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.

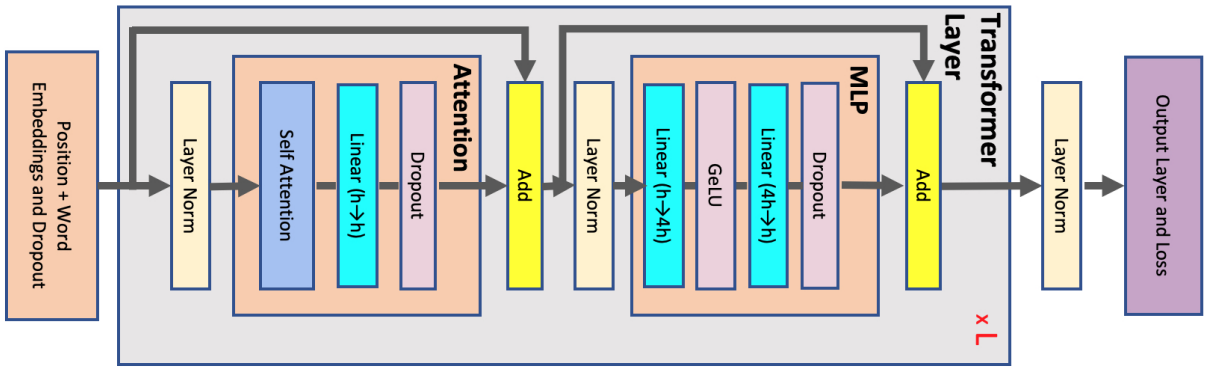

Figure 1. The full transformers architecture. Diagram taken from [5]

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. Equivalently, given some initial text, transformers can be used to give a prediction for the likelihood of any possible continuation of that text. 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:

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

(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. Specifically, removing the batch index for simplicity, a `Softmax` of y_{bsv} gives the conditional probability $p_{bsv} = P(t_{s+1} : t_s \dots t)$ for the indicated series of tokens. Because of the chain rule of probability, these individual probabilities can be combined to form the probability that any sequence of tokens follows a given initial seed[3].

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 [4] 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[5].

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

[3] In more detail, these probabilities are created by products: $P(t_{s+n} \dots t_{s+1}|t_s \dots t_0) = P(t_{s+n}|t_{s+n-1} \dots t_s \dots t_0) \times \dots \times P(t_{s+1}|t_s \dots t_0)$..

[4] 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.

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

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. Explicitly, we have the equivalence:

$$(x_{bsd}\gamma_d + \beta_d)\, W_{dd'} + b_{d'} = x_{bsd} W'_{dd'} + b'_{d'} \ , \quad W'_{dd'} \equiv \gamma_d W_{dd'} \ , \quad b'_{d'} \equiv b_{d'} + \beta_d W_{dd'} \ . \tag{1.2}$$

The scaling the biases performed by these parameters can be equivalently performed by the weight matrix and bias (if included) in `Linear` layer.

1.3 Causal Attention

The **causal attention** layer is the most complex layer. It features A triplets[6] of weight matrices[7] $Q^a_{df}, K^a_{df}, V^a_{df}$ where $a \in \{0, \ldots, H-1\}$ and $f \in \{0, \ldots, D/A\}$. From these, we form three different vectors:

$$q^a_{bsf} = z_{bsd} Q^a_{df} \ , \quad k^a_{bsf} = z_{bsd} K^a_{df} \ , \quad v^a_{bsf} = z_{bsd} V^a_{df} \tag{1.3}$$

These are the **query, key, and value** tensors, respectively [8].

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:

$$w^a_{bss'd} = \text{Softmax}_{s'}\left(m_{ss'} + q^a_{bsf} k^a_{bs'f} / \sqrt{D/A}\right) \ , \quad \text{s.t.} \quad \sum_{s'} w^a_{bdss'} = 1 \tag{1.4}$$

[6] We assume throughout that A divides the hidden dimension D evenly.

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

[8] 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.

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 $w^a_{bss'd} = 0$ 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 [9].

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

$$t^a_{bsf} = \text{Drop}\left(w^a_{bdss'}\right) v^a_{bs'f} \tag{1.5}$$

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[10]

$$u_{bsd} = \text{Concat}^a_{fd}\left(t^a_{bsf}\right) \ . \tag{1.6}$$

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

$$z_{bsd} = \text{Drop}\left(u_{bsd'} O_{d'd}\right) \ . \tag{1.7}$$

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

$$z \leftarrow \text{Drop}\left(\text{Concat}\left(\text{Drop}\left(\text{Softmax}\left(\frac{(z\cdot Q^a)\cdot(z\cdot K^a)}{\sqrt{D/A}}\right)\right)\cdot z\cdot V^a\right)\cdot O\right) \tag{1.8}$$

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

Below is pedagogical[11] sample code for such a `CausalAttention` layer[12]:

```
class CausalAttention(nn.Module):
    def __init__(
        self,
        block_size=K,
        dropout=0.1,
        hidden_dim=D,
```

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

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

[11] 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

[12] 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.

```
        num_attn_heads=A,
    ):
        super().__init__()
        self.block_size = block_size
        self.dropout = dropout
        self.hidden_dim = hidden_dim
        self.num_attn_heads = num_attn_heads

        self.head_dim, remainder = divmod(hidden_dim, num_attn_heads)
        assert not remainder, "num_attn_heads must divide hidden_dim evenly"

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

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

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

$$z_{bsd} \leftarrow \text{Drop}\left(\phi\left(z_{bsd'}W^0_{d'e}\right)W^1_{ed}\right) \tag{1.9}$$

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

```
class MLP(nn.Module):
    def __init__(
        self,
        hidden_dim=D,
        expansion_factor=E,
        dropout=0.1,
    ):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.expansion_factor = expansion_factor
        self.dropout = dropout

        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)
        self.dropout = nn.Dropout(dropout)

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

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

```python
class TransformerBlock(nn.Module):
    def __init__(
        self,
        block_size=K,
        dropout=0.1,
        expansion_factor=E,
        hidden_dim=D,
        num_attn_heads=A,
        num_layers=L,
        vocab_size=V,
    ):
        super().__init__()
        self.block_size = block_size
        self.dropout = dropout
        self.expansion_factor = expansion_factor
        self.hidden_dim = hidden_dim
        self.num_attn_heads = num_attn_heads
        self.num_layers = num_layers
        self.vocab_size = vocab_size

        self.attn_ln = nn.LayerNorm(hidden_dim)
        self.attn = CausalAttention(
            block_size=block_size,
            dropout=dropout,
            hidden_dim=hidden_dim,
            num_attn_heads=num_attn_heads,
        )

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

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

        z_mlp = self.mlp_ln(z_attn)
        z_mlp = self.mlp(z_mlp) + z_attn
        return z_mlp
```

which corresponds to the schematic function

$$z \leftarrow z + \texttt{MLP}\left(\text{LayerNorm}\ (z + \texttt{CausalAttention}\left(\text{LayerNorm}\ (z)\right))\right)\ , \tag{1.10}$$

indices suppressed.

And then the entire architecture:

```python
class DecoderOnly(nn.Module):
    def __init__(
        self,
        block_size=K,
        dropout=0.1,
        expansion_factor=E,
        hidden_dim=D,
        num_attn_heads=A,
```

```
        num_layers=L,
        vocab_size=V,
    ):
        super().__init__()
        self.block_size = block_size
        self.dropout = dropout
        self.expansion_factor = expansion_factor
        self.hidden_dim = hidden_dim
        self.num_attn_heads = num_attn_heads
        self.num_layers = num_layers
        self.vocab_size = vocab_size

        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(
                    block_size=block_size,
                    dropout=dropout,
                    expansion_factor=expansion_factor,
                    hidden_dim=hidden_dim,
                    num_attn_heads=num_attn_heads,
                    num_layers=num_layers,
                    vocab_size=vocab_size,
                )
                for _ in range(num_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[14] 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:

$$p(x_{b(s+1)} = v | x_{bs}, x_{b(s-1)}, \ldots, x_{b0}) = \underset{v}{\text{Softmax}}\, z_{bsv} \tag{1.11}$$

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

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

$$\begin{aligned}\mathcal{L} &= -\frac{1}{BK}\sum_{b,s}\ln p(x_{b(s+1)} = |x_{bs}, x_{b(s-1)}, \ldots, x_{b0}) \\ &= \frac{-1}{BK}\sum_{b,s}\text{Softmax}_v\ z_{bsv}\big|_{v=y_{bs}}\ . \end{aligned} \tag{1.12}$$

Note that the losses for all possible context lengths are included in the sum[15].

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

```
def test_loss():
    model = DecoderOnly(
        num_attn_heads=A,
        block_size=K,
        dropout=0.1,
        expansion_factor=E,
        hidden_dim=D,
        num_layers=L,
        vocab_size=V,
    )
    tokens = torch.randint(model.vocab_size, size=(B, model.block_size + 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)
    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[16].

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.

[15] 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 $= e^{\mathcal{L}} = \left(\prod_{b,s} p(x_{b(s+1)} = |x_{bs}, x_{b(s-1)}, \ldots, x_{b0})\right)^{\frac{-1}{BK}}$.

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

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[17]

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

Note that a `LayerNorm` instance is also removed.

2.3 RoPE Embeddings

A shortcoming of traditional embeddings $x_{bsd} \longrightarrow x_{bsd} + p_{sd}$ is that they do not generalize very well: a model trained on such embeddings with a maximum sequence length K will do very poorly when evaluated on longer sequences. RoPE (Rotary Position Embedding) and variants thereof can extend the viable context length by more clever mechanisms with stronger implicit biases.

RoPE and its variants can be motivated by a few natural conditions. Given the queries and keys for an input q_{sd}, k_{sd} (suppressing batch indices), the corresponding attention scores computation $a_{ss'}\left(q_s, k_{s'}\right)$ should reasonably satisfy the below:

1. The attention score should only depend on the position indices s, s' through their difference $s - s'$, i.e., through their relative distance to each other.

2. The score computation should still be efficient, i.e., based on matrix-mulitiplies.

3. The operation should preserve the scale of the intermediate representations and attention scores, in order to avoid issues with standard normalization.

These conditions suggest a very natural family of solutions: rotation of the queries by some fixed element of $SO(d)$ with a generator proportional to the position index, and rotation of keys by the conjugate element,

$$\begin{aligned} q_{sd} &\longrightarrow \left[e^{is\hat{n}\cdot T}\right]_{dd'} q_{sd'} \equiv R(s)_{dd'} q_{sd'} \\ k_{sd} &\longrightarrow \left[e^{-is\hat{n}\cdot T}\right]_{dd'} k_{sd'} \equiv R(s)^{\dagger}_{dd'} \ . \end{aligned} \tag{2.3}$$

Performing the above computation with a dense element of $SO(D)$ is infeasible, as it would require a new dense matrix-multiply by a unique $D \times D$ matrix at each sequence position[18]

[17] 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).

[18] For one, the $\mathcal{O}\left(SD^2\right)$ memory cost to store the matrices would be prohibitive. The FLOPs cost is only $2BSD^2$, the same as for other matrix multiplies, but because different matrices are needed at position, these FLOPs would be much more GPU memory-bandwidth intensive.

In the original RoPE paper, the rotation $\hat{n}$ was chosen such that the matrices are 2×2 block-diagonal with the entries of the form[19]

$$R(s)_{[d:d+2][d:d+2]} = \begin{pmatrix} \cos(s\theta_d) & -\sin(s\theta_d) \\ \sin(s\theta_d) & \cos(s\theta_d) \end{pmatrix} \tag{2.4}$$

where

$$\theta_d = 10^{-8d/D} \ . \tag{2.5}$$

Part II
Training

3 Memory

In this section we summarize the train-time memory costs of Transformers under various training strategies[20].

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

[19] If D isn't even, the vectors are padded by an extra zero.

[20] A nice related blog post is here.

[21] 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.

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`[22] 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

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

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[23], 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

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

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

[23] So, in the usual $E = 4$ case, the `MLP` layers are twice as costly as the `CausalAttention` layers.

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.

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

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

MLP Activations First we pass 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.

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

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:

$$\begin{aligned} M_{\text{act}}^{\text{total}}\Big|_{E=4,p=4} &= BLS(66D+15AS) \\ M_{\text{act}}^{\text{total}}\Big|_{E=4,p=2} &= BLS(34D+5AS) \end{aligned} \tag{3.8}$$

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 $E = 4$, vanilla mixed-precision case with no parallelism[25], the minimum batch size which leads to memory savings is

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

Plugging in numbers for the typical $\mathcal{O}(40\text{ GiB})$ model in the Summer of 2023 gives $B_{\text{min}} \sim \mathcal{O}(1)$, 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[26]. 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.

[25] 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.

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

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: $W^0_{de} \longrightarrow W^{0t}_{de}$, 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 $W^1_{ed} \longrightarrow W^{1t}_{ed}$ (dividing up the e dimension), such that the desired output is computed as in

$$z_e \cdot W^1_{ed} = z^t_e \cdot W^{1t}_{ed} \ , \tag{3.10}$$

sum over t implied, with each device computing one term in the sum over t. Every device is thus left holding a `(B, S, D)`-shaped tensor which must be `AllReduce`-d over the `D`-dimension. Note that no `AllReduce` is needed for the backwards pass, however.

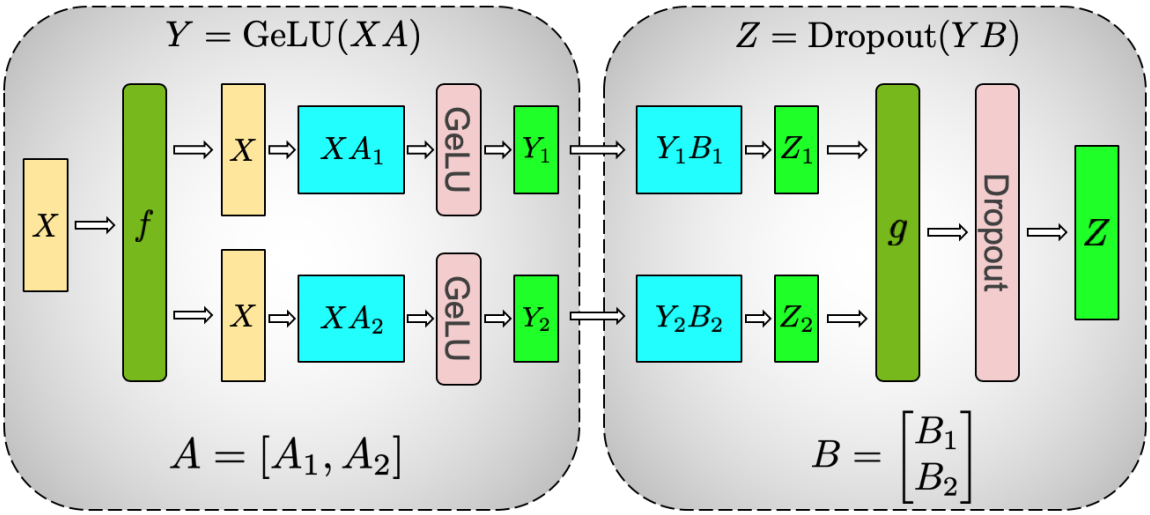

Figure 2. Tensor parallelism for the `MLP` layers. Graphic from [12]. The f/g operations are the collective identity/`AllReduce` operations in the forwards pass and the `AllReduce`/identity operations in the backwards pass.

Attention The computations for each the A individual attention heads, which result in the various re-weighted values t^a_{bsf} (1.5), 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

$$\begin{aligned} &\mathrm{Concat}\left(\left[t^0, \ldots, t^{H-1}\right]\right) \cdot O \\ &= \mathrm{Concat}\left(\left[\mathrm{Concat}\left(\left[t^0, \ldots, t^{\frac{A}{T}-1}\right]\right) \cdot O^{(0)}, \ldots, \mathrm{Concat}\left(\left[t^{H-\frac{A}{T}}, \ldots, t^{A}\right]\right) \cdot O^{(T-1)}\right]\right) \ , \end{aligned} \tag{3.11}$$

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, such that every worker is again left with a `(B, S, D)`-shaped shard which much be `AllReduce`-d, exactly as in the `MLP` case above, to get the final result. Hence, the collective communication costs for both layers are the same. The backwards pass requires similar collective communications to the `MLP` case above.

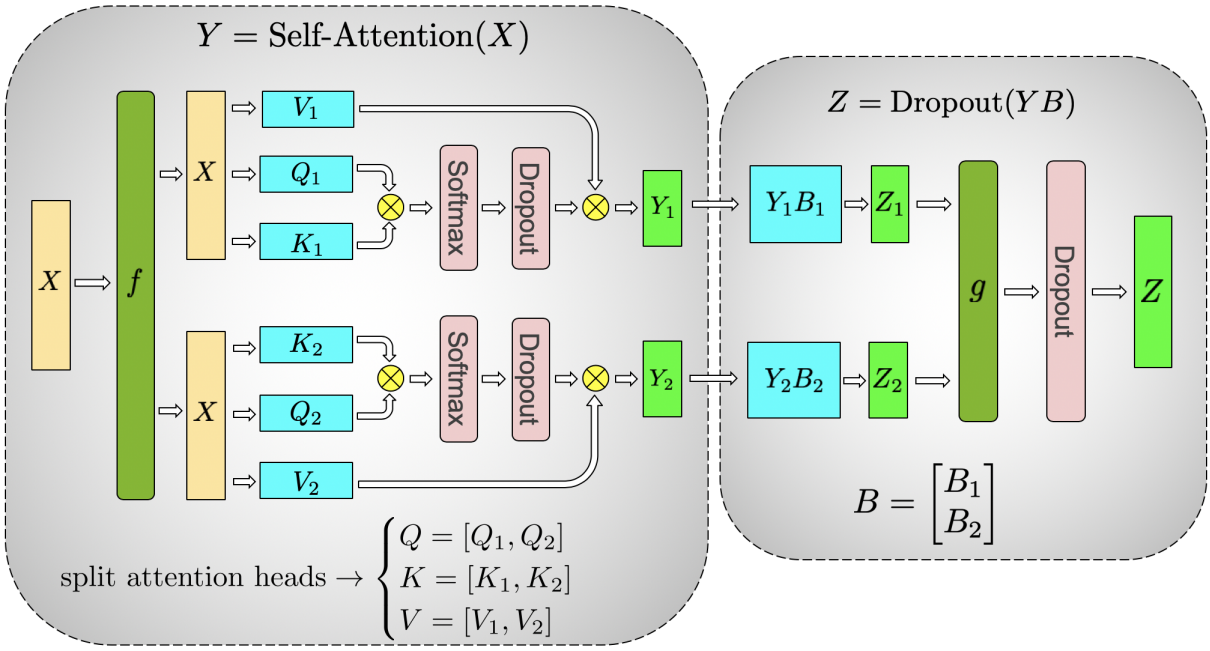

Figure 3. Tensor parallelism for the `CausalAttention` layers. Graphic from [12]. The f/g operators play the same role as in Fig. 2.

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[27].

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.

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

[27] 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.

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:

$$\begin{aligned} M_{\text{act}}^{\texttt{Attention}} &= BS\left(\left(p+\frac{4p}{T}+1\right)D+\left(\frac{2p+1}{T}\right)AS\right) \\ M_{\text{act}}^{\texttt{MLP}} &= \left(\frac{2Ep}{T}+p+1\right)BDS \ . \end{aligned} \tag{3.13}$$

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 unsharded `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`[28], 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

[28]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.

3. The sequence dimension should be restored for the `MLP` layer [29]

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 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:

$$M_{\text{act}}^{\text{total}} = \frac{2BDLS\left(p(E+4)+1\right)}{T} + \frac{ABLS^2\left(2p+1\right)}{T} + \mathcal{O}\left(BSV\right) \ . \tag{3.15}$$

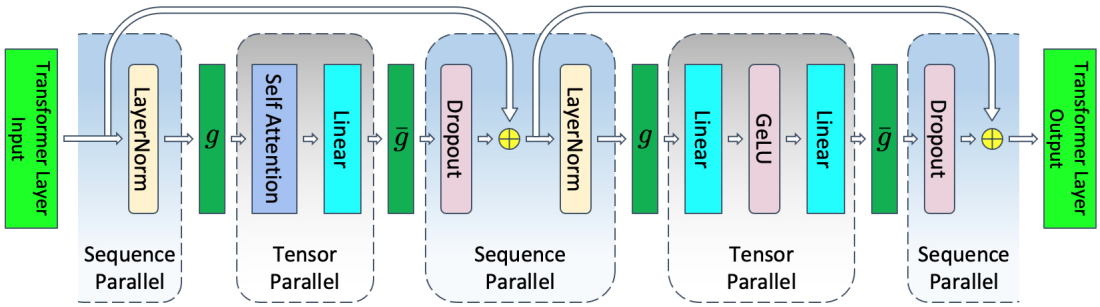

Figure 4. Interleaved sequence and tensor parallel sections. Graphic from [12].

3.4 Pipeline Parallelism

TODO

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

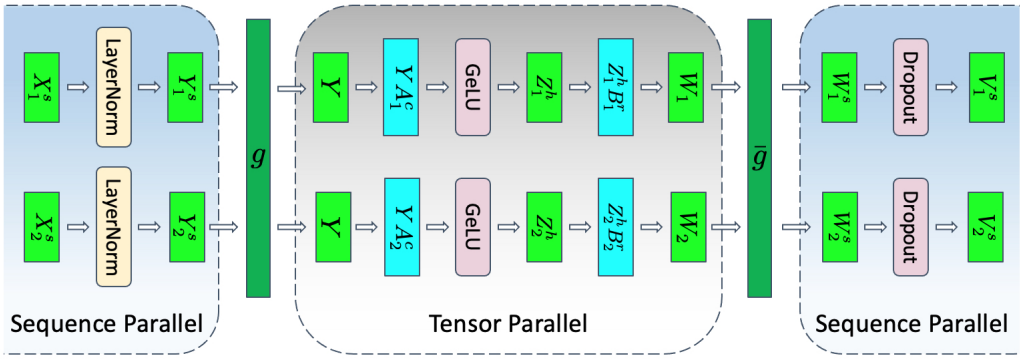

Figure 5. Detail of the sequence-tensor parallel transition for the `MLP` . Graphic from [12].

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

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

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

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

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)[30] 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 [31]. 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.

Essentials

The number of FLOPs to push a batch of B of sequence-length S examples through the forwards-pass of a decoder-only transformer is approximately $2BSN_{\text{params}}$ where the number of parameters accounts for any reductions due to tensor- and sequence-parallelism[a]. The backwards-pass costs about twice as much as the forwards-pass. This is true as long as $S \lesssim D$).

[a] A quick argument: a computation of the form $T_{a_0 \dots a_n j} = V_{a_0 \dots a_A i} M_{ij}$ requires $2A_0 \dots A_n IJ$ FLOPs where the capital letters represent the size of their similarly-index dimensions. Thus, the FLOPs essentially count the size of the matrix M (that is, IJ), up to a factor of 2 times all of the dimensions in V which weren't summed over. Therefore, passing a `(B, S, D)`-shaped tensor through the Transformer architecture would give $\sim 2BS\times$(sum of sizes of all weight-matrices) FLOPs, and that this last factor is also approximately the number of parameters in the model (since that count is dominated by weights). Thus, FLOPs $\approx 2BSN_{\text{params}}$. This is the correct as long as the self-attention FLOPs with $\mathcal{O}\left(S^2\right)$-dependence which we didn't account for here are actually negligible (true for $S \lesssim 10D$).

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 FLOPs $4BSED^2$.

[30] 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.

[31] 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.

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 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, J)`-shaped and let W be `(I, J)`-shaped such that it acts on the last index of z, making z' `(D0, ... , Dn, I)`-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

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

which have shapes `(I, J)` and `(D0, ..., Dn, J)`, respectively. On the right side, z and $W \cdot z$ are cached and the element-wise computation of $\phi'\left(W \cdot z\right)$ has negligible FLOPs count, as discussed above: its contribution is $\mathcal{O}\left(1/I\right)$ suppressed relative to the matrix-multiplies. The FLOPs count is instead dominated by the broadcast-multiplies, sums, and matrix-products.

The two derivatives in (4.1) each have the same first two factors in common, and it takes DI FLOPs to multiply out these two `(D0, ... , Dn, J)`-shaped tensors into another result with the same shape. This contribution is again $\mathcal{O}\left(1/I\right)$ suppressed and hence negligible. 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 $4DIJ$, which is double the FLOPs for this same sub-computation in the forward-direction, hence the rough rule of thumb[32].

Backwards Pass: More Precise **TODO**

Total Model FLOPs The grand sum is then[33]:

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

[32]Note also that the very first layer does not need to perform the second term in (4.1), since we do not need to backpropagate to the inputs, so the total backwards flops is more precisely $4DIJ(L-1) + 2DIJ$.

[33]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.

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

$$C^{\text{model}}\big|_{T=1} = 6BSN_{\text{params}} \times (1 + \mathcal{O}(S/D)) \tag{4.3}$$

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.

5 Training Training

Training is generally compute bound (see App. D) and based on the results of Sec. 4 the quickest one could possibly push a batch of data through the model is

$$t_{\text{min}} = \frac{C^{\text{model}}}{\lambda_{\text{FLOP/s}}} \ . \tag{5.1}$$

Expanding to the entire training run, then with perfect utilization training will take a time

$$t_{\text{total}} \approx \frac{6N_{\text{params}}N_{\text{tokens}}}{\lambda_{\text{FLOP/s}}} \ . \tag{5.2}$$

Adjust $\lambda_{\text{FLOP/s}}$ to the actual achievable FLOP/s in your setup to get a realistic estimate.

How many tokens should a model of size N_{params}? Scaling laws (Sec. 6) provide the best known answer, and the Summer 2023 best-guess is that we optimally have $N_{\text{tokens}} \approx 20N_{\text{params}}$. So that the above is

$$t_{\text{total}} \approx \frac{120N_{\text{params}}^2}{\lambda_{\text{FLOP/s}}} \ , \tag{5.3}$$

leading to quadratic growth in training time.

Note that the above is only correct if we are actually only spending C^{model} compute per iteration. This is not correct if we use gradient checkpointing and recomputation, in which case we alternatively spend true compute $C^{\text{hardware}} > C^{\text{model}}$, a distinction between **hardware FLOPs** and **model FLOPs**. Two corresponding efficiency measures are **model FLOPs utilization** (MFU) and **hardware FLOPs utilization** (HFU). If our iterations take actual time t_{iter}, then these are given by

$$\text{MFU} = \frac{t_{\text{iter}}}{t_{\text{min}}^{\text{model}}} \ , \quad \text{HFU} = \frac{t_{\text{iter}}}{t_{\text{min}}^{\text{hardware}}} \ , \tag{5.4}$$

where $t_{\text{min}}^{\text{model}}$ is (5.1) and $t_{\text{min}}^{\text{hardware}}$ is similar but using C^{hardware}.

6 Scaling Laws

Empirically-discovered scaling laws have driven the race towards larger and larger models.

Essentials

Decoder-only model performance improves predictably as a function of the model size, dataset size, and the total amount of compute. As of Summer 2023, there is little sign of hitting any kind of wall with respect to such scaling improvements.

The central parameters are:

- The number of non-embedding model parameters, as excising embedding params was found to generate cleaner scaling laws. Because our N_{params} has already been typically neglecting these parameters, we will just use this symbol in scaling laws and keep the above understanding implicit.[34] [13].
- C: total compute, often in units like PFLOP/s-days $\sim 10^{20}$ FLOPs
- N_{tokens}: dataset-size in tokens
- $\mathcal{L}$: cross-entropy loss in nats

The specific form of any given scaling law should also be understood to apply to a pretty narrowly defined training procedure, in which choices like the optimizer, learning-rate scheduler, hyperparameter search budget, vocabulary size, tokenization, etc. are often rigidly set. Changing different components of the training procedure is liable to create different scaling laws (though nice laws of some form are still expected to exist).

6.1 Original Scaling Laws

The first scaling-laws were reported in [13]. Their simplest form relates the value of the cross-entropy loss *at convergence* (and in nats), $\mathcal{L}$, to the number of non-embedding parameter, dataset size in token, and the amount of compute, *in the limit* where only one of this factors is bottlenecking the model[35]. The laws (in our notation):

- $\mathcal{L}(N_{\text{params}}) \approx \left(N^{\star}_{\text{params}}/N_{\text{params}}\right)^{\alpha_N}$, with $\alpha_N \approx 0.076$ and $N^{\star}_{\text{params}} \approx 8.8 \times 10^{13}$
- $\mathcal{L}(N_{\text{tokens}}) \approx (N^{\star}_{\text{tokens}}/N_{\text{tokens}})^{\alpha_T}$, with $\alpha_T \approx 0.095$ and $N^{\star}_{\text{tokens}} \approx 5.4 \times 10^{13}$
- $\mathcal{L}(C) \approx (C^{\star}/C)^{\alpha_C}$, with $\alpha_C \approx 0.050$ and $C^{\star} \approx 3.1 \times 10^{8}$ PFLOP/s-days, where the batch size was assumed to be chosen to be compute optimal per the criteria they outline

6.2 Chinchilla Scaling Laws

As of Summer 2023, the Chinchilla scaling laws in [14] are the de facto best scaling laws for guiding training. The central difference between [14] and [13] is that in the former they adjust their cosine learning-rate schedule to reflect the amount of planned training, while in the latter they do not[36].

[34] Presumably, the scaling laws are cleaner with these neglected because these params do not contribute directly to FLOPs, unlike most other parameters.

[35] Unclear to me how you know when this is the case?

[36] The learning-rate schedule consist of a linear warm-up stage from a very small η up to the largest value $\eta_{\max}$, after which the cosine bit kicks in: $\eta(s) = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \times \cos\left(\frac{\pi s}{2 s_{\max}}\right)$ with s the step number. In Fig. A1 of

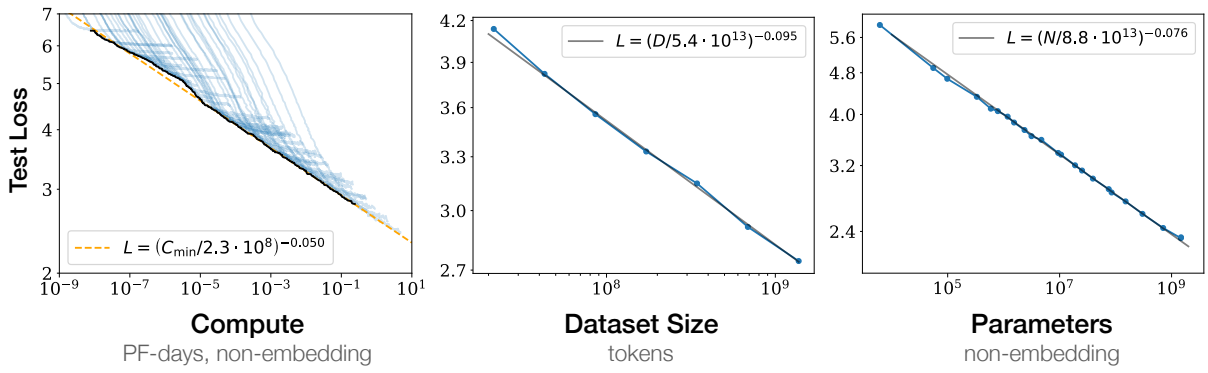

Figure 6. Original scaling laws from [13].

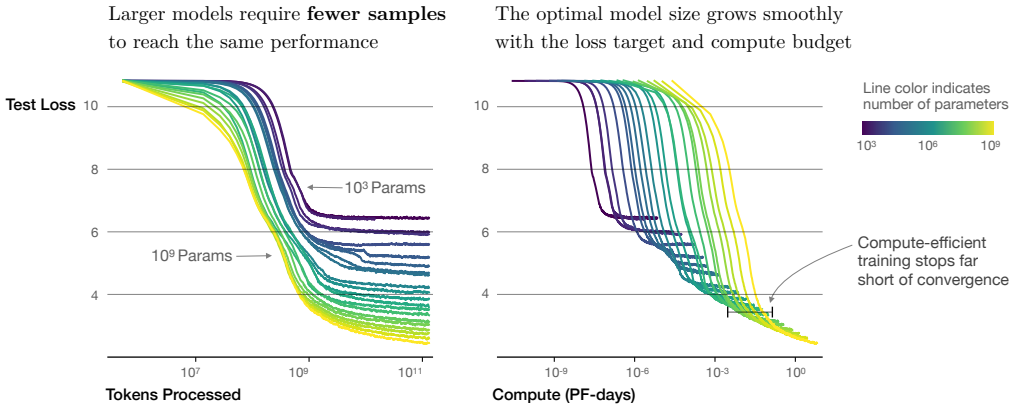

Figure 7. From [13]. Larger models are much more sample-efficient (faster).

Several different analyses are performed which all give very similar results. The outputs are the optimal values of $N_{\text{params}}, N_{\text{tokens}}$ given a compute budget C.

- They fix various buckets of model sizes and train for varying lengths. In their resulting loss-vs-FLOPs plot, they determine the model size which led to the best loss at each given FLOPs value, thereby generating and optimal model size vs compute relation.
- They fix various buckets of FLOPs budget and train models of different sizes with that budget, finding the optimal model size in each case. A line can then be fit to the optimal settings across FLOPs budgets in both the parameter-compute and tokens-compute planes.
- They perform a parametric fit to the loss[37]:

$$\mathcal{L}(N_{\text{params}}, N_{\text{tokens}}) = E + \frac{A}{N_{\text{params}}^{\alpha}} + \frac{B}{N_{\text{tokens}}^{\beta}} \ , \tag{6.1}$$

[14] they demonstrate that having the planned $s_{\max}$ duration of the scheduler be longer than the actual number of training steps is detrimental to training (they do not study the opposite regime), which is effectively what was done in [13]. Probably the more important general point is again that the precise form of these scaling laws depend on details of fairly arbitrary training procedure choices, such as the choice of learning-rate scheduler.

[37] In [14] they model the scaling of the test loss, while in [13] they use the training loss.

fit over a large range of parameter and token choices. The best-fit values are:

$$E = 1.69 \ , \quad A = 406.4 \ , \quad B = 410.7 \ , \quad \alpha = 0.34 \ , \quad \beta = 0.28 \ . \tag{6.2}$$

Using $C \approx 6 N_{\text{params}} N_{\text{tokens}}$, the above can be minimized at fixed compute either for number of parameter or the size of the dataset.

In all cases, the findings are that at optimality $N_{\text{params}} \sim N_{\text{tokens}} \sim C^{.5}$: both the parameter and tokens budget should be scaled in equal measure.

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):
```

```
        """
        Naive, minimal generation method. Assumes inputs are already tokenized. max_length can be
        longer than the block_size, but only up to block_size tokens can ever be included in the
        context.
        """
        self.eval()
        outputs = inputs.clone()
        while outputs.shape[1] < max_length:
            context = outputs[:, -self.block_size :]
            last_token_pred_logits = self(context)[:, -1]
            most_probable_token = last_token_pred_logits.argmax(dim=-1)[:, None]
            outputs = torch.cat([outputs, most_probable_token], dim=-1)
        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 [15].

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 [16] 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 (and accept a very reasonable approximation).

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[38] $w^a_{bss'd}v^a_{bs'f}$ 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[39]

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:

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

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

[39] This is where we need to accept an approximation. With an infinite context window, if we add a label t which indexes the iteration of generation we are on, then we would have that $z^{(t+1)}_{bsd} = z^{(t)}_{b(s-1)d}$ for every tensor in the network, except for when $s = -1$, the last position. The finiteness of the context window makes this statement slightly inaccurate because we can only ever keep K positions in context and the loss of the early tokens upon sliding the window over will slightly change the values in the residual stream.

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:

```python
class CausalAttentionWithCache(CausalAttention):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.cached_keys = self.cached_values = None

    def forward(self, inputs, use_cache=True):
        """Forward method with optional cache. When use_cache == True, the output will have a
        sequence length of one."""
        if not use_cache:
            return super().forward(inputs)
        if self.cached_keys is None:
            # If the cache is not yet initialized, we need all q, k, v values.
            assert (
                self.cached_values is None
            ), "If cached_keys is None, cached_values should be None, too"
            queries, keys, values = self.get_qkv(inputs)
        else:
            # Otherwise, we only need q, k, v values for the last sequence position.
            queries, new_keys, new_values = self.get_qkv(inputs[:, [-1]])
            keys = [torch.cat([ck, nk], dim=1) for ck, nk in zip(self.cached_keys, new_keys)]
            values = [torch.cat([cv, nv], dim=1) for cv, nv in zip(self.cached_values, new_values)]
        # Update or initialize the cache.
        self.cached_keys = [k[:, -self.block_size + 1 :] for k in keys]
        self.cached_values = [v[:, -self.block_size + 1 :] for v in values]
        last_queries = [q[:, [-1]] for q in queries]
        attn_maps = self.get_attn_maps(last_queries, keys)
        weighted_values = torch.cat(
            [self.attn_dropout(a) @ v for a, v in zip(attn_maps, values)], dim=-1
        )
        z = self.O(weighted_values)
        z = self.out_dropout(z)
        return z
```

10 Basic Memory, FLOPs, Communication, and Latency

The essentials of inference-time math, much of it based on [17].

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[40] $2BSN_{\text{params}}/\lambda_{\text{FLOP/s}}$

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

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

$$\frac{C_{\text{infer}}^{\text{kv-cache}}}{M_{\text{infer}}^{\text{kv-cache}}} \sim \frac{BD}{S} \ , \tag{10.2}$$

dropping $\mathcal{O}(1)$ factors, is 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.

Intra-Node Communication For T-way tensor parallelism, two `AllReduce`s are needed, one for each `MLP` and each `CausalAttention` layer, where each accelerator is sending $pBDS$ bytes of data (see Sec. 3.2). This requires a total of $4\,(T-1)\,pBDS/T \approx 4pBDS$ bytes to be transferred between workers in the tensor-parallel group (see Foot. 47), taking a total of $\sim 4pBDLS/\lambda_{\text{comms}}$ time for the model as a whole. For an A100 80GiB, `torch.float16` setup, this is $\sim BDS\times 10^{-11}$ sec

Latency

11 Case Study: Falcon-40B

Let's work through the details of the kv-cache for Falcon-40B[41] 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 that we are compute-bound, the per-iteration latency from computation alone is approximately $\frac{2BN_{\text{params}}}{\lambda_{\text{FLOP/s}}T}$ ~13ms, i.e. we can give ~200 customers about ~75 tokens-per-second at this rate[42], if this were the only latency consideration.

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

[42] Average human reading speed is about $\sim$ 185 words/minute, or $\sim$ 4tokens/sec.

A Conventions and Notation

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

- A: number of attention heads
- B: microbatch size
- C: compute (FLOPs)
- D: the hidden dimension size
- E: expansion factor for MLP layer (usually $E = 4$)
- K: the block size (maximum sequence length[43])
- L: number of transformer layers
- N_{params}: total number of model parameters
- P: pipeline parallel size
- S: input sequence length
- T: tensor parallel size
- V: vocabulary size
- t: 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

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

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[44] $S \lesssim D$ or the weaker[45] $BS \lesssim D$.

[43] In the absence of methods such as ALiBi [18] can be used to extend the sequence length at inference time.

[44] 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.

[45] 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.

As indicated above, we use zero-indexing. We also use `python` code throughout[46] 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.

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`: all data $x^{(n)}$ is communicated to all workers. Functionally equivalent to a `Gather` followed by `Broadcast`.

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

- `AllReduce`: all workers receive the same tensor x' produced by operating on the $x^{(n)}$ with `sum`, `mean`, etc. Functionally equivalent to a `Reduce` followed by `Broadcast`, or a `ReduceScatter` followed by a `AllGather` (the more efficient choice[47]).
- `ReduceScatter`: a reducing operation (`sum`, `mean`, etc.) 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. Much of the following is from the NVIDIA docs.

C.1 NVIDIA GPU Architecture

NVIDIA GPUs consist of some amount of relatively-slow off-chip DRAM memory[48], relatively-fast on-chip SRAM, and a number of **streaming multiprocessors** (SMs) which perform the parallel computations. Inside more-recent GPUs, the SMs carry both "CUDA cores" and "Tensor cores", where the latter are used for matrix-mulitiplies and the former for everything else.

A few numbers of primary importance:

- The rate at which data can be transferred from DRAM to SRAM (λ_{mem})
- The number of FLOP/s, which is more fundamentally computed by multiplying the number of SMs by the FLOPS/cycle of each SM for the specific operation under consideration (see the NVIDIA docs) by the clock rate: $N_{\text{SM}} \cdot \lambda_{\text{FLOPs/cycle}} \cdot \lambda_{\text{clock}}$

The terminology and structure of the memory hierarchy is also important to understand. Types of memory, from slowest to fastest:

- **Global** memory is the slow, but plentiful, off-chip DRAM. It is the type of memory typically used as kernel arguments
- **Constant** memory is read only and accessible by all threads in a given block. The size of arrays in constant memory must be known at compile time
- **Local Memory** is similarly slow to global memory, but more plentiful than register memory, and privately to individual threads and is allocated from within a kernel. When registers run out, local memory fills the gap
- **Shared** memory is shared between all threads in a given block. Shared memory is effectively a user-controlled cache. The size of arrays in shared memory must be known at compile time

[47] 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: if each worker carries data of size D which is to be `AllReduce`-d, a total of $\frac{2(N-1)D}{N}$ elements need to be passed around. See this blog post for a nice visualization or [19] for a relevant paper.

[48] This is the number usually reported when discussing a given GPU, e.g. 32GiB for the top-of-the-line A100

- **Registers** hold scalar values and small tensors whose values are known at compile time. They are local to each thread and they are plentiful since each thread needs its own set of registers: 65,536 = 2^{16} registers per SM an A100.

An excellent video overview of CUDA and NVIDIA GPU architecture which covers some of the above is here.

C.2 CUDA Programming Model

The CUDA programming model uses a hierarchy of concepts:

- **Threads** are the fundamental unit of execution which each run the same CUDA **Kernel**, or function, on different data inputs in parallel. Threads within the same block (below) may share resources, like memory, and may communicate with each other. Individual threads are indexed through the **threadIdx** variable, which has `threadIdx.{x, y, z}` attributes with `threadIdx.x` in `0, ..., blockDim.x -1` and similar.
- Threads are always launched in **Warps** which consist of 32 threads.
- Threads (and hence warps) are organized into 3D **blocks**. The size and indices of the blocks can be accessed through the `blockDim` and `blockIdx` variables, respectively, with `blockIdx.x` in `0, ..., gridDim.x - 1`. `blockDim.x *blockDim.y * blockDim.z` total threads run in a block.
- Blocks are organized into 3D **groups**. The size of the gird dimensions can be accessed through the `gridDim` variable, with similar attributes to the above.
  `gridDim.x * gridDim.y * gridDim.z` total blocks run in a grid.

The number of threads which can be launched in a given block is hardware limited; A100 80GiB GPUs can run up to 1024 threads in a SM at a time (32 blocks with 32 threads each), for instance. Hence, block and grid sizes need to be adjusted to match the problem size. There are also important memory access considerations here. The 1024 threads which can be launched can also read sequentially from memory and efficient usage implies that choosing the block size such that we are doing these reads as often as possible is ideal.

C.3 NVIDIA GPU Stats

Summary of some relevant NVIDIA GPU statistics:

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

A useful approximate conversion rate is that 1 TFLOP/s $\approx$ 100 PFLOP/day.

Important practical note: the $\lambda_{\text{FLOP/s}}$ numbers should be taken as aspirational. Out-of-the box, `torch.float16` matrix-multiplies in `torch` with well-chosen dimensions tops out around $\sim$ 250 FLOPS/s

D Compute-bound vs Memory-bound

If your matrix-multiplies are not sufficiently large on, you are wasting resources [20]. 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.3. 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 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[49] $\sim M/\lambda_{\text{mem}}$.

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

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

We want to compare this against λ_{math}, which from App. C.3 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

[49] 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.

written to memory are $\mathcal{O}(BDLS + N_{\text{params}}) \sim \mathcal{O}\left(\frac{BSN_{\text{params}}}{D} + N_{\text{params}}\right)$ which is $\mathcal{O}(N_{\text{params}})$ for not-super-long sequence lengths. The arithmetic intensity is therefore $\mathcal{O}(BS)$ and so training is compute-bound in any usual scenario, even at small $B \sim \mathcal{O}(1)$ 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 during generation, 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 $\mathcal{O}(1)$ factors now, while the attention layers have a ratio of the form

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

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[50].

D.3 Intra- and Inter-Node Communication

For intra-node communication, GPUs are connected by either PCIe or NVLink, generally.

- NVLink interconnects are continually updated and achieve speeds of $\lambda_{\text{comm}}^{\text{intra}} \sim 300$ GiB/s.

For inter-node communication, nodes are often connected by:

- InfiniBand apparently also achieves speeds $\lambda_{\text{comm}}^{\text{intra}} \sim 100$ GiB/s? Haven't found a clear reference. But in any case, the bandwidth is divided amongst the GPUs in the node, leading to a reduction by ~ 8.

E Batch Size, Compute, and Training Time

The amount of compute directly determines the training time, but not all ways of spending compute are equivalent. We follow the discussion in [21] which gives a rule of thumb for determining the optimal batch size which is sometimes used in practice. The basic point is that all of the optimization steps take the gradient $\mathbf{g}$ as an input, and since the gradient is the average over

[50] 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].

randomly selected datapoints, steps are more precise as the batch size increases (with diminishing returns, past a certain point, but the computational cost also rises with batch size, and a balance between the two concerns should be struck.

Consider vanilla SGD and study how the training loss changes with each step. We randomly sample B datapoints $x \in \mathcal{D}$ from the dataset through some i.i.d. process[51]. Each corresponding gradient $\mathbf{g}(x) = \partial_w \mathcal{L}(w, x)$ is itself a random variable whose average is the true gradient across the entire dataset $\bar{\mathbf{g}}$ and we take the variance to be

$$\mathrm{Var}[\mathbf{g}(x), \mathbf{g}(x')] = \Sigma \tag{E.1}$$

for some matrix Σ with (supressed) indices spanning the space of model weights. Taking instead the mean of a sum of such estimates, $\mathbf{g}_B \equiv \frac{1}{B}\sum_{x\in\mathcal{B}} \mathbf{g}(x)$, the mean stays the same, but the variance reduces in the usual way: $\mathrm{Var}[\mathbf{g}_B(x), \mathbf{g}_B(x')] = \Sigma/B$.

Study the mean loss across the entire dataset: $\mathcal{L}(w) = \langle \mathcal{L}(w, x)\rangle$. Using SGD we take a step $w \longrightarrow w - \eta \mathbf{g}_B$ and change the loss as

$$\mathcal{L}(w - \eta \mathbf{g}_B) = \mathcal{L}(w) - \eta \bar{\mathbf{g}} \cdot \mathbf{g}_B + \frac{1}{2}\mathbf{g}_B \cdot H \cdot \mathbf{g}_B + \mathcal{O}\left(\mathbf{g}_B^3\right) \ , \tag{E.2}$$

where H is the true hessian of the loss over the entire dataset at this value of the weights. Taking the expectation value and minimizing the results over η gives the optimal choice:

$$\eta_\star = \frac{\eta_{\max}}{1 + \frac{B_{\rm noise}}{B}} \ , \quad \eta_{\max} \equiv \frac{\bar{\mathbf{g}}^2}{\bar{\mathbf{g}} \cdot H \cdot \bar{\mathbf{g}}}, \quad B_{\rm noise} \equiv \frac{\mathrm{Tr}\, H \cdot \Sigma}{\bar{\mathbf{g}} \cdot H \cdot \bar{\mathbf{g}}} \ . \tag{E.3}$$

Notably, the above supports the usual rule of thumb that the learning rate should be increased proportionally to the batch size, at least whenever $B \ll B_{\rm noise}$. The diminishing returns of pushing batch sizes past $B_{\rm noise}$ are also evident. In practice it is too expensive to compute the Hessian, but thankfully the entirely unjustified approximation in which the Hessian is multiple of the identity such that

$$B_{\rm noise} \approx B_{\rm simple} \equiv \frac{\mathrm{Tr}\,\Sigma}{\bar{\mathbf{g}}^2} \ , \tag{E.4}$$

is somehow a decent approximation empirically, and an estimator can be created for $B_{\rm noise}$ in a data-parallel setup; see [21] or Katherine Crowson's implementation or neox for more.

We can further characterize the trade-off between compute and optimization steps. The expected decrease in loss per update is then

$$\langle \delta \mathcal{L} \rangle \approx \frac{\eta_{\max}}{1 + \frac{B_{\rm noise}}{B}} \bar{\mathbf{g}}^2 + \mathcal{O}\left(\eta_{\max}^2\right) \ , \tag{E.5}$$

that is, we would need $1 + \frac{B_{\rm noise}}{B}$ times as many SGD steps to make the same progress we would have as compared to full-batch SGD. If $S_{\min}$ is the number of steps that would have been needed for full-batch SGD, we would need $S = S_{\min} + S_{\min}\frac{B_{\rm noise}}{B}$ steps for minibatch SGD. The total number of examples seen is correspondingly $E = S_{\min} \times (B_{\rm noise} + B) \equiv E_{\min} + S_{\min} B$, and so we see the

[51] The below uses sampling with replacement, while in practice we sample without replacement, but the different is negligible for all practical cases.

trade-off between SGD steps S and compute E alluded to above. These relations can be written as[52]

$$\left(\frac{S}{S_{\min}}-1\right)\left(\frac{E}{E_{\min}}-1\right)=1 \tag{E.6}$$

which represent hyperbolic Pareto frontier curves. So, solutions are of the form $S=(\alpha+1)\,S_{\min}$, $E=\left(\frac{1}{\alpha}+1\right)E_{\min}$ and since $E=BS$ the corresponding batch size is $B_{\rm crit}\equiv\frac{1}{\alpha}B_{\rm noise}$. The parameter α characterizes how much you value the trade-off between these two factors and a reasonable balance is the $\alpha=1$ solution for which $S=2S_{\min}$, $E=2E_{\min}$ and $B_{\rm crit}=B_{\rm noise}$ exactly.

Correspondingly, in [21] they suggest training at precisely this batch size. But it seems much more relevant to balance time against compute directly, rather than optimization steps vs compute. Modeling the total training time by $T\approx S\,(\kappa B+\sigma)$ for some κ,σ to model compute costs[53], then the above is equivalent to

$$T=\frac{(E_{\min}+S_{\min}B)\,(\kappa B+\sigma)}{B}\ . \tag{E.7}$$

which has a minimum at

$$B=\sqrt{\frac{\sigma E_{\min}}{\kappa S_{\min}}}\ . \tag{E.8}$$

for which the total time is

$$T_{\min}=\left(\sqrt{\kappa E_{\min}}-\sqrt{\sigma S_{\min}}\right)^2\ . \tag{E.9}$$

In comparison, the total time for the $B_{\rm crit}=\frac{E_{\min}}{S_{\min}}$ strategy of [21] gives $T_{\min}=2\,(\kappa E_{\min}+\sigma S_{\min})$ which is a factor of $\frac{2}{1-\frac{\sqrt{\sigma\kappa B_{\rm noise}}}{\kappa B_{\rm noise}+\sigma}}$ larger. So, this seems like a better choice of optimal batch size, if you value your time.

F Cheat Sheet

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

Number of model parameters:

$$N_{\rm params}=(4+2E)LD^2+VD+\mathcal{O}\left(DL\right)\approx(4+2E)\,LD^2\ , \tag{F.1}$$

assuming no sharding of the embedding matrix.

[52] The analysis here is simplified in that it assumes that the noise scale and the chosen batch size are both time-independent. There is confusing logic treating the more general case where both $B_{\rm noise}$ and B vary with step in [21], but in any case, the ultimate relations they use are effectively the same.

[53] Computation and communication costs each scale with B, the optimizer step does not (and maybe some overhead?), for instance.

Training Memory costs for mixed-precision training:

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

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

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

G TODO

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

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]. 3, 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. 3, 32

[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]. 32

[4] OpenAI, "Gpt-4 technical report," arXiv:2303.08774 [cs.CL]. 3

[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, 15, 20, 22, 32

[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, 37

[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]. 6

[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]. 12

[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]. 13

[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]. 14

[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]. 16, 17, 18, 20, 21

[13] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," arXiv:2001.08361 [cs.LG]. 25, 26

[14] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," arXiv:2203.15556 [cs.CL]. 25, 26

[15] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," arXiv:1904.09751 [cs.CL]. 29

[16] 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]. 29

[17] C. Chen, "Transformer inference arithmetic,". https://kipp.ly/blog/transformer-inference-arithmetic/. 30

[18] 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. 32

[19] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing* (2009) . 34

[20] H. He, "Making deep learning go brrrr from first principles,". https://horace.io/brrr_intro.html. 36

[21] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team, "An empirical model of large-batch training," arXiv:1812.06162 [cs.LG]. 37, 38, 39