - 1\}$, and $e \in \{t\frac{EH}{T}, \ldots, (t+1)\frac{EH}{T}\}$. 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 $z_e^{(t)}$, which are (B, S, E*A/T)-shaped. The non-linearity ϕ acts element wise, and using the subsequent $z_e^{(t)}$ to compute the second matrix multiply requires a splitting the weights as in $W_{ed}^1 \longrightarrow W_{ed}^{1(t)}$, such that the desired output is computed as in

$$z_e \cdot W_{ed}^1 = z_e^{(t)} \cdot W_{ed}^{1(t)} \ , \tag{3.10}$$

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.

---

[22]With both tensor- and sequence-parallelism, the parallelism degree T actually drops out in the comparison (since both form of memory are decrease by $1/T$, so this restriction can be lifted.

[23]Ideally, all the workers in a group reside on the same node.
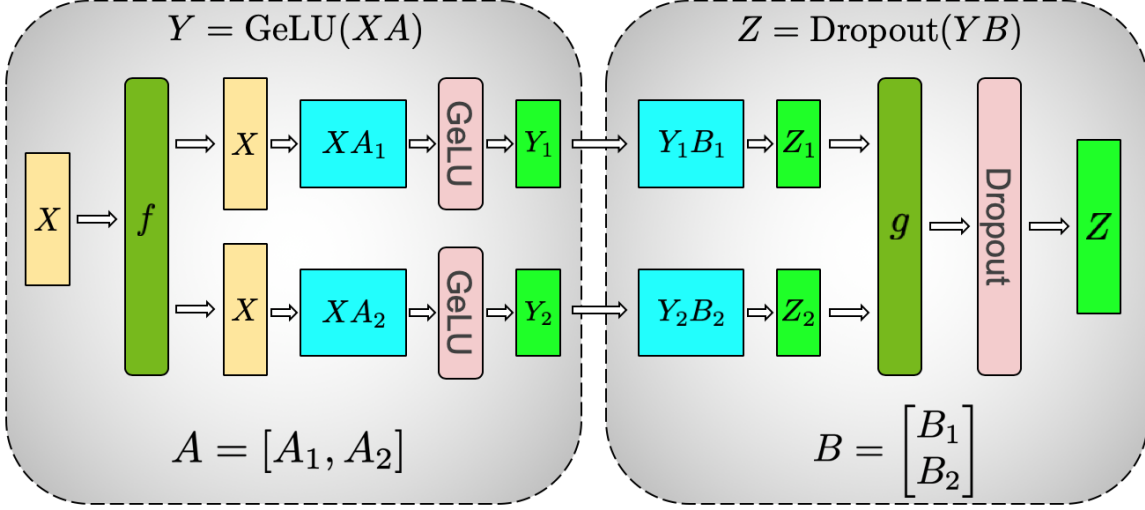
Figure 2. Tensor parallelism for the `MLP` layers. Graphic from [12]. The f/g operations are the collective identity/`AllReduce` operations in the forwards pass and the `AllReduce`/identity operations in the backwards pass.

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

$$\text{Concat}\left(\left[t^0, \ldots, t^{H-1}\right]\right) \cdot O$$
$$= \text{Concat}\left(\left[\text{Concat}\left(\left[t^0, \ldots, t^{\frac{A}{T}-1}\right]\right) \cdot O^{(0)}, \ldots, \text{Concat}\left(\left[t^{H-\frac{A}{T}}, \ldots, t^A\right]\right) \cdot O^{(T-1)}\right]\right), \quad (3.11)$$

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[24].

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.

---

[24]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.
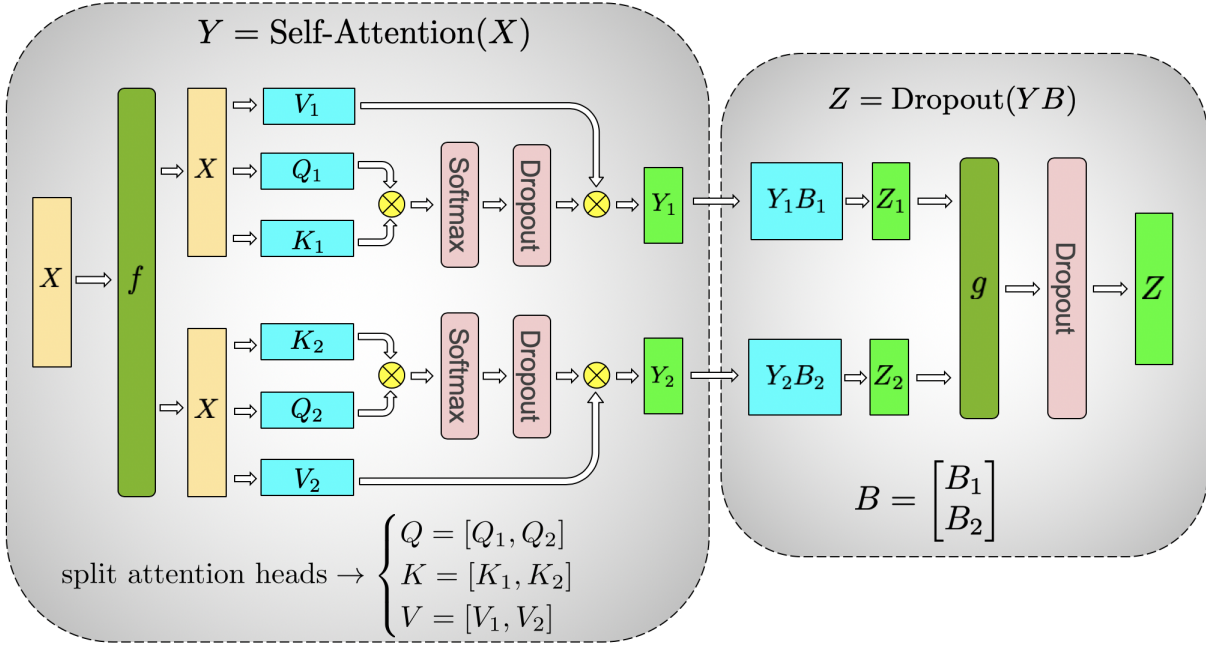
Figure 3. Tensor parallelism for the `CausalAttention` layers. Graphic from [12]. The f/g operators play the same role as in Fig. 2.

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. 3.3.

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

$$N_{\text{params}} \approx (4 + 2E)\frac{LD^2}{T} \ , \tag{3.12}$$

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 (3.5) and (3.6) are altered to:

$$M_{\text{act}}^{\texttt{Attention}} = BS\left(\left(p + \frac{4p}{T} + 1\right)D + \left(\frac{2p+1}{T}\right)AS\right)$$

17

$$M_{\text{act}}^{\texttt{MLP}} = \left(\frac{2Ep}{T} + p + 1 \right) BDS \ . \tag{3.13}$$

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

$$M_{\text{act}}^{\text{total}} \approx 2BDLS \left(p \left(2 + \frac{E+2}{T} \right) + 1 \right) + ABLS^2 \left(\frac{2p+1}{T} \right) \ . \tag{3.14}$$

Again, the terms which did not receive the $1/T$ enhancement correspond to activations from un-sharded `LayerNorm` and `Dropout` instances and the $1/T$'s improvements can be enacted by layering sequence parallelism on top (Sec. 3.3).

3.3 Sequence Parallelism

In (3.14), not every factor is reduced by T. **Sequence Parallelism** fixes that by noting that the remaining contributions, which essentially come from `Dropout` and `LayerNorm`[25], 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 [26]

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 (3.13)). In [5], they solve

---

[25]Recall, though, from Sec. 1.2 that the parameters in `LayerNorm` are completely redundant and can simply be removed without having any effect on the expressive capabilities of the architecture.

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

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. The activation memory is then reduced to:

$$M_{\text{act}}^{\text{total}} = \frac{2BDLS\left(p(E+4)+1\right)}{T} + \frac{ABLS^2\left(2p+1\right)}{T} + \mathcal{O}\left(BSV\right) . \tag{3.15}$$
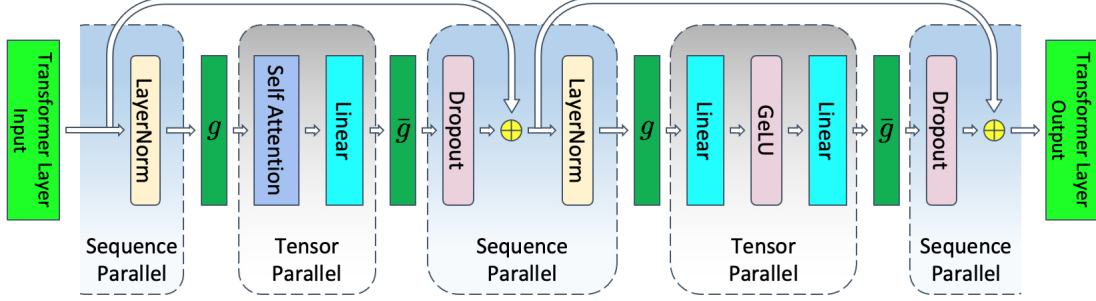
Figure 4. Interleaved sequence and tensor parallel sections. Graphic from [12].
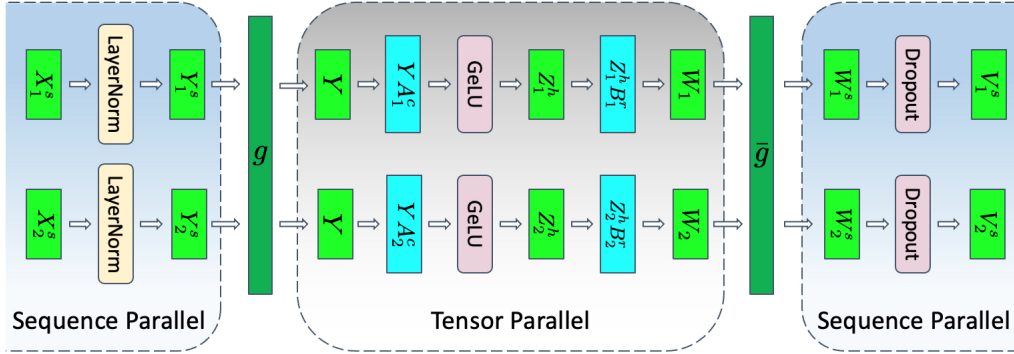
Figure 5. Detail of the sequence-tensor parallel transition for the `MLP` . Graphic from [12].

3.4   Pipeline Parallelism

TODO

3.5   Case Study: Mixed-Precision GPT3

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

$$L = 96 , \quad D = 12288 , \quad A = 96 , \quad V = 50257 . \tag{3.16}$$

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

$$M_{\text{non}-\text{act}}^{\texttt{GPT-3}} \approx 1463 \text{ TiB} \tag{3.17}$$

which can be broken down further as

$$M_{\text{params}}^{\texttt{GPT-3}} \approx 162 \text{ TiB} \ , \quad M_{\text{grads}}^{\texttt{GPT-3}} \approx 325 \text{ TiB} \ , \quad M_{\text{optim}}^{\texttt{GPT-3}} \approx 975 \text{ TiB} \ . \tag{3.18}$$

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

$$M_{\text{act}}^{\texttt{GPT-3}} \approx 3 \times 10^{-2} BS \times \left(1 + \frac{S}{10^3}\right) \text{ TiB} \ . \tag{3.19}$$

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

$$\left. \frac{M_{\text{act}}^{\texttt{GPT-3}}}{M_{\text{non-act}}^{\texttt{GPT-3}}} \right|_{S=2048} \approx .2B \ . \tag{3.20}$$

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

$$M_{\text{total}}^{\texttt{GPT-3}} \approx 187 \text{ TiB} + 10^{-2} BS + 5 \times 10^{-6} BS^2 \ . \tag{3.21}$$

4 Training FLOPs

The total number of floating point operations (FLOPs)[27] 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 $(2m - 1) \times n \approx 2mn$ 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 [28]. 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.

---

[27]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.

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

4.1 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 $\mathcal{O}(n^2)$ 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 $z' = \phi(W \cdot z)$ where z', a are intermediate representations z, z', ϕ is some non-linearity, and where 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

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{ij}} &= \frac{\partial \mathcal{L}}{\partial z'_{d_0...d_n j}} \phi' \left((W \cdot z)_{d_0...d_n i} \right) z_{d_0...d_n i} \\
\frac{\partial \mathcal{L}}{\partial z_{d_0...d_n i}} &= \frac{\partial \mathcal{L}}{\partial z'_{d_0...d_n j}} \phi' \left((W \cdot z)_{d_0...d_n i} \right) W_{ij} \ ,
\end{aligned}
\tag{4.1}
$$

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 $z_{d_0...d_n i}$ 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[29]:

$$
F_{\text{total}}^{\text{model}} \approx 12 BDLS \left(S + (2 + E) D \right) \ .
\tag{4.2}
$$

We can also phrase the FLOPs in terms of the number of parameters (3.12) as

$$
F_{\text{total}}^{\text{model}} \big|_{T=1} = 6 BS N_{\text{params}} \times \left(1 + \mathcal{O}\left(S/D \right) \right)
\tag{4.3}
$$

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[30]

$$
(\text{Achievable FLOPs/sec}) \times (\text{Training time}) \approx 6 N_{\text{params}} \times (\text{Total tokens}) \ .
\tag{4.4}
$$

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

Lastly, some common terminology. 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

---

[29]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, $2BDSV$ contribution here.

[30]See this blog post or the link in Foot. 17.

maximum possible given the hardware. If $\lambda_{\text{FLOP/s}}$ is the maximum FLOPs/s of the GPU (see App. C.1), then the quickest one could possibly push a batch of data through the model is

$$t_{\min} = \frac{F_{\text{total}}^{\text{model}}}{\lambda_{\text{FLOP/s}}} \tag{4.5}$$

where $F_{\text{total}}^{\text{model}}$ is the total FLOPs per GPU. If t_{iter} is the actual time it took to process the batch, then the ratio defines the MFU:

$$\text{MFU} = \frac{t_{\min}}{t_{\text{iter}}} \ . \tag{4.6}$$

The above implicitly assumed that we just do a straightforward forwards-and-backwards pass without any re-computation. In cases where we *do* recompute in order to save activation memory, we just update the $t_{\min}$ value above to reflect the actual amount of recomputation performed (raising $t_{\min}$) and compute the same ratio above. This defined the **hardware FLOPs utilization** (HFU).

5   Training Latency

The main factors which determine training times are the speed at which the necessary FLOPs can be completed and the communication times. We analyze both here.

5.1   FLOPs Latency

For any given operation in the model, the data needs to be first loaded in from GPU memory (DRAM) to the processor's SRAM, after which the computation begins. When training, the latter part is the limiting factor (see App. D), in which case the time per iteration is about

$$t_{FLOPs} = \frac{F_{\text{total}}^{\text{model}}}{\lambda_{\text{FLOP/s}}} \approx \frac{2BSN_{\text{params}}}{\lambda_{\text{FLOP/s}}} \ . \tag{5.1}$$

6   Scaling Laws

Part III

Inference

7   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.

8   Generation Strategies

A quick tour of generation strategies. A very readable blog post comparing strategies can be found here.

8.1   Greedy

The most obvious generation strategy is to take the final, (B, S, V)-shaped outputs z_{bsv} and just take the next token to be the most-probable one (for the final position in the sequence): `next_token = z[:, -1].argmax(dim=-1)`. A very minimal `generate` method is as below:

```
6    class DecoderOnlyGreedy(DecoderOnly):
7        def __init__(self, *args, **kwargs):
8            super().__init__(*args, **kwargs)
9
10       def generate(self, inputs, max_length):
11           """
12           Naive, minimal generation method. Assumes inputs are already tokenized. max_length can be
13           longer than the block_size, but only up to block_size tokens can ever be included in the
14           context.
15           """
16           self.eval()
17           outputs = inputs.clone()
18           while outputs.shape[1] < max_length:
19               context = outputs[:, -self.block_size :]
20               last_token_pred_logits = self(context)[:, -1]
21               most_probable_token = last_token_pred_logits.argmax(dim=-1)[:, None]
22               outputs = torch.cat([outputs, most_probable_token], dim=-1)
23           return outputs
```

There are various important, practical considerations which are ignored in the above implementation, including:

- Since we are taking the prediction from the last (-1-indexed) element in each sequence, it is crucial that all padding is *left*-padding, so that these final elements are meaningful.

- Models will signal the end of generation by outputting tokenizer-specific codes, and generation must respect these.

See the `generate` method from the `transformers` library for more fully-featured code (which, correspondingly, is not always easy to follow).

8.2 Simple Sampling: Temperature, Top-k, and Top-p

The next-most-obvious strategy is to choose the next token by drawing from the probability distribution defined by the z_{bsv}. There are various refinements of this idea.

A one-parameter generalization of this strategy introduces a (physics-motivated) **Temperature** which just adjusts the scale of the logits:

```
next_token = torch.multinomial((z[:, -1] / temp).softmax(dim=-1), num_samples=1)
```

assuming `z` are the final logits. Larger temperature yields a larger variance in the chosen tokens.

With temperature sampling, there is still a non-zero chance of choosing an extremely improbable token, which is undesirable if you do not trust the tails of the distribution. Two common truncation strategies which guard against this:

- **Top-k**: Only choose from the top-k most-probable examples (re-normalizing the probabilities across those k samples)

- **Top-p**: Only choose from the top-however-many most-probable examples whose probabilities sum to p (again re-normalizing probabilities). This is also sometimes called **nucleus sampling**.

8.3 Beam Search

Choosing, say, the most-probable next-token at each step is not guaranteed to yield the most probable *sequence* of tokens. So, **Beam Search** explores multiple sequences, using different branching strategies, and the probabilities of the various beam sequences can be compared at the end. Important note: generating the most-probable text is not necessarily equal to the most human-like text [13].

9 The Bare Minimum 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 [14] 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[31] $w^a_{bss'd} v^a_{bs'f}$ 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:

---

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

- Only the attention weights $w_{bss'd}^a$ with $s = -1$ are needed

- The only query values q_{bsd}^a needed to get the above are those with $s = -1$

- Every component of the key and value vectors k_{bsd}^a, v_{bsd}^a 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[32]

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:

$$M_{\text{kv-cache}} = 2pBDLS/T \ , \tag{9.1}$$

in the general case with tensor-parallelism. This can easily be larger than the memory costs of the model parameter: $M_{\text{params}}^{\text{inference}} \sim pN_{\text{params}} \sim pLD^2$ (dropping $\mathcal{O}(1)$ factors), so that the cache takes up more memory when $BS \gtrsim D$. Also, re-processing an entire input sequence every time also exhibits the usual $\sim S^2$ FLOPs dependence, but this is reduced to being *linear* in S when the kv-cache is used.

A very minimal implementation is below:

```
6   class CausalAttentionWithCache(CausalAttention):
7       def __init__(self, *args, **kwargs):
8           super().__init__(*args, **kwargs)
9           self.cached_keys = self.cached_values = None
10
11      def forward(self, inputs, use_cache=True):
12          """Forward method with optional cache. When use_cache == True, the output will have a
13          sequence length of one."""
14          if not use_cache:
15              return super().forward(inputs)
16          if self.cached_keys is None:
17              # If the cache is not yet initialized, we need all q, k, v values.
18              assert (
19                  self.cached_values is None
20              ), "If cached_keys is None, cached_values should be None, too"
21              queries, keys, values = self.get_qkv(inputs)
22          else:
23              # Otherwise, we only need q, k, v values for the last sequence position.
24              queries, new_keys, new_values = self.get_qkv(inputs[:, [-1]])
25              keys = [torch.cat([ck, nk], dim=1) for ck, nk in zip(self.cached_keys, new_keys)]
26              values = [torch.cat([cv, nv], dim=1) for cv, nv in zip(self.cached_values, new_values)]
27          # Update or initialize the cache.
28          self.cached_keys = [k[:, -self.block_size + 1 :] for k in keys]
29          self.cached_values = [v[:, -self.block_size + 1 :] for v in values]
30          last_queries = [q[:, [-1]] for q in queries]
31          attn_maps = self.get_attn_maps(last_queries, keys)
32          weighted_values = torch.cat(
```

---

[32]That is, if we add a label t which indexes the iteration of generation we are on, then $z_{bsd}^{(t+1)} = z_{b(s-1)d}^{(t)}$ for every tensor in the network, except for when $s = -1$, the last position.

```
33              [self.attn_dropout(a) @ v for a, v in zip(attn_maps, values)], dim=-1
34          )
35          z = self.O(weighted_values)
36          z = self.out_dropout(z)
37          return z
```

10   Basic Memory, FLOPs, Communication, and Latency

The essentials of inference-time math, much of it based on [15].

Naive Inference   Processing a single (B, S, D)-shaped tensor to generate a single next input costs the $2BSN_{\text{params}}$ FLOPs we found for the forwards-pass in Sec. 4. Memory costs just come from the parameters themselves: $M_{\text{infer}}^{\text{naive}} = pN_{\text{params}}$. Per the analysis of App. D, naive inference is generally compute-bound and so the per-token-latency is approximately[33] $2BSN_{\text{params}}/\lambda_{\text{FLOP/s}}$ where the FLOPs bandwidth in the denominator is again defined in App. D.

kv-Cache Inference   The FLOPs requirements for the hidden-dimension matrix multiplies during generation are $2BN_{\text{params}}$, since we are only processing a single token, per previous results. This is in addition to the up-front cost of $2BSN_{\text{params}}$ for the prefill. But, the memory requirements are raised to

$$M_{\text{infer}}^{\text{kv−cache}} = pN_{\text{params}} + 2pBDLS/T \; . \tag{10.1}$$

Inference now has a computational-intensity of

$$\frac{F_{\text{infer}}^{\text{kv−cache}}}{M_{\text{infer}}^{\text{kv−cache}}} \sim \tag{10.2}$$

now memory-bound (again, see App. D) and has per-token-latency of approximately $M_{\text{infer}}/\lambda_{\text{mem}}$, unless the batch-size is very large.

11   Case Study: Falcon-40B

Let's work through the details of the kv-cache for Falcon-40B[34] with $D = 8192$, $L = 60$, $S = 2048$. In half, $p = 2$ precision, the model weights just about fit on an 80GiB A100, but this leaves no room for the cache, so we parallelize T ways across T GPUs, assumed to be on the same node. The total memory costs are then

$$M_{\text{total}} \approx \frac{80\text{GiB} + 4\text{GiB} \times B}{T} \; . \tag{11.1}$$

This means that in order to hit the compute-bound threshold of $B \sim 200$ (see App. D) we need at least $T = 4$ way parallelism. Taking $T = 4$, and running at capacity with $B \sim 200$ so

---

[33]Assuming we do the naive thing here and generate the next token in a similarly naive way, shifting over the context window.

[34]Falcon actually uses multi-query attention, which changes the computations here, but we will pretend it does not in this section for simplicity.

that we are compute-bound, the per-iteration latency from computation alone is approximately $\frac{2BN_{\text{params}}}{\lambda_{\text{FLOP/s}}T}$ $\sim$13ms, i.e. we can give $\sim$200 customers about $\sim$75 tokens-per-second at this rate[35], if this were the only latency consideration.

A Conventions and Notation

We loosely follow the conventions of [5]. Common parameters:

- A: number of attention heads

- B: microbatch size

- D: the hidden dimension size

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

- F: FLOPs

- K: the block size (maximum sequence length[36])

- L: number of transformer layers

- N_{params}: number of model parameters held by a given device

- P: pipeline parallel size

- S: input sequence length

- T: tensor parallel size

- V: vocabulary size

- τ: various timescales

- p: the precision of the elements of a tensor in bytes

- λ: various rates, e.g. λ_{mem} is memory bandwidth

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

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

---

[35] Average human reading speed is about ~ 185 words/minute, or ~ 4tokens/sec.
[36] In the absence of methods such as ALiBi [16] can be used to extend the sequence length at inference time.

For instance, GPT-2 and GPT-3 [2, 3] have $V \sim \mathcal{O}\left(10^4\right)$, $S, L \sim \mathcal{O}\left(10^3\right)$, $L, A \sim \mathcal{O}\left(10^2\right)$. We will often assume also assume that[37] $S \lesssim D$ or the weaker[38] $BS \lesssim D$.

As indicated above, we use zero-indexing. We also use `python` code throughout[39] 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 $x_{bvd...}$ as

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

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 $[x^0, x^1, \ldots, x^{N-1}]$.

---

[37]This condition ensures that the $\mathcal{O}\left(S^2\right)$ FLOPs cost from self-attention is negligible compared to $\mathcal{O}\left(D^2\right)$ contributions from other matrix multiplies. It should be noted that in Summer 2023 we are steadily pushing into the regime where this condition does *not* hold.

[38]This condition ensures that the cost of reading the $\mathcal{O}\left(D^2\right)$ weights is more than the cost of reading in the $\mathcal{O}\left(BSD\right)$ entries of the intermediate representations.

[39]Written in a style conducive to latex, e.g. no type-hints and clarity prioritized over optimization.

- **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[40]).

- **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

Summary of some relevant NVIDIA GPU statistics:

GPU	Memory	$\lambda_{\text{FLOP/s}}$	λ_{mem}	λ_{math}	λ_{comms}
A100	40GiB	312 TFLOP/s	1.6 TiB/s	195 FLOPS/B	300 GiB/s
A100	80GiB	312 TFLOP/s	2.0 TiB/s	156 FLOPS/B	300 GiB/s
V100	32GiB	130 TFLOP/s	1.1 TiB/s	118 FLOPS/B	16 GiB/s

where

- $\lambda_{\text{FLOP/s}}$ is flops bandwidth (for `float16`/`bfloat16` multiply-accumulate ops)

- λ_{mem} is memory bandwidth

- $\lambda_{\text{math}} = \frac{\lambda_{\text{FLOP/s}}}{\lambda_{\text{mem}}}$ is math bandwidth

- λ_{comms} is one-way communication bandwidth

D   Compute-bound vs Memory-bound

If your matrix-multiplies are not sufficiently large on, you are wasting resources [18]. The relevant parameters which determine sufficiency are $\lambda_{\text{FLOP/s}}$ and λ_{mem}, the FLOPs and memory bandwidth, respectively. The ratio $\lambda_{\text{math}} \equiv \frac{\lambda_{\text{FLOP/s}}}{\lambda_{\text{mem}}}$ determines how many FLOPS you must perform for each byte loaded from memory; see App. C.1. If your computations have a FLOPs/B ratio which is larger than λ_{math}, then you are compute-bound (which is good, as you're maximizing compute), and otherwise you are memory(-bandwidth)-bound (which is bad, since your compute capabilities are

---

[40]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 [17] for a relevant paper.

idling). The FLOPs/B ratio of your computation is sometimes called the **compute intensity** or **arithmetic intensity**. When compute bound, a process takes time $\sim F/\lambda_{\text{FLOP/s}}$, while memory-bound processes take time[41] $\sim M/\lambda_{\text{mem}}$.

D.1 Matrix-Multiplications vs. Element-wise Operations

For instance, to multiply a (`B`, `S`, `D`)-shaped tensor z_{bsd} by a (`D`, `D`)-shaped weight-matrix $W_{dd'}$, $p\left(BDS + D^2\right)$ bytes must be transferred from DRAM to SRAM at a rate λ_{mem}, after which we perform $2BSD^2$ FLOPs, and write the (`B`, `S`, `D`)-shaped result back to DRAM again, for a ratio of

$$\frac{1}{p}\frac{BDS}{2BS + D} \quad (\text{FLOPs/B}) \ . \tag{D.1}$$

We want to compare this against λ_{math}, which from App. C.1 we take to be $\mathcal{O}\left(100 \text{ FLOPs/B}\right)$, and plugging in any realistic numbers, shows that such matrix-multiplies are essentially always compute-bound. Compare this to the case of some element-wise operation applied to the same z_{bsd} tensor whose FLOPs requirements are $\sim C \times BDS$ for some constant-factor $C \ll S, D$. Then, then FLOPS-to-bytes ratio is $\sim \frac{C}{p}$, which is *always* memory-bound for realistic values of C. The moral is to try and maximize the number of matrix-multiplies and remove as many element-wise operations that you can get away with.

D.2 Training vs. Inference

Finally, we note that the above has implications for the Transformers architecture as a whole, and in particular it highlights the difficulties in efficient inference. Under the assumptions of Sec. 4, $\sim \mathcal{O}\left(BSN_{\text{params}}\right)$ total FLOPs needed during training, while the number of bytes loaded from and written to memory are $\mathcal{O}\left(BDLS + N_{\text{params}}\right) \sim \mathcal{O}\left(\frac{BSN_{\text{params}}}{D} + N_{\text{params}}\right)$ which is $\mathcal{O}\left(N_{\text{params}}\right)$ for not-super-long sequence lengths. The arithmetic intensity is therefore $\mathcal{O}\left(BS\right)$ and so training is compute-bound in any usual scenario, even at small $B \sim \mathcal{O}\left(1\right)$ batch sizes (as long as individual operations in the network don't suffer from outlandish memory-boundedness). The problem during inference is that (if using the kv-cache; see Sec. 9) we only need to process a *single* token at a time and so $S \longrightarrow 1$ in the numerator in the preceding, while the denominator is also weighed down by the kv-cache in the attention layers.

In more detail, the `MLP` layers just process $S = 1$ length tensors, but are insensitive to the kv-cache, so their intensity comes from just setting $S = 1$ in the above,

$$\sim \frac{BD}{B + D} \ , \tag{D.2}$$

dropping $\mathcal{O}\left(1\right)$ factors now, while the attention layers have a ratio of the form

$$\sim \frac{BDS + BD^2}{BD + D^2 + BDS} \ , \tag{D.3}$$

---

[41]Note that the time is not additive, e.g. compute-bound tasks do not take time $\sim F/\lambda_{\text{FLOP/s}} + M/\lambda_{\text{mem}}$ because they are not sequential: compute and memory-communications can be concurrent.

where the last term in the denominator is due to the cache. Now at small $B \sim \mathcal{O}(1)$ batch sizes, both intensities reduce to $\mathcal{O}(B)$, which is insufficient to be compute-bound. In the large $B \gtrsim D/S$ limit, they at least become $\mathcal{O}(D)$ and $\mathcal{O}\left(1 + \frac{D}{S}\right)$, respectively, which may be enough to be compute-bound, but it's hard to even get into this regime. Note, the importance of the ratio D/S. The hidden dimension fixes the context length scale at which inference can never be compute-bound, in the absence of additional tricks not considered here[42].

E Cheat Sheet

Collecting all of the most fundamental equations, given to various degrees of accuracy.

Number of parameters per device:

$$N_{\text{params}} = \frac{(4 + 2E)LD^2}{T} + VD + \mathcal{O}(DL) \approx \frac{(4 + 2E)LD^2}{T} \,, \tag{E.1}$$

assuming no sharding of the embedding matrix.

Training Memory costs for fp16-like training:

$$M_{\text{model}} = p_{\text{model}} N_{\text{params}}$$
$$M_{\text{optim}} = (s_{\text{states}} + 1) \times p_{\text{master}} N_{\text{params}}$$
$$M_{\text{act}}^{\text{total}} = \frac{2BDLS\left(p(E+4)+1\right)}{T} + \frac{ABLS^2\left(2p+1\right)}{T} + \mathcal{O}(BSV) \tag{E.2}$$

where s_{optim} is the number of optimizer states, e.g. $s = 0$ for SGD and $s = 2$ for Adam. FLOPs total:

$$F_{\text{total}}^{\text{model}} \approx 12BDLS\left(S + (2+E)D\right). \tag{E.3}$$

F TODO

- Tokenizers
- Activations
- Flash attention
- BERT family
- Residual stream
- Scaling laws
- Cheat sheet

---

[42]One such trick: the multi-query attention of Sec. 2.1 improves everything a factor of A: the large batch regime is $B \gtrsim \frac{D}{AS}$ and the intensity ratio becomes $\mathcal{O}\left(1 + \frac{D}{AS}\right)$. An analysis equivalent to the one performed here can be found in the original paper [7].

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, 29

[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]`. 29

[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]`. 3, 13, 18, 21, 28

[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]`. 5, 11, 32

[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] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," `arXiv:2204.02311 [cs.CL]`. 11

[10] 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]`. 12

[11] 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]`. 13

[12] 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]`. 15, 16, 17, 19

[13] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," `arXiv:1904.09751 [cs.CL]`. 25

[14] 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]`. 25

[15] C. Chen, "Transformer inference arithmetic,". `https://kipp.ly/blog/transformer-inference-arithmetic/`. 27

[16] 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. 28

[17] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations,"
Journal of Parallel and Distributed Computing (2009) . 30

[18] H. He, "Making deep learning go brrrr from first principles,". https://horace.io/brrr_intro.html.
30