EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade

Homework 11

This homework is due on November 18, at 10:59PM.

1. LoRA

A common strategy for adapting a large pre-trained model to a new task is to update only a subset of its parameters, keeping the rest frozen. Low-Rank Adaptation (LoRA) offers a more flexible approach to this idea. In this problem, we focus on a single weight matrix W with m rows and ℓ columns, where W_0 is the pre-trained value. During LoRA-based training, W is replaced by $W_0 + AB$, where W_0 remains frozen and only A and B are learnable. Here, A is an $m \times k$ matrix and B is a $k \times \ell$ matrix, typically with $k \ll \min(m, \ell)$.

(a) Suppose you are using LoRA to adapt a pretrained deep neural net to a new task and observe that the model is "underfitting" the training data. **What would you try to adjust in the LoRA to get better performance?**

Solution: To address underfitting in LoRA, consider the following strategies:

i. Increase the rank k of the low-rank matrices A and B to allow for more expressive power.
Increasing the rank adds parameters to the LoRA and this extended expressive power generally allows better fitting.

ii. Change the initialization you're using to leverage the SVD of either the original pretrained weight matrix or the matrix-valued gradient for a large amount of training data.
By aligning the initializations to the natural gradient directions in either the training data or the original matrix, we are likely to be able to have more significant impacts faster.

iii. Change the learning rate or other hyperparameters to ensure better convergence.
We know that learning rate is generally the single most important hyperparameter to tune. And underfitting can be a manifestation of simply a too-small learning rate. Furthermore, we know from recent work that there are benefits from having a higher learning rate for the "second" matrix in a LoRA — in this case the A matrix. (See the paper: LoRA+: Efficient Low Rank Adaptation of Large Models, arXiv:2402.12354)

(b) Suppose both A and B are initialized to all zeros. **Why will this cause problems for LoRA-based finetuning?**

Remember, this is going to be trained using SGD-style updates over a training set with a loss function.

Solution: If both A and B are initialized to zero, the LoRA adaptation starts as a zero matrix and the gradients for both A and B will also be zero at initialization. This means the parameters will not update during training, preventing the model from learning any adaptation.

The gradients are zero for B because no change in B will have any effect on the output because it will be multiplied by a zero matrix. The gradients are zero for A because the input activations to A are always zero since they are multiplied by a zero matrix.

(c) Consider the following pseudocode for LoRA initialization:

```
A = torch.nn.Parameter(torch.empty(m, k))
```

```
B = torch.nn.Parameter(torch.empty(k, l))
torch.nn.init.xavier_uniform_(A)
torch.nn.init.xavier_uniform_(B)
```

Why might LoRA fine-tuning not work well with this initialization?

Solution: Since we are fine-tuning a pretrained model, the weights of A and B should be such that the product AB is 0. Why? Because we want to leverage the features that the pretrained model has.

Xavier initialization does not provide this property, as it initializes A and B independently to be nonzero, leading to a nonzero product. The scale of this shift is not infinitesimal, and so we could move the activations to the subsequent layer far outside of the distribution it was trained for.

The resulting model behavior is likely very different from the pretrained model at initialization.

(d) **How much memory is required to store the LoRA adaptation weights (A and B)?** Assume we are using floats (4 bytes per real number) and give your answer in bytes. **How does this compare to storing a single full-rank adaptation matrix?**

Solution: Storing the LoRA weights requires memory for A ($m \times k$) and B ($k \times \ell$), for a total of $mk + k\ell$ parameters. In bytes, this is $4k(\ell + m)$ bytes. In contrast, storing a full-rank adaptation would require $m\ell$ parameters and thus $4m\ell$ bytes. When $k \ll \min(m, \ell)$, LoRA uses much less memory.

2. A Brief Introduction to Transformer Interpretability

Modern neural networks, particularly large language models like Transformers, achieve remarkable performance but are often treated as "black boxes". Mechanistic interpretability is a research field dedicated to reverse-engineering the specific circuits these models learn. Instead of just observing their input-output behavior, we aim to understand the internal mechanisms and computations that lead to their capabilities. This problem explores a mathematical framework proposed by Elhage et al. that treats Transformers as a collection of interacting "circuits" built from attention and MLP layers.

The Residual Stream: A Central Communication Channel

A core concept in this framework is the **residual stream**. At any given layer l, the residual stream, denoted $\mathbf{X}^l$, is a sequence of vectors representing the state of the computation for each token in the input. Crucially, the Transformer, like all modern post-ResNet architectures, is based on residual connections, meaning the output of each layer is *added* to the stream from the previous layer. For a layer l containing a component (e.g., an attention head) with function $f^{(l)}$, the update rule is:

$$\mathbf{X}^{l+1} = \mathbf{X}^l + f^{(l)}(\mathbf{X}^l)$$

The final output of an L-layer model is therefore the sum of the initial embedding and the outputs of all layers:

$$\mathbf{X}^{\text{final}} = \mathbf{X}^0 + \sum_{l=0}^{L-1} f^{(l)}(\mathbf{X}^l)$$

This additive structure is fundamental. It allows us to view the residual stream as a central communication bus or channel where different components read information from and additively write their results back to. This reframing is key to analyzing the function of individual components in relative isolation.

Throughout this problem, we refer to the following variables and their shapes.

Variable	Shape	Description
$\mathbf{T}$	$\mathbb{R}^{n_{\text{vocab}} \times n_{\text{context}}}$	Matrix of one-hot encoded input tokens. Each column is a token.
$\mathbf{X}^l$	$\mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}$	"Residual stream" or "embedding" vectors of model at layer l. Each column is a vector for a token.
$\mathbf{W}_E$	$\mathbb{R}^{d_{\text{model}} \times n_{\text{vocab}}}$	Token embedding matrix. Maps one-hot vectors to the residual stream.
$\mathbf{W}_U$	$\mathbb{R}^{n_{\text{vocab}} \times d_{\text{model}}}$	Token unembedding matrix. Maps final residual stream vectors to logits.
$\mathbf{W}_Q^h, \mathbf{W}_K^h, \mathbf{W}_V^h$	$\mathbb{R}^{d_{\text{head}} \times d_{\text{model}}}$	Query, Key, and Value weight matrices for attention head h.
$\mathbf{W}_O^h$	$\mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$	Output weight matrix for attention head h. This weight matrix left multiplies the output from attention head h and projects it into the dimension of the residual stream.
$\mathbf{A}^h$	$\mathbb{R}^{n_{\text{context}} \times n_{\text{context}}}$	The attention pattern matrix (post-softmax) for attention head h.
$\mathbf{W}_{OV}^h$	$\mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$	The Output-Value matrix for attention head h, defined as $\mathbf{W}_{OV}^h = \mathbf{W}_O^h \mathbf{W}_V^h$.
$\mathbf{W}_{QK}^h$	$\mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$	The Query-Key matrix for attention head h, defined as $\mathbf{W}_{QK}^h = \mathbf{W}_Q^{hT} \mathbf{W}_K^h$.

Note that the superscript h is omitted when context is clear about which attention head we are referring to.

(a) **The Simplest Transformer (A Zero-Layer Model)**

i. Given a matrix of one-hot input tokens $\mathbf{T} \in \mathbb{R}^{n_{\text{vocab}} \times n_{\text{context}}}$, write the mathematical expression for the final logits, $\mathbf{L} \in \mathbb{R}^{n_{\text{vocab}} \times n_{\text{context}}}$. Your expression should be in terms of $\mathbf{T}$, the token embedding matrix $\mathbf{W}_E$, and the unembedding matrix $\mathbf{W}_U$.

Solution: The computation proceeds in two steps:

(1) **Embedding:** The one-hot tokens $\mathbf{T}$ are mapped into the residual stream. Since there are no layers, this initial stream is also the final stream.

$$\mathbf{X}^{\text{final}} = \mathbf{X}^0 = \mathbf{W}_E \mathbf{T}$$

(2) **Unembedding:** The final residual stream vectors are mapped to logits.

$$\mathbf{L} = \mathbf{W}_U \mathbf{X}^{\text{final}}$$

Substituting the first equation into the second gives the full expression:

$$\mathbf{L} = \mathbf{W}_U \mathbf{W}_E \mathbf{T}$$

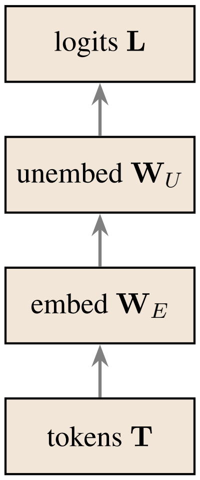


Figure 1: The Simplest Transformer: A Zero-Layer Model

ii. In simple terms, what algorithm does this zero-layer model implement? What information does the prediction for the token at position t depend on? This reveals the baseline functionality of the Transformer architecture before any contextual processing is introduced.

Solution: This model implements a simple, position-wise word lookup and prediction. To see this clearly, let's analyze the computation for a single token at position t. The t-th column of $\mathbf{T}$, denoted $\mathbf{T}_t$, is a one-hot vector in $\mathbb{R}^{n_{\text{vocab}}}$. The product $\mathbf{W}_E\mathbf{T}_t$ selects the column of $\mathbf{W}_E$ corresponding to the input token at that position, giving its embedding $\mathbf{X}_t$. The product $\mathbf{W}_U\mathbf{X}_t$ then computes the logits $\mathbf{L}_t$ for the output token at position t. It is important to note that the calculation of logits $\mathbf{L}_t$ **only depends on the input token $\mathbf{T}_t$**. It has no access to any other tokens in the context. Therefore, this zero-layer model is equivalent to a simple bigram model (predicting the next word based only on the current word) applied independently at every position in the sequence. It has no contextual understanding. All contextual processing in a Transformer comes from the attention and MLP layers. This zero-layer computation is sometimes called the "direct path" in the mechanistic interpretability literature.

(b) **Multi-Head Attention: Concatenation vs. Addition**

In the original Vaswani et al. paper on transformers, a multi-head attention layer is described differently than our "independent circuits" view. There, the outputs of all heads are concatenated and then multiplied by a single large output matrix. This question asks you to prove this is equivalent to an "additive and independent" view.

Consider an attention layer with H heads. Let the value computation output for head h be $\mathbf{r}^h \in \mathbb{R}^{d_{\text{head}} \times n_{\text{context}}}$. In the "concatenation" view, these are stacked vertically to form a matrix $\mathbf{R}^H \in \mathbb{R}^{d_{\text{head}} \cdot H \times n_{\text{context}}}$ and multiplied by a single large output matrix $\mathbf{W}_O^H \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}} \cdot H}$, and the final output is $\mathbf{H} = \mathbf{W}_O^H \mathbf{R}^H$. Typically, the dimension $d_{\text{head}} \cdot H$ is equal to d_{model} where $H = d_{\text{model}}/d_{\text{head}}$. In the "additive and independent" view, each head h has its own output matrix, $\mathbf{W}_O^h \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$, and the total output is $\mathbf{H} = \sum_{h=1}^{H} \mathbf{W}_O^h \mathbf{r}^h$.

(i) Show that these formulations are equivalent. Specifically, demonstrate how $\mathbf{W}_O^H$ can be constructed from the individual $\mathbf{W}_O^h$ matrices to make the two expressions for $\mathbf{H}$ identical.

Solution: We want to show that $\mathbf{W}_O^H \mathbf{R}^H = \sum_{h=1}^{H} \mathbf{W}_O^h \mathbf{r}^h$.
To make the formulations equivalent, we construct the large output matrix $\mathbf{W}_O^H$ by horizontally

concatenating the individual head output matrices:

$$\mathbf{W}_O^H = \begin{bmatrix} \mathbf{W}_O^1 & \mathbf{W}_O^2 & \cdots & \mathbf{W}_O^H \end{bmatrix}$$

where each block $\mathbf{W}_O^h$ has dimensions $d_{\text{model}} \times d_{\text{head}}$.
Now, we compute the product using the rules of block matrix multiplication:

$$\begin{aligned} \mathbf{H} = \mathbf{W}_O^H \mathbf{R}^H &= \begin{bmatrix} \mathbf{W}_O^1 & \mathbf{W}_O^2 & \cdots & \mathbf{W}_O^H \end{bmatrix} \begin{bmatrix} \mathbf{r}^1 \\ \mathbf{r}^2 \\ \vdots \\ \mathbf{r}^H \end{bmatrix} \\ &= \mathbf{W}_O^1 \mathbf{r}^1 + \mathbf{W}_O^2 \mathbf{r}^2 + \cdots + \mathbf{W}_O^H \mathbf{r}^H \\ &= \sum_{h=1}^{H} \mathbf{W}_O^h \mathbf{r}^h \end{aligned}$$

This final expression is identical to the "additive and independent" formulation. This proves that the two perspectives are mathematically equivalent, showing us that we can analyze the contributions of each head independently.

(ii) What is an advantage and disadvantage of each view?

Solution: The concatenation view is more computationally efficient because it only requires a single matrix multiplication. However, it is less interpretable because it obscures the independent contribution of each head. The additive view makes it explicit that each head independently writes to the residual stream, which is more interpretable and aligns with the "residual stream as communication channel" perspective. However, it requires H separate matrix multiplications, which is less efficient.

For the following parts, we consider a Transformer with a single attention layer with H heads and no normalizations as shown in Figure 2.

(c) **The QK Circuit: Determining Attention Patterns**

An attention head can conceptually be split into two independent operations: one that decides *where to look* (the QK circuit) and one that decides *what information to move* (the OV circuit). Let's first analyze the "where to look" mechanism.

The pre-softmax attention score from a "query" token at position i to a "key" token at position j is computed as:

$$\mathbf{S}_{ij} = \mathbf{Q}_i^T \mathbf{K}_j$$

where $\mathbf{Q}_i = \mathbf{W}_Q \mathbf{X}_i$ and $\mathbf{K}_j = \mathbf{W}_K \mathbf{X}_j$ are the query and key vectors.

(i) Starting from the definitions above, derive the expression for $\mathbf{S}_{ij}$ as a bilinear form involving the residual stream vectors $\mathbf{X}_i$ and $\mathbf{X}_j$, and a single "virtual" weight matrix $\mathbf{W}_{QK}$. Explicitly define $\mathbf{W}_{QK}$ in terms of $\mathbf{W}_Q$ and $\mathbf{W}_K$.

Solution: The pre-softmax attention score $\mathbf{S}_{ij} = \mathbf{Q}_i^T \mathbf{K}_j$. We substitute the definitions of query and key vectors, $\mathbf{Q}_i = \mathbf{W}_Q \mathbf{X}_i$ and $\mathbf{K}_j = \mathbf{W}_K \mathbf{X}_j$:

$$\mathbf{S}_{ij} = (\mathbf{W}_Q \mathbf{X}_i)^T (\mathbf{W}_K \mathbf{X}_j)$$

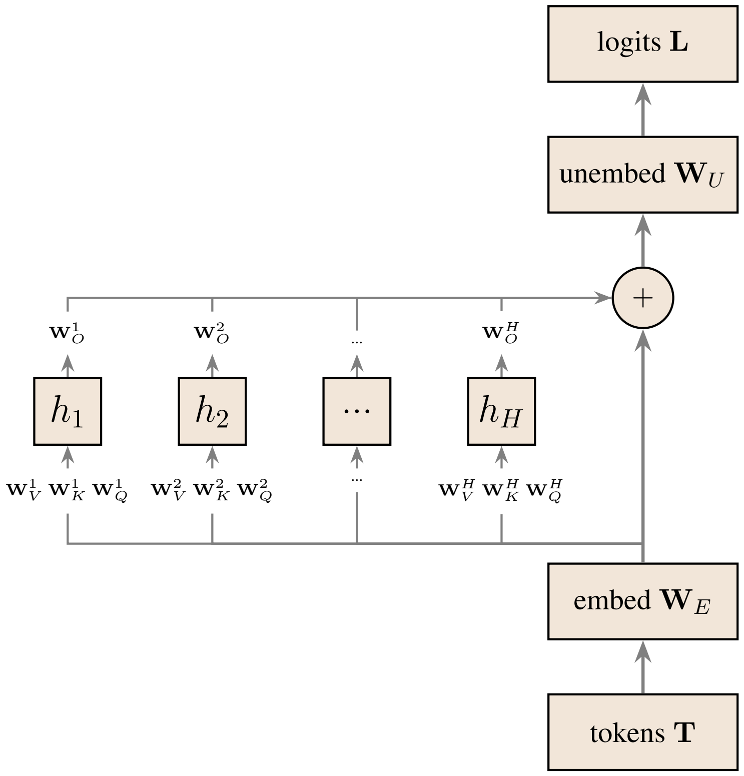


Figure 2: A Transformer With A Single Attention Layer with No Normalizations

Using the property $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$, we get:

$$\mathbf{S}_{ij} = \mathbf{X}_i^T\mathbf{W}_Q^T\mathbf{W}_K\mathbf{X}_j$$

We can define the virtual "Query-Key" matrix $\mathbf{W}_{QK}$ as $\mathbf{W}_Q^T\mathbf{W}_K$, which leads us to the final expression

$$\mathbf{S}_{ij} = \mathbf{X}_i^T\mathbf{W}_{QK}\mathbf{X}_j$$

(ii) The matrix $\mathbf{W}_{QK}$ can be interpreted as defining the "question" the attention head asks to determine which tokens to attend to. Consider a toy scenario where $d_{\text{model}} = 3$.

(1) If $\mathbf{W}_{QK} = \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, what kind of relationship between $\mathbf{X}_i$ and $\mathbf{X}_j$ would lead to a high attention score $\mathbf{S}_{ij} = \mathbf{X}_i^T\mathbf{W}_{QK}\mathbf{X}_j$? Describe in words what this head "looks for."

Solution: If $\mathbf{W}_{QK} = \mathbf{I}$, then $\mathbf{S}_{ij} = \mathbf{X}_i^T\mathbf{I}\mathbf{X}_j = \mathbf{X}_i^T\mathbf{X}_j$. This is the dot product of the two residual stream vectors. A high score means the vectors are pointing in a similar direction. This head "looks for" tokens whose residual stream vectors are most similar to its own.

(2) If $\mathbf{W}_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, what kind of relationship would lead to a high attention score? What happens to information in the second and third dimensions?

Solution: Let $\mathbf{X}_i = \begin{bmatrix} \mathbf{X}_{i1} \\ \mathbf{X}_{i2} \\ \mathbf{X}_{i3} \end{bmatrix}$ and $\mathbf{X}_j = \begin{bmatrix} \mathbf{X}_{j1} \\ \mathbf{X}_{j2} \\ \mathbf{X}_{j3} \end{bmatrix}$. The score is:

$$\begin{aligned} \mathbf{S}_{ij} &= \mathbf{X}_i^T \mathbf{W}_{QK} \mathbf{X}_j \\ &= \begin{bmatrix} \mathbf{X}_{i1} & \mathbf{X}_{i2} & \mathbf{X}_{i3} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{j1} \\ \mathbf{X}_{j2} \\ \mathbf{X}_{j3} \end{bmatrix} \\ &= \mathbf{X}_{i1} \mathbf{X}_{j1} \end{aligned}$$

This head only cares about the first dimension of the residual stream vectors. Information in the second and third dimensions is completely ignored—effectively "deleted" from the attention computation. This demonstrates how the QK circuit can selectively attend to specific features while ignoring others.

(3) If $\mathbf{W}_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$, what pattern does this head look for? How does this differ from the identity case?

Solution: The score is:

$$\begin{aligned} \mathbf{S}_{ij} &= \mathbf{X}_i^T \mathbf{W}_{QK} \mathbf{X}_j \\ &= \mathbf{X}_{i1} \mathbf{X}_{j1} + \mathbf{X}_{i2} \mathbf{X}_{j2} - \mathbf{X}_{i3} \mathbf{X}_{j3} \end{aligned}$$

This head rewards similarity in the first two dimensions but *dissimilarity* in the third dimension. For example, it would give a high score if $\mathbf{X}_i$ and $\mathbf{X}_j$ are both positive in dimensions 1 and 2, but have opposite signs in dimension 3. This demonstrates that the QK circuit can implement complex logical queries that combine multiple conditions, including both positive and negative correlations.

(4) If $\mathbf{W}_{QK} = -\mathbf{I} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$, what pattern does this head look for? What is the effect of this on the attention pattern after softmax?

Solution: The score is:

$$\mathbf{S}_{ij} = \mathbf{X}_i^T (-\mathbf{I}) \mathbf{X}_j = -\mathbf{X}_i^T \mathbf{X}_j$$

This is the *negative* dot product. High scores occur when vectors point in *opposite* directions. Importantly, this "suppresses" or "deletes" attention to similar tokens. After softmax, positions with similar residual stream vectors will receive low attention weights, while positions with dissimilar vectors will receive higher weights. This demonstrates that the QK circuit can be programmed to *avoid* certain patterns, not just seek them out. Such "negative queries" could be useful for preventing the model from attending to redundant information or for implementing more complex logical operations like "attend to tokens that are different from the current one."

(d) **The OV Circuit: Reading and Writing Information**

Now let's analyze the information-moving part of the attention head. Once the attention pattern $\mathbf{A}^h$ is determined (by the QK circuit), the head reads information from attended tokens and writes it to the destination. This is governed by the OV circuit. Note that the output of head h is:

$$h(\mathbf{X}^0) = \mathbf{W}_O^h \mathbf{W}_V^h \mathbf{X}^0 \mathbf{A}^{hT} = \mathbf{W}_{OV}^h \mathbf{X}^0 \mathbf{A}^{hT}$$

where $\mathbf{A}^h$ is the attention pattern (post-softmax) determined by the QK circuit.

(i) **[Understanding the Attention-Weighted Average]**
Before proceeding, let's clarify what $\mathbf{X}^0\mathbf{A}^T$ computes. Recall that:

- $\mathbf{X}^0 \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}$ where the j-th column $\mathbf{X}_j^0$ is the residual stream vector for the token at position j.
- $\mathbf{A} \in \mathbb{R}^{n_{\text{context}} \times n_{\text{context}}}$ is the attention pattern where $\mathbf{A}_{ij}$ is the attention weight from destination position i to source position j.
- Therefore, $\mathbf{A}^T \in \mathbb{R}^{n_{\text{context}} \times n_{\text{context}}}$ where $(\mathbf{A}^T)_{ji} = \mathbf{A}_{ij}$.

Show that the i-th column of $\mathbf{X}^0\mathbf{A}^T$ is an attention-weighted average of the source token vectors. Specifically, show:

$$(\mathbf{X}^0\mathbf{A}^T)_i = \sum_{j=1}^{n_{\text{context}}} \mathbf{A}_{ij}\mathbf{X}_j^0$$

Solution: The i-th column of the matrix product $\mathbf{X}^0\mathbf{A}^T$ is obtained by multiplying $\mathbf{X}^0$ with the i-th column of $\mathbf{A}^T$:

$$(\mathbf{X}^0\mathbf{A}^T)_i = \mathbf{X}^0 \cdot (\mathbf{A}^T)_i$$

The i-th column of $\mathbf{A}^T$ equals the i-th row of $\mathbf{A}$ (written as a column vector):

$$(\mathbf{A}^T)_i = \begin{bmatrix} \mathbf{A}_{i1} \\ \mathbf{A}_{i2} \\ \vdots \\ \mathbf{A}_{i,n_{\text{context}}} \end{bmatrix}$$

This vector contains all the attention weights from destination position i to each source position j.

Now, we can write $\mathbf{X}^0$ as a block matrix of its columns:

$$\mathbf{X}^0 = \begin{bmatrix} \mathbf{X}_1^0 & \mathbf{X}_2^0 & \cdots & \mathbf{X}_{n_{\text{context}}}^0 \end{bmatrix}$$

The matrix-vector product is:

$$\begin{aligned}(\mathbf{X}^0\mathbf{A}^T)_i &= \begin{bmatrix} \mathbf{X}_1^0 & \mathbf{X}_2^0 & \cdots & \mathbf{X}_{n_{\text{context}}}^0 \end{bmatrix} \begin{bmatrix} \mathbf{A}_{i1} \\ \mathbf{A}_{i2} \\ \vdots \\ \mathbf{A}_{i,n_{\text{context}}} \end{bmatrix} \\ &= \mathbf{A}_{i1}\mathbf{X}_1^0 + \mathbf{A}_{i2}\mathbf{X}_2^0 + \cdots + \mathbf{A}_{i,n_{\text{context}}}\mathbf{X}_{n_{\text{context}}}^0 \\ &= \sum_{j=1}^{n_{\text{context}}} \mathbf{A}_{ij}\mathbf{X}_j^0\end{aligned}$$

This is precisely an attention-weighted average! The i-th column of $\mathbf{X}^0\mathbf{A}^T$ combines the source token vectors $\mathbf{X}^0_j$ with weights $\mathbf{A}_{ij}$. Since attention weights satisfy $\sum_j \mathbf{A}_{ij} = 1$ (from softmax), this is a convex combination of the source vectors.

Interpretation: For each destination position i, the attention pattern $\mathbf{A}$ determines how much to "gather" from each source position j. The result is a new vector that blends information from multiple source positions according to the attention weights.

(ii) Write the full expression for the final residual stream, $\mathbf{X}^{\text{final}}$, in terms of the initial stream $\mathbf{X}^0$ and the heads' weight matrices.

Solution: The final residual stream $\mathbf{X}^{\text{final}}$ is the sum of the initial stream $\mathbf{X}^0$ and the outputs of all attention heads:

$$\begin{aligned}\mathbf{X}^{\text{final}} &= \mathbf{X}^0 + \sum_{i=1}^{H} h_i(\mathbf{X}^0) \\ &= \mathbf{X}^0 + \sum_{i=1}^{H} \mathbf{W}^i_O \mathbf{W}^i_V \mathbf{X}^0 \mathbf{A}^{iT} \\ &= \mathbf{X}^0 + \sum_{i=1}^{H} \mathbf{W}^i_{OV} \mathbf{X}^0 \mathbf{A}^{iT}\end{aligned}$$

(iii) The update vector added to the residual stream at destination position t by head i is the t-th column of the head's output, denoted $h_i(\mathbf{X}^0)_t$. Prove that this update vector must lie within the column space of $\mathbf{W}^i_{OV}$. That is, show that $h_i(\mathbf{X}^0)_t \in \text{Col}(\mathbf{W}^i_{OV})$. This demonstrates that the head can only write to a low-dimensional subspace of the d_{model}-dimensional residual stream.

Solution: Let the head's output be $h_i(\mathbf{X}^0) = \mathbf{W}^i_{OV}\mathbf{X}^0\mathbf{A}^{iT}$. Define an intermediate matrix $\mathbf{Z}^i = \mathbf{X}^0\mathbf{A}^{iT} \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}$, which represents the attention-weighted average of the residual stream vectors. The head's output can then be written as:

$$h_i(\mathbf{X}^0) = \mathbf{W}^i_{OV}\mathbf{Z}^i$$

Let $h_i(\mathbf{X}^0)_t$ be the t-th column of $h_i(\mathbf{X}^0)$ and $\mathbf{Z}^i_t$ be the t-th column of $\mathbf{Z}^i$. By the definition of matrix-matrix multiplication, the t-th column of the product is:

$$h_i(\mathbf{X}^0)_t = \mathbf{W}^i_{OV}\mathbf{Z}^i_t$$

A matrix-vector product $\mathbf{W}^i_{OV}\mathbf{Z}^i_t$ is, by definition, a linear combination of the columns of $\mathbf{W}^i_{OV}$, where the coefficients are the elements of $\mathbf{Z}^i_t$:

$$h_i(\mathbf{X}^0)_t = \mathbf{W}^i_{OV}\mathbf{Z}^i_t = \sum_{j=1}^{d_{\text{model}}} (\mathbf{W}^i_{OV})_j(\mathbf{Z}^i_t)_j$$

where $(\mathbf{W}^i_{OV})_j$ is the j-th column of $\mathbf{W}^i_{OV}$ and $(\mathbf{Z}^i_t)_j$ is the j-th element of $\mathbf{Z}^i_t$.

Since $h_i(\mathbf{X}^0)_t$ is a linear combination of the columns of $\mathbf{W}^i_{OV}$, it must lie within the column space of $\mathbf{W}^i_{OV}$, i.e., $h_i(\mathbf{X}^0)_t \in \text{Col}(\mathbf{W}^i_{OV})$. This holds for any position t and any head i, so every update vector is constrained to this "write subspace."

(e) **Formalizing Read and Write Subspaces via SVD**

Important: Separating the QK and OV Circuits
In this analysis, we focus exclusively on the **OV circuit** $\mathbf{W}_{OV}$, which characterizes *what* information is moved and *how* it is transformed.
Recall from part (d) that the full output of an attention head is:

$$h(\mathbf{X}^0) = \mathbf{W}_{OV}\mathbf{X}^0\mathbf{A}^T$$

This can be decomposed into two independent operations:

- **QK Circuit** (analyzed in part c): Computes the attention pattern $\mathbf{A}$, which determines *which* source tokens to attend to at each destination position.
- **OV Circuit** (analyzed here): The linear transformation $\mathbf{W}_{OV}$ applied to source tokens, determining *what* information is extracted and *where* it is written.

In our SVD analysis below, when we write $\mathbf{W}_{OV}\mathbf{X}_{\text{src}}$, we are analyzing what happens to information from a *single* source token. The attention pattern then determines which source tokens this transformation is applied to and with what weights. These two circuits operate *independently*: you can understand what the OV circuit does to each source token without knowing which tokens will be attended to.

In part (d), we showed that each attention head can only write to the column space of $\mathbf{W}_{OV}$, a low-dimensional subspace. But this raises deeper questions: *which* directions in the source token does the head read from? And *which* directions in the destination does it write to? The Singular Value Decomposition (SVD) provides a complete answer, revealing that attention heads are specialized communication channels between specific subspaces.

(i) Prove that $\text{rank}(\mathbf{W}_{OV}) \le d_{\text{head}}$. Why does this imply that $\mathbf{W}_{OV}$ is a low-rank matrix (in typical transformer architectures)?

Solution: Recall from linear algebra that the rank of a matrix product is bounded by the minimum rank of its factors:

$$\text{rank}(\mathbf{AB}) \le \min(\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B}))$$

Here, $\mathbf{W}_{OV} = \mathbf{W}_O\mathbf{W}_V$ where $\mathbf{W}_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$ and $\mathbf{W}_V \in \mathbb{R}^{d_{\text{head}} \times d_{\text{model}}}$.
The rank of any matrix is bounded by its smallest dimension. Since $\mathbf{W}_O$ has d_{head} columns, $\text{rank}(\mathbf{W}_O) \le d_{\text{head}}$. Similarly, since $\mathbf{W}_V$ has d_{head} rows, $\text{rank}(\mathbf{W}_V) \le d_{\text{head}}$.
Therefore:

$$\text{rank}(\mathbf{W}_{OV}) \le \min(d_{\text{head}}, d_{\text{head}}) = d_{\text{head}}$$

In typical architectures, $d_{\text{head}} \ll d_{\text{model}}$ (e.g., $d_{\text{head}} = 64$ while $d_{\text{model}} = 768$), making $\mathbf{W}_{OV}$ a very low-rank approximation of a full $d_{\text{model}} \times d_{\text{model}}$ matrix. This low-rank structure is what allows us to interpret the head as a specialized channel.

(ii) Because $\mathbf{W}_{OV}$ is low-rank, its SVD has a special structure. Let $\mathbf{W}_{OV} = \mathbf{U\Sigma V}^T$ be the (compact) SVD, where $\mathbf{U} \in \mathbb{R}^{d_{\text{model}} \times r}$, $\mathbf{\Sigma} \in \mathbb{R}^{r \times r}$, and $\mathbf{V} \in \mathbb{R}^{d_{\text{model}} \times r}$, with $r = \text{rank}(\mathbf{W}_{OV}) \le d_{\text{head}}$. Consider a source token with residual stream vector $\mathbf{X}_{\text{src}} \in \mathbb{R}^{d_{\text{model}}}$. Show that the output of the OV circuit can be decomposed as:

$$\mathbf{W}_{OV}\mathbf{X}_{\text{src}} = \sum_{k=1}^{r} \sigma_k(\mathbf{V}_k^T\mathbf{X}_{\text{src}})\mathbf{U}_k \tag{1}$$

where $\mathbf{U}_k$ and $\mathbf{V}_k$ are the k-th columns of $\mathbf{U}$ and $\mathbf{V}$, respectively, and σ_k is the k-th singular value.

Solution: Starting with the SVD:

$$\begin{aligned}\mathbf{W}_{OV}\mathbf{X}_{\text{src}} &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)\mathbf{X}_{\text{src}} \\ &= \mathbf{U}(\mathbf{\Sigma}(\mathbf{V}^T\mathbf{X}_{\text{src}}))\end{aligned}$$

We analyze this computation in three steps:
Step 1 - Reading: $\mathbf{V}^T\mathbf{X}_{\text{src}}$ computes the projections of the source vector onto each right singular vector. Since $\mathbf{V} = [\mathbf{V}_1\ \mathbf{V}_2\ \cdots\ \mathbf{V}_r]$ with orthonormal columns, we have:

$$\mathbf{V}^T\mathbf{X}_{\text{src}} = \begin{bmatrix}\mathbf{V}_1^T\mathbf{X}_{\text{src}} \\ \mathbf{V}_2^T\mathbf{X}_{\text{src}} \\ \vdots \\ \mathbf{V}_r^T\mathbf{X}_{\text{src}}\end{bmatrix} \in \mathbb{R}^r$$

Each element $\mathbf{V}_k^T\mathbf{X}_{\text{src}}$ is a scalar measuring how much of $\mathbf{X}_{\text{src}}$ lies in the direction $\mathbf{V}_k$.
Step 2 - Weighting: $\mathbf{\Sigma}$ scales each projection by its importance:

$$\mathbf{\Sigma}(\mathbf{V}^T\mathbf{X}_{\text{src}}) = \begin{bmatrix}\sigma_1(\mathbf{V}_1^T\mathbf{X}_{\text{src}}) \\ \sigma_2(\mathbf{V}_2^T\mathbf{X}_{\text{src}}) \\ \vdots \\ \sigma_r(\mathbf{V}_r^T\mathbf{X}_{\text{src}})\end{bmatrix} \in \mathbb{R}^r$$

Step 3 - Writing: $\mathbf{U}$ maps these weighted projections to the output:

$$\mathbf{U}\begin{bmatrix}\sigma_1(\mathbf{V}_1^T\mathbf{X}_{\text{src}}) \\ \vdots \\ \sigma_r(\mathbf{V}_r^T\mathbf{X}_{\text{src}})\end{bmatrix} = \sum_{k=1}^{r}\sigma_k(\mathbf{V}_k^T\mathbf{X}_{\text{src}})\mathbf{U}_k$$

This final expression shows that the output is a linear combination of the left singular vectors $\{\mathbf{U}_k\}$, with coefficients determined by how much the input aligns with each right singular vector $\{\mathbf{V}_k\}$.

(iii) Based on the decomposition above, define precisely:

(a) The **read subspace**: Which directions in $\mathbf{X}_{\text{src}}$ can the head extract information from?

(b) The **write subspace**: Which directions in the destination residual stream can the head write to?

(c) What happens to information in $\mathbf{X}_{\text{src}}$ that is orthogonal to the read subspace?

Hint: Think about which matrix "touches" the input first (this determines what can be read) and which matrix determines the form of the output (this determines what can be written).

Solution: **Intuition:** When we compute $\mathbf{W}_{OV}\mathbf{X}_{\text{src}} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{X}_{\text{src}}$ from right to left:

- $\mathbf{V}^T$ operates on the *input*, extracting information by projecting onto its rows (the $\mathbf{V}_k$ vectors). If $\mathbf{V}_k^T\mathbf{X}_{\text{src}} = 0$, that direction cannot be read. So $\mathbf{V}$ defines what can be *read*.
- $\mathbf{U}$ produces the *output* as a linear combination of its columns (the $\mathbf{U}_k$ vectors). The output must lie in span($\mathbf{U}$). So $\mathbf{U}$ defines what can be *written*.

From $\mathbf{W}_{OV}\mathbf{X}_{\text{src}} = \sum_{k=1}^{r}\sigma_k(\mathbf{V}_k^T\mathbf{X}_{\text{src}})\mathbf{U}_k$:
(a) **Read Subspace:** span$(\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_r) \subseteq \mathbb{R}^{d_{\text{model}}}$

The head extracts information by projecting $\mathbf{X}_{\text{src}}$ onto each right singular vector $\mathbf{V}_k$ via the inner products $\mathbf{V}_k^T \mathbf{X}_{\text{src}}$. Only components of $\mathbf{X}_{\text{src}}$ that lie in this r-dimensional subspace can influence these projections. The read subspace has dimension $r \leq d_{\text{head}} \ll d_{\text{model}}$.

(b) **Write Subspace:** $\text{span}(\mathbf{U}_1, \mathbf{U}_2, \ldots, \mathbf{U}_r) \subseteq \mathbb{R}^{d_{\text{model}}}$

The output is explicitly a linear combination $\sum_{k=1}^{r} c_k \mathbf{U}_k$ where $c_k = \sigma_k(\mathbf{V}_k^T \mathbf{X}_{\text{src}})$ are scalars. By definition, any such linear combination must lie in $\text{span}(\mathbf{U}_1, \ldots, \mathbf{U}_r)$. This confirms our earlier result from part (d): $\text{span}(\mathbf{U}_1, \ldots, \mathbf{U}_r) = \text{Col}(\mathbf{W}_{OV})$.

(c) If $\mathbf{X}_{\text{src}}$ has a component $\mathbf{X}_\perp$ orthogonal to all $\mathbf{V}_k$, then $\mathbf{V}_k^T \mathbf{X}_\perp = 0$ for all k. This component contributes nothing to the sum, so it is completely ignored—the head is "blind" to information outside its read subspace.

(iv) **[Connecting to the Value Projection]** The value vector computed by the head is $\mathbf{v} = \mathbf{W}_V \mathbf{X}_{\text{src}}$, which lives in $\mathbb{R}^{d_{\text{head}}}$. Explain in 2-3 sentences how the read subspace $\text{span}(\mathbf{V}_1, \ldots, \mathbf{V}_r)$ relates to what $\mathbf{W}_V$ can "see" in the source token. Why is it natural that the read subspace dimension is at most d_{head}?

Solution: The value projection $\mathbf{W}_V$ maps the d_{model}-dimensional residual stream to a d_{head}-dimensional value vector, so it can only extract at most d_{head} independent pieces of information from $\mathbf{X}_{\text{src}}$. The read subspace, defined by the right singular vectors of $\mathbf{W}_{OV} = \mathbf{W}_O \mathbf{W}_V$, captures precisely which $r \leq d_{\text{head}}$ directions in the source residual stream the value vector is sensitive to. Any information orthogonal to this subspace is lost in the projection to $\mathbb{R}^{d_{\text{head}}}$ and cannot be recovered by $\mathbf{W}_O$, which is why the full OV circuit can only read from this limited subspace.

(v) **[Synthesis]** The paper describes attention heads as moving information "from the residual stream of one token to another." In 2-3 sentences, explain how the SVD $\mathbf{W}_{OV} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ provides a complete characterization of this information movement. What role does each component ($\mathbf{V}$, $\mathbf{\Sigma}$, $\mathbf{U}$) play, and how does this relate to the attention pattern $\mathbf{A}$?

Solution: The SVD reveals that information movement happens in three stages: (1) The right singular vectors $\mathbf{V}$ define the *read subspace*—which specific directions in each source token the head extracts information from. (2) The singular values $\mathbf{\Sigma}$ weight how important each extracted direction is. (3) The left singular vectors $\mathbf{U}$ define the *write subspace*—where this information is written in the destination token's residual stream. This OV circuit characterization applies to *every* source token identically and is completely independent of the attention pattern $\mathbf{A}$ (determined by the QK circuit), which only controls *which* source tokens to move information from and *how much weight* to give each. Together, the QK circuit (where to look) and OV circuit (what to move) fully specify the attention head's operation.

3. Coding Question: Transformer Interpretability

Please follow the instructions in this notebook. You will implement a single attention head and then create an induction copy head by combining a previous token head with a copying head.

This question was adapted from a past interview question by Anthropic.

4. Scaling Laws of Batch Size

A common question in neural network training is, how should I select my hyperparameters? While a proper hyperparameter sweep will always provide the best answer, sweeps can become impractical especially at larger network sizes.

In this homework question, we will derive a simple scaling law for the optimal learning rate under varying batch sizes. Complete this notebook, and answer the following questions in your submission:

(a) **For linear regression using SGD, attach curves of learning rate vs MSE loss for all the considered batch sizes, and curve showing relationship between batch size and optimal learning rate. What function does this resemble?**

(b) **Attach the same curves but for an MLP instead of linear regression. How do the scaling laws differ?**

(c) **Finally, show the same curves when using the Adam optimizer instead of SGD. Does the scaling for learning rate change when using Adam?**

5. Fermi Estimation for Large-scale Deep Learning Models

Fermi estimation is a technique for estimating quantities through rough approximations and educated guesses. Named after physicist Enrico Fermi, this method involves breaking down a problem into simpler, more manageable parts, making reasonable assumptions, and using simple arithmetic to arrive at a good enough answer.

In this question, you will be walked through a simple example of Fermi estimation of a deep learning model. Specifically, we will try to estimate the design parameters of a hypothetical GPT-6 with 100 trillion parameters.

This question is perhaps a bit reading-heavy, but the calculations are very simple, and I hope you find the lesson interesting.

(a) (**GPT-like AGI**) This is **not a question**, but some reading material to get you into the mood for Fermi estimation. We estimate the number of parameters necessary for achieving human-level Artificial General Intelligence, under two assumptions: that a GPT-like architecture is enough; that it requires only matching the human brain in the "numbers".

Let's estimate how many layers the hypothetical GPT model should have. When prompted with a question, the first answer comes to mind in about a second, and it is generally a good one, if not the best. Perhaps slow deliberation is nothing but stringing together a long chain of snap judgments, guided by snap judgments about snap judgments.

The characteristic time-scale of a brain is 0.01 seconds – the fastest brain wave, gamma wave, is 100 Hz. This indicates that the brain takes on the order of 100 steps to make a snap decision. This is the "hundred-step-rule" of Jerome Feldman.[1]

This corresponds very well with the largest model of GPT-3, which has 96 layers.

How many parameters would such a model require? The brain has 10^{15} synapses. It's unclear how precise each synapse is, but one estimate states that the hippocampal synapse has a precision of about 5 bits[2], which can be stored within a 16-bit floating point number, with room to spare.

Assuming that, we expect an AGI GPT to have 10^{15} (1000 trillion) parameters. This homework question does not scale all the way up to 1000 trillion parameters, but only up to 100 trillion parameters. You can do the same calculations for the 1000 trillion parameter model, however.

(b) (**Chinchilla Scaling Law**) The paper "Training Compute-Optimal Large Language Models" (2022) reported a series of training runs on language models, trained by Google DeepMind researchers. Each training run is characterized by four numbers:

- L: the final loss (negative log-likelihood per token) achieved by the trained model.
- N: the number of parameters in the model.
- D: training dataset size, measured in tokens.
- C: training compute cost, measured in FLOP.

[1] Feldman, Jerome A., and Dana H. Ballard. "Connectionist models and their properties." Cognitive science 6.3 (1982): 205-254.

[2] Bartol Jr, Thomas M., et al. "Nanoconnectomic upper bound on the variability of synaptic plasticity." elife 4 (2015): e10778.

After training a few hundred models, they obtained a large dataset of (L, N, D, C), and they fitted a statistical law of the form

$$L = \frac{A}{N^\alpha} + \frac{B}{D^\beta} + L_0,$$

where the parameters are

$$\alpha = 0.34, \beta = 0.28, A = 406.4, B = 410.7, L_0 = 1.69.$$

They also estimated that the cost of training compute C is proportional to ND. This is understandable, because each token must flow through the entire model and "hit" each parameter once, incurring a fixed number of floating point operations. They estimated that it takes 6 FLOPs per parameter per token. That is,

$$C = C_0 ND, \quad C_0 = 6$$

Given the assumptions, for each fixed computing budget C, we can solve for the optimal D and N, which is usually referred to as "Chinchilla optimal" training:

$$\begin{cases} \min_{N,D} L = \frac{A}{N^\alpha} + \frac{B}{D^\beta} + L_0 \\ \text{such that } C_0 ND = C \end{cases}$$

Solve the above equations symbolically to find N_{opt}, D_{opt} as a function of $C, C_0, \alpha, \beta, A, B$. Then, plug in the numerical values of the parameters, to find a numerical expression for N_{opt}, D_{opt} as a function of C.

Solution:

Since $C = C_0 ND$, we have $N = \frac{C}{C_0 D}$. Plug it into $\min_{N,D} L = \frac{A}{N^\alpha} + \frac{B}{D^\beta} + L_0$, we obtain

$$\min_D L = \frac{A}{(\frac{C}{C_0 D})^\alpha} + \frac{B}{D^\beta} + L_0$$

Take derivative with respect to D and set it to zero. We get an expression for D_{opt}. Plug it back to $C = C_0 ND$, we get an expression for D_{opt}. These simplify to:

$$N_{opt}(C) = G\left(\frac{C}{C_0}\right)^a, \quad D_{opt}(C) = G^{-1}\left(\frac{C}{C_0}\right)^b, \quad \text{where} \quad G = \left(\frac{\alpha A}{\beta B}\right)^{\frac{1}{\alpha+\beta}}, a = \frac{\beta}{\alpha+\beta}, b = \frac{\alpha}{\alpha+\beta}.$$

Plugging in the numerical values, we get

$$\begin{cases} N_{opt}(C) = 0.6\, C^{0.45} \\ D_{opt}(C) = 0.3\, C^{0.55} \\ L_{opt}(C) = 1070\, C^{-0.154} + 1.7 \end{cases}$$

(c) In the same paper, they also performed a *direct* statistical fitting, to find the optimal N, D for a given C, without going through the intermediate steps above. This gives a slightly different result (only slightly different – as you would know after solving the previous problem):

$$N_{opt}(C) = 0.1C^{0.5}; \quad D_{opt}(C) = 1.7C^{0.5}.$$

For the rest of the question, we will use **these equations** as the Chinchilla scaling laws. **Do not** use the equations you derived for the previous problem.

Suppose we decide that our next AI should have 100 trillion ($N = 10^{14}$) parameters, and we use Chinchilla scaling laws. **How much compute would it cost to train, and how many tokens would its training dataset have?**

Solution: $N = 0.1 \times C^{0.5} = 10^{14}$, so $C = 10^{30}$ FLOP, and $D = 1.7 \times 10^{15}$, or 1700 trillion tokens.

(d) (**Dataset size**) Assuming each English word cost about 1.4 tokens, **how many English words would 1000 trillion tokens be?** Assuming each page has 400 words, and each book has 300 pages, **how many books is that?** To put it into context, look up the size of Library of Congress, and Google Books, and **compare with the number we just calculated.**

Solution: 1000 trillion / 1.4 = 700 trillion words. If each book has 400 * 300 = 0.12 million words, then that is 6 billion books, if they were all in English.

Since humans are kind of the same everywhere, book lengths should be kind of the same everywhere – information-dense languages would naturally have books with lower apparent word-count, but the same information (measured in bytes).

For some context...

- The Library of Congress has about 50 million books, and Google Books has scanned "more than 40 million books in over 400 languages". They estimated that there are "about 130 million books in the world".
- Sci-Hub and LibGen together cover a large portion of the world's scientific books and papers, and their datasets "together take up about 110 TB as of 2020". Most of the data is due to high-resolution scanned pages, which can take up 100x storage than a page of pure text.

This shows that the world's high-quality text is not enough to train a 100 trillion parameter language model. Further progress demands us to use non-textual training data.

(e) (**Memory requirement**) Typically, a deep learning model has 16-bit floating point parameters. Modern systems sometimes use lower precision (e.g. 8-bit) floating point numbers to save space, but generally it is necessary to use at least 16-bit during training, and the model is converted to lower precision after training ("post-training quantization").

Given that each parameter is a 16-bit floating point number, how much memory does it cost to store a model with 1 billion parameters? How about our hypothetical GPT-6, which has 1 trillion parameters? How many H200 GPUs (VRAM $\approx$ 100 GB) would be required to contain the full GPT-6 model?

Solution: 1 billion parameters is 2 billion bytes, or 2 GB. Our hypothetical GPT-6 would take up to 200 TB. It would take 2000 H200s to contain it.

Now, 2000 H200 GPUs has a running cost of about 3000 USD/hr, but not such a terrible problem, but then, during training, you would have to run an optimizer like Adam, which would need three floating point numbers per parameter (gradient, momentum, scale), instantly increasing the memory requirement to 6000 H200 GPUs.

(f) (**Memory cost**)

This table[3] gives the price per megabyte of different storage technology, in price per megabyte, up to 2025.

[3]Source: Storage 2: Cache model – CS 61 2025.

Year	Memory (DRAM)	Flash/SSD	Hard disk
~1955	$613,000,000		$9,290
1970	$1,090,000		$388
1990	$221		$8.13
2003	$0.134	$0.455	$0.00194
2010	$0.0283	$0.00358	$0.000108
2025	$0.0040	$0.00005	$0.000039

The same costs *relative* to the cost of a hard disk in ~2025:

Year	Memory (DRAM)	Flash/SSD	Hard disk
~1955	15,700,000,000,000		238,000,000
1970	28,000,000,000		9,950,000
1990	5,670,000		208,000
2003	3,440	11,600	49.7
2010	727	91.8	2.79
2025	102	1.28	1.00

Suppose long-term memory storage can last 1 year before being replaced. How much money does it cost per year to store a 100 trillion parameter model on an SSD, the most expensive form of long-term storage? How much money does it cost to store a 100 trillion parameter model on DRAM memory? Use 2025 prices.

Solution: SSD cost 0.00005 USD/MB in 2025, or 0.05 USD/GB. 100 trillion parameters cost 200000 GB of storage, or about 10000 USD. So the total cost of storage is 10000 USD/year.

In contrast, DRAM cost 0.004 USD/MB, which is 80x that of SSD, so the total cost is 800000 USD.

Now, compared with the cost of the H200 GPUs themselves? It takes about 2000 H200 GPUs to run the model, and each would cost about 30,000 USD. So the cost of long-term memory is essentially zero, and the cost of DRAM is about 12000 / (30000*2000) = 0.02% of the total cost of GPU.

So what is the limit? The memory bandwidth, which we will see in the next question.

(g) (**Memory bandwidth and latency**) While the memory itself is cheap, moving the data requires expensive high-bandwidth wiring. Indeed, the memory bandwidth between the DRAM (or "VRAM" for "Video RAM") and the little processors on the GPU is a main bottleneck on how good the GPU can perform.

During a single forward pass of a model, the parameters of the model are loaded from the DRAM of the GPU into the fast cache memories, then pushed through the thousands of computing processors on the GPU.

H200 GPU has a memory bandwidth of 4.8 TB/s.

What is the minimal latency, in seconds, for the GPU to perform a single forward pass through our hypothetical GPT-6 model with 100 trillion parameters? How many tokens can it output (autoregressively) in one minute? How about GPT-3 (175 billion parameters)?

Note: Since we are just trying to compute an order-of-magnitude estimate, let's assume for the problem that the model fits onto a single DRAM on a single GPU. You can also ignore the need to read/write model activations and optimizer states.

Solution: Since the model takes up 200 TB of memory, it would take at least 41.67 seconds to even load the model from DRAM, during a single forward pass. That is extremely slow!

Autoregression means that the next token cannot be generated until the previous one is already generated, so the model can only generate one token per 41.67 seconds, or 1.44 tokens per minute. A Chat-GPT-6 would be a very slow talker!

GPT-3 with 175 billion parameters would run 100000/175 times faster, at 822.86 tokens per minute.

There are some ways to improve latency. For example, the model can be parallelized over several GPUs, which effectively increases the memory bandwidth. For example, "tensor parallelism" splits each layer into several GPUs.

There is also "pipeline parallelism", which splits the model into layers. The first few layers go into one GPU, the next few go into another, and so on. This does NOT decrease latency, but it does increase throughput.

The fundamental bottleneck in an autoregressive model like GPT is that, by design, they have to start a sentence without knowing how it will end. That is, they have to generate the first token before generating the next one, and so on. This can never be parallelized (except by egregeous hacks like speculative decoding).

One reason Transformers dominated over RNN is that training and inferring an RNN *both* must be done one-token-at-a-time. For Transformers, training can be done in parallel over the entire string. Inferring however still cannot be parallelized.

Parallization can be a subtle business. However, there is a very simple method to improve thoroughput: batch mode. For example, GPT-6 might be able to run 1 million conversations in parallel, outputting one token for all conversations per forward pass. This trick works until the batch size is so large that the activations due to tokens takes up about as much memory bandwidth as the model parameters.

Concretely, we can estimate what is a good batch size for GPT-3 175B. It has 96 attention layers, each with 96x 128-dimension heads. It is typically run with 16-bit precision floating point numbers. **For a single token, how many megabytes would all the floating point activations cost?**

The model itself has 175 billion 16-bit floating point parameters, taking up about 350 GB. **How many tokens do we need to put into a batch, before the activations occupy the same amount of memory as the model parameters?**

Solution: A single token would cost $96 \times 96 \times 128$ floating point activations, or about 2.4 MB.

So in order for the activations of tokens to occupy the same memory as the GPT-3 parameters itself, we need about 350 GB / 2.4 MB = 0.15 million tokens.

If we count the optimizer states for the model during training, then GPT-3 takes up 4x350 GB = 1.4 TB, and so we need about 0.6 million tokens.

This explains why during training, the batch sizes of the largest models typically are on the order of 1 million tokens. It also shows why there is a natural drive towards scale – only the largest companies can expect to regularly run 0.2 million chats simultaneously.

(h) (**Training cost**) How much money does compute cost? We can work through an example using the current standard computing hardware: Nvidia H200 GPU (100 GB VRAM version).

The most important specifications are:

- Unit price: 30000 USD.
- Rental price: 1.50 USD/hr.
- Speed: 1.98 petaFLOP/s = 1.98e15 FLOP/s.
- Power: 0.7 kiloWatt.
- Memory bandwidth: 4800 GB/s.

In the literature about the largest AI models, the training cost is often reported in units of “petaFLOP-day”, which is equal to 1 petaFLOP/second x 1 day. **How many FLOP is 1 petaFLOP-day? What**

is the equivalent number of H200-hour? If we were to buy 1 petaFLOP-day of compute with rented H200 GPU, how much would it cost?

Solution: 1 petaFLOP-day = 1 petaFLOP/second x 1 day = 1e15 FLOP/s * 8.64e4 s = 8.64e19 FLOP. Since 1 H200 can run at 1.98 petaFLOP/s, getting 1 petaFLOP-day requires us to run 1/1.98 H200s for 1 day, or 12.12 H200-hours. At the price of 1.50 USD/hr, it would cost 18.18 USD.

The largest model of GPT-3 cost 3640 petaFLOP-days to train (according to Table D.1 of the report). How much would it cost if it were trained with an H200? How much money does it cost to train our hypothetical GPT-6?

In reality, the GPU cannot be worked to their full speed, and we only use about 30% of its theoretical peak FLOP/sec (so for example, for H200, we only get 0.594 petaFLOP/s, instead of 1.98 petaFLOP/s).[4] For this question, we assume that the utilization rate is 100%.

Also, we are assuming the training happens in one go, without hardware failures, divergences, and other issues requiring restart at a checkpoint. There is not much published data from large-scale training, but the OPT-175B training run (described later) took 3 months to complete, but would have taken only 33 days if there were no need for the many restarts. This suggests that the restarts would increase the computing cost by 2x to 3x.

Solution: GPT-3 should cost about 18.18 USD * 3640 = 66175.20 USD.

The hypothetical GPT-6 cost 1e30 FLOP to train = 5e14 H200-seconds = 1.4e11 H200-hours = 2.1e11 USD or 210 billion USD

Assuming 30% utilization rate, we should expect the training to cost about 701 billion USD. And if we are to account for restarts, it suggests on the order of 1 trillion USD.

It is rumored that GPT-4 cost 200,000 petaFLOP-days to train. That gives us a price estimate of 3.6 million USD. Correcting for the typical utilization rate of only 30%, that would give us a price estimate of 20 million USD. And if we are to account for restarts, it suggests 40 – 60 million USD.

Oh, and if you want some kind of official confirmation? OpenAI's CEO Says the Age of Giant AI Models Is Already Over | WIRED

> At the MIT event, Altman was asked if training GPT-4 cost $100 million; he replied, "It's more than that."

For context, here are the costs of *development* of various items[5]:

- iPhone 1: 150 million USD.
- A typical 5 nm chip: 0.5 billion USD.
- Airbus A380: 18 billion USD.[6]
- Three Gorges Dam: 250 billion CNY, or about 30 billion USD.
- Manhattan Project: 24 billion USD (2021 level)
- Apollo Program: 178 billion USD (2022 level)

Comment on the cost of our hypothetical GPT-6. Is it on the order of a large commercial actor like Google, or a state actor like China?

Solution: The cost is on the order of 100 billion USD. Larger models, as those contemplated by AGI projects in Google, OpenAI, etc, have historically been undertaken by state actors only.

[4]The utilization rate of 30% is according to EpochAI.

[5]Sorry, not adjusted for inflation to the same year, but they are roughly in the range of 2000 – 2020 USD.

[6]Table 4.3 of Bowen, John T. The economic geography of air transportation: space, time, and the freedom of the sky. Routledge, 2010.

Here is another way to estimate. Building a large AI model, according to accountancy, would be investment into a production machine, and thus part of "capital expenditure" ("Capex"). We can find out what proportion is 100 billion USD in their total capex. I looked it up here.

- In 2024, Microsoft has a capex of about 65 billion USD.
- Google has about 44 billion USD.
- Meta, 37.
- Amazon, 83.

In short, training something like GPT-6 would cost a lot, more than 100% of total capex, even by the standards of the largest companies.

In order to train even larger AI models, those AI models *must* enter production. They must become a productive member of society, otherwise the company wouldn't have the money to train them.

OpenAI and Broadcom announce strategic collaboration to deploy 10 gigawatts of OpenAI-designed AI accelerators and OpenAI and NVIDIA announce strategic partnership to deploy 10 gigawatts of NVIDIA systems are just two 10 gigawatt data centers that OpenAI plans to use. In perfect conditions, each data center would be able to run $10 \times 10^9 / 0.7 \times 10^3 \approx 14$ million H200 chips.

The largest companies, like OpenAI have GPUs on the order of tens of millions of H200s. What is the wall clock hour of training GPT-6, assuming you have 30 million H200s, perfect utilization, and no interruptions?

Solution: 1.4e11 H200-hours / 30000000 H200s = 194 days. A bit more than half a year.

(i) (**the difficulty of large-scale training**) This is **not a problem**, but some interesting reading of some "stories from the trenches".

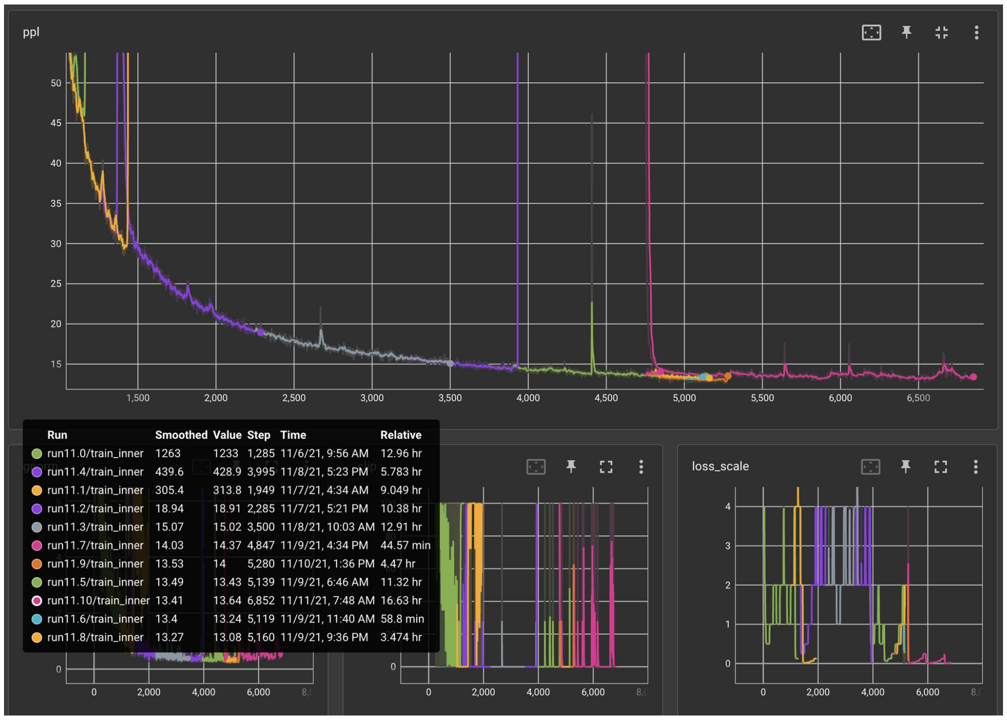
Large models do end up diverging many times during training runs, requiring restarts. Sometimes the divergence is so bad that the whole training run must be started from scratch. Sometimes the convergence is too slow, requiring fixes to the optimizer and learning rate schedules, etc.

All together, we should expect the failures, restarts, deadends... to triple the cost at least, to ~1 billion USD.

In 2021, a team of 5 engineers from Meta trained a LLM with 175 billion parameters, in 3 months, using 1024 80GB A100 GPUs from. Excluding all the divergences, hardware failures, and other issues that caused lost progress, the final model would have taken about 33 days of continuous training.

They have kept journals during their training. This is now published at metaseq/projects/OPT/chronicles at main · facebookresearch/metaseq · GitHub. You can see how difficult it is to train a large model. Selected quotes:

> These notes cover ~90 restarts over the course of training the lineage of this current model (experiments 12.xx).
>
> Found issues with the new dataset where perplexity was unreasonably low... After applying as much regex-ing as we could to salvage the dataset, we relaunched another set of experiments to test LPE (experiments 20-29) on the new dataset. We didn't have time to retrain a new BPE on the final dataset, so we fell back to using the GPT-2 BPE.
>
> From experiment 11.4 onward, we saw grad norm explosions / loss explosions / nans after a couple hundred updates after each restart, along with extremely unstable loss scales that would drop to the point of massively underflowing. We started taking more and more drastic actions then, starting with increasing weight decay to 0.1, lowering Adam beta2 to 0.95, lowering LR again, until finally by experiment 11.10 we hot-swapped in ReLU and also switched to a more stable MHA calculation (noting that the x**3 term in GeLU might be a source of instability with FP16).
>
> On November 11, we started training our 12.00 experiment with all of these changes, and since then, the only restarts we've had to make were all related to hardware issues (missing GPUs on instances, training randomly hanging after including a new node, ECC errors, partial checkpoint upload after hardware error, CUDA errors, NCCL errors, etc.).

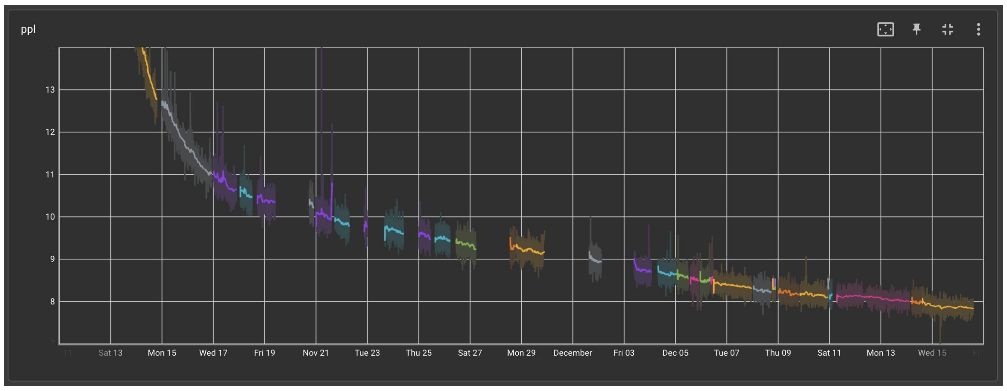


	Run	Smoothed	Value	Step	Time	Relative
🟢	run11.0/train_inner	1263	1233	1,285	11/6/21, 9:56 AM	12.96 hr
🟣	run11.4/train_inner	439.6	428.9	3,995	11/8/21, 5:23 PM	5.783 hr
🟠	run11.1/train_inner	305.4	313.8	1,949	11/7/21, 4:34 AM	9.049 hr
🟣	run11.2/train_inner	18.94	18.91	2,285	11/7/21, 5:21 PM	10.38 hr
●	run11.3/train_inner	15.07	15.02	3,500	11/8/21, 10:03 AM	12.91 hr
●	run11.7/train_inner	14.03	14.37	4,847	11/9/21, 4:34 PM	44.57 min
🟠	run11.9/train_inner	13.53	14	5,280	11/10/21, 1:36 PM	4.47 hr
🟢	run11.5/train_inner	13.49	13.43	5,139	11/9/21, 6:46 AM	11.32 hr
●	run11.10/train_inner	13.41	13.64	6,852	11/11/21, 7:48 AM	16.63 hr
🔵	run11.6/train_inner	13.4	13.24	5,119	11/9/21, 11:40 AM	58.8 min
🟠	run11.8/train_inner	13.27	13.08	5,160	11/9/21, 9:36 PM	3.474 hr

Replacement through the cloud interface can take hours for a single machine, and we started finding that more often than not we would end up getting the same bad machine again.

There were also issues with blob store when downloading 1.6TB of a single model checkpoint (992 files, each ~1.7GB) on restarts, at which point the downloads themselves would start hanging nondeterministically, which then delayed training recovery even further.

We managed to hit our top three record long runs of the experiment these past two weeks, lasting 1.5, 2.8, and 2 days each! If we were to look at only the runs that have contributed to pushing training further and plot training perplexity against wall clock time, we get the following [The breaks are due to the Thanksgiving holiday]:



(j) (**Inference cost**) Inference cost a lot less money than training, but it's still a substantial cost. For Transformer language models, it costs about 2 FLOPs per parameter to infer on one token.

If we estimate that GPT-5 has 1 trillion parameters, how many FLOPs would it take to infer on 1 million tokens? How much money would it cost if it were run with a H200?

Solution: 1 trillion * 1 million * 2 = 2e18 FLOPs. Now one H200-hour is 1.08e18 FLOPs, so that is about 2.78 USD.

The price offered by OpenAI is 120 USD per 1 million tokens, so it's a very profitable business... but see next question.

The price offered by OpenAI is 120 USD per 1 million tokens. Assuming that GPT-5 cost 1 billion USD to develop and train, **how many tokens must be sold, just to recoup the cost?** Assuming each English word cost about 1.4 tokens, and each essay is 1000 words, **how many essays would it be equivalent to?**

Solution: Since the cost is almost negligible (1/40 of the revenue), we can pretend it's all profit. So, the profit is 117 USD per 1 million tokens. We need to recoup 1 billion USD, so we need (1 billion / 117) * 1 million = 8.5e15 tokens, or 8.5e15/ 1400 = 6.1 billion essays.

About one essay per person on earth, or 18 essays per person in America... is that too much to ask?

(k) (**Energetic cost**) The Landauer limit states that the cost of erasing one bit of information is $E = k_B T \ln 2$, where k_B is the Boltzmann constant, and T is the temperature of the computing machinery. At room temperature, $T = 300K$, giving us $E = 3 \times 10^{-21} J$.

Now, one FLOP is a floating point operation, meaning that you start with two 32-bit objects and end up with a single 32-bit object. You start with 64 bits and end up with just 32 bits, and so you lose 32 bits. So by the Landauer limit, the minimal energetic cost is $32 k_B T \ln 2$.

Given this, what is the minimal energy required for performing one FLOP? What is the minimal power (in Watts) required to perform 1980 TFLOP/sec, as in a H200 GPU? Compare this to the actual value of 700 Watts.

Solution: The energy per FLOP is $E_{FLOP} = 32 \times 3 \times 10^{-21} J = 10^{-19} J$. At 1980 TFLOP/sec, we need $P_{H200} = 1.98 \times 10^{15} E_{FLOP}/sec = 1.98 \times 10^{-4} W$. The actual value of 700 Watts is 3.5 million times more than the theoretical minimum.

There is still plenty of room at the bottom!

For context, the equivalent FLOP/sec for the human brain is controversial, but a brief scan through the review article says that it should be about 1e18 FLOP/sec. The energetic power of the brain is well-known: about 30 Watts. This means the brain is about 12000x more energy-efficient than H200 GPU, but still 300x less efficient than the theoretical minimum.

(l) (**Environmental cost**) According to "Carbon emissions and large neural network training" (Patterson et al, 2021), the carbon emission of training GPT-3 is 552 tCO2. According to a 2021 poll of climate economists, 1 tCO2 emission should cost somewhere between 50 and 250 USD. Let's take their geometric average of 112 USD.

When GPT-3 was trained, the standard chip was the NVIDIA A100, which was slower and much less energy-efficent. We estimate that GPT-3 training on A100 GPUs cost 6 million USD. If we add all the tCO2 cost to the training of GPT-3, how much more expensive would it be? Compare that with its A100-GPU cost of training.

Solution: 112 * 552 = 62k USD.

We estimate the A100 GPU cost of training to be around 6 million USD. So fully carbon-pricing the emission would make it 1% more expensive.

Notice that we earlier calculated how much training GPT-3 would have cost using NVIDIA H200 GPUs and it was only 66175.20 USD. This amount is just above the carbon-pricing that we estimated. Modern hardware makes a big difference.

Generally, adding in the tCO2 cost of training the largest models increase the cost by about 1 to 2 % (I have computed it in several ways). Since the global GDP increases by about 2% a year, this shows that fully accounting for the carbon cost of AI would delay it by perhaps 1 year.

To put the number in another context, compare it with some typical American food. According to Our World in Data, it cost about 50 kg of CO2 emission per 1 kg of beef.

Assuming each burger ("quarter pounder") contains 1/4 pound (113 grams) of beef. **How many burgers would be equivalent to the CO2 emission of GPT-3?**

Solution: 113 grams of beef emits about 5.6 kg of CO2, so GPT-3 emits about (552 ton)/(5.6 kg) = 90,000 burgers. It is about the same amount as 250 people eating one burger everyday, for a whole year.

6. Soft-Prompting Language Models

You are using a pretrained language model with prompting to answer math word problems. You are using chain-of-thought reasoning, a technique that induces the model to "show its work" before outputting a final answer.

Here is an example of how this works:

```
[prompt] Question: If you split a dozen apples evenly among yourself
and three friends, how many apples do you get? Answer: There are 12
apples, and the number of people is 3 + 1 = 4. Therefore, 12 / 4 = 3.
Final answer: 3\n
```

If we were doing hard prompting with a frozen language model, we would use a hand-designed [prompt] that is a set of tokens prepended to each question (for instance, the prompt might contain instructions for the task). At test time, you would pass the model the sequence and end after "Answer:" The language model will continue the sequence. You extract answers from the output sequence by parsing any tokens between the phrase "Final answer: " and the newline character "\n".

(a) Let's say you want to improve a frozen GPT model's performance on this task through soft prompting and training the soft prompt using a gradient-based method. This soft prompt consists of 5 vectors prepended to the sequence at the input — these bypass the standard layer of embedding tokens into vectors. (Note: we do not apply a soft prompt at other layers.) Imagine an input training sequence which looks like this:

```
["Tokens" 1-5: soft prompt] [Tokens 6-50: question]
[Tokens 51-70: chain of thought reasoning]
[Token 71: answer] [Token 72: newline]
[Tokens 73-100: padding].
```

We compute the loss by passing this sequence through a transformer model and computing the cross-entropy loss on the output predictions. If we want to train the soft-prompt to output correct reasoning and produce the correct answer, **which output tokens will be used to compute the loss?** (Remember that the target sequence is shifted over by 1 compared to the input sequence. So, for example, the answer token is position 71 in the input and position 70 in the target).

Solution: We include output tokens 50-71 (the chain of thought, answer, and newline) in the loss.

In more depth:

- Output tokens 1-4 - soft prompt: there is no ground truth output for these tokens, so we cannot train a loss with them.

- Output tokens 5-49 - question: we technically have ground-truth here we could use for the loss, but there is no point in training the model to be good at generating questions since at test-time it will be given the question and only needs to output the reasoning and answer.
- Output tokens 50-69 - reasoning: This is important to include if we want the model to learn to output reasoning. (There may be multiple forms of valid reasoning which could go here, but we still can train on the reasoning examples in our training set.)
- Output token 70 - answer: This is important to include in the loss.
- Output token 71 - newline: We include this in the loss too. If the model is not trained to output a newline after it finishes producing the answer, then when we parse the answers we will get extra tokens at the end.
- Output tokens 72-100: This is padding, so we don't need to include this.

(Note that the indices provided were for **this particular example**. Other training examples in the dataset will have different length questions, reasoning, and answers.)

(b) Continuing the setup above, **how many parameters are being trained in this model?** You may write this in terms of the max sequence length S, the token embedding dimension E, the vocab size V, the hidden state size H, the number of layers L, and the attention query/key feature dimension D.

Solution: The gradient updates will be applied to the 5 vectors that constitute the soft prompt. Everything else in the model will be frozen.

There are 5 vectors of size E, so 5E total.

(c) **Mark each of the following statements as True or False and give a brief explanation.**

(i) If you are using an autoregressive GPT model as described in part (a), it's possible to precompute the representations at each layer for the indices corresponding to prompt tokens (i.e. compute them once for use in all different training points within a batch).

Solution: True, since the masking is autoregressive, the prompt representations will be the same for each data point.

Note that during training, the fact that the inputs and representations are the same also means that the linear mapping (induced by the chain rule) that lets gradients get pulled back onto the parameters is also locked into place. Of course, the actual updates to parameters from those gradients have to be computed for each training point in the batch so that they can be combined to do an update, but this gradient update is not a representation at any layer.

(ii) If you compare the validation-set performance of the *best possible* K-token hard prompt to the *best possible* K-vector soft prompt, the soft-prompt performance will always be equal or better.

Solution: True, since the embedding of the best possible hard prompt is contained within the set of soft prompts, the best soft prompt must be at least as good.

(iii) If you are not constrained by computational cost, then fully finetuning the language model is always guaranteed to be a better choice than soft prompt tuning.

Solution: False, full finetuning, especially on a small dataset may hurt generalization and encourage overfitting. In practice, this tends not to happen often because of the regularizing effects of gradient-descent training in the context of so much overparameterization. What typically happens with full fine-tuning is that only a low-rank update actually ends up happening.

(iv) If you use a dataset of samples from Task A to do prompt tuning to generate a soft prompt which is only prepended to inputs of Task A, then performance on some other Task B with its own soft prompt might decrease due to catastrophic forgetting.

Solution: False, Task B performance is unaffected since the core model's parameters don't change. Those are frozen.

(d) Suppose that you had a family of related tasks for which you want use a frozen GPT-style language model together with learned soft-prompting to give solutions for the task. Suppose that you have substantial training data for many examples of tasks from this family. **Describe how you would adapt a meta-learning approach like MAML for this situation?**

(HINT: This is a relatively open-ended question, but you need to think about what it is that you want to learn during meta-learning, how you will learn it, and how you will use what you have learned when faced with a previously unseen task from this family.)

Solution: The spirit of MAML is to train with an eye towards how you will act at test time. At test time, when faced with a previously unseen task from this family, we will initialize our soft-prompt to the known initialization. Then, we will use the training data to fine-tune the soft prompt to improve model performance by gradient descent. We'll see how we do on some held-out set.

Consequently, the thing we want to learn during meta-learning is the initial condition for the prompt.

How we could do it is to first take our training data for each task and split it into a train-train set and a held-out set. Then, we start with a random initialization for the parameters (we will note some variations later) defining the soft prompt start. Now, we begin to sample tasks where we:

- Initialize the soft prompt to our current soft-prompt-start and then;
- take a few steps of SGD using the task-specific loss and a random sampling of the training data.
- Having done that, we now use the held-out data to compute a gradient for the loss with respect to the intialization.
- We update the intialization of the soft-prompt-start by taking a step along this gradient
- Repeat with a freshly sampled task.

This updates the soft prompt initialization to an initialization that will hopefully work better after a few steps of training.

There are many variations on this basic MAML approach that can be done.

- We could use a mini-batch of tasks instead of sampling a single task.
- We could also allow our tasks to spawned partially trained copies and persist across meta-training iterations. (Essentially, after a task is used for a step of meta-training we flip a coin to decide whether to keep it or kill it. If we keep it, it gets added back to our pool of tasks with its now updated initialization intact. When it gets sampled again, it keeps its state but the gradient calculated using that is used to update the global soft-prompt-start — this incorporates ideas from "first-order MAML" and the partial spirit of approaches like REPTILE. Because only the original task can spawn copies, and each of the copies has a Poisson death in the training process, the expected number of tasks in our training pool stays bounded.)
- We could also combine with global pre-training where we use a fraction of the tasks to train a warm initializaiton using standard pre-training approaches (no MAML style gradients over unrolled gradient updates).
- We could also try to combine with k-means (or EM) style thinking and keep a number of initializations for the soft prompt in play. Then, when it comes to applying the gradient updates, we could have them softmaxed by the performance of that particular initialization on the held-out set. (This basically trains up a small library of initial soft-prompts with the expectation that one of them will hopefully be in the right neighborhood to rapidly train up a good soft-prompt for a new task.)
- You could further try to combine with human-designed/conjectured "hard prompts" for each task with the idea that upon encountering a new task, a human could be asked to guess a hard prompt for it. Then, the initialization of the soft-prompt before tuning could be a convex combination of

the human hard-prompt with the MAML-learned good initialization. This can also be combined with a regularizer that prevents the finally learned soft-prompt from wandering too far away from the embedding of the human-provided hard prompt for the task.

The advantage of learning deep learning in the context of a solid grasp of machine learning fundamentals is that you as students should be able to generate many such variations yourselves. Not all of them are going to work well and empirical testing is always required to make choices, but before you can try stuff our empirically you have to be able to come up with things to try.

7. TinyML - Quantization and Pruning.

(This question has been adapted with permission from MIT 6.S965 Fall 2022)

TinyML aims at addressing the need for efficient, low-latency, and localized machine learning solutions in the age of IoT and edge computing. It enables real-time decision-making and analytics on the device itself, ensuring faster response times, lower energy consumption, and improved data privacy.

To achieve these efficiency gains, techniques like quantization and pruning become critical. Quantization reduces the size of the model and the memory footprint by representing weights and activations with fewer bits, while pruning eliminates unimportant weights or neurons, further compressing the model.

(a) Please complete `pruning.ipynb`. Then answer the following questions.

i. In part 1 the histogram of weights is plotted. **What are the common characteristics of the weight distribution in the different layers?**

Solution: The distribution of weights is centered on zero with tails dropping off quickly.

ii. **How do these characteristics help pruning?**

Solution: Weights that are close to 0 are the ones that are lease important, which helps alleviate the impact of pruining on model accuracy.

iii. After viewing the sensitivity curves, please answer the following questions. **What's the relationship between pruning sparsity and model accuracy? (i.e., does accuracy increase or decrease when sparsity becomes higher?)**

Solution: The relationship between pruning sparsity and model accuracy is inverse. When sparsity becomes higher, the model accuracy decreases.

iv. **Do all the layers have the same sensitivity?**

Solution: No.

v. **Which layer is the most sensitive to the pruning sparsity?**

Solution: The first convolution layer ('backbone.conv0').

vi. (Optional) After completing part 7 in the notebook, please answer the following questions. **Explain why removing 30 percent of channels roughly leads to 50 percent computation reduction.**

Solution: The most layers in the given neural network are convolution layers. For convolution, $\#MACs = c_o \cdot h_o \cdot w_o \cdot k_h \cdot k_w \cdot c_i$. Removing 30 percent of channels leads to $\#MACs' = (1-0.3) \cdot c_o \cdot h_o \cdot w_o \cdot k_h \cdot k_w \cdot (1-0.3) \cdot c_i = 0.49\#MACs$. Therefore, removing 30 percent of channels roughly leads to $1 - 0.49 \approx 50\%$ computation reduction.

vii. (Optional) **Explain why the latency reduction ratio is slightly smaller than computation reduction.**

Solution: The latency mostly depends on both computation time and data movement time. For convolution, removing 30 percent of channels does not reduce the size of activation by half, and thus the data movement time does not reduce by half. For layers including BatchNorm and ReLU, the computation time is linear to the size of input, and thus the latency of these layers does not reduce by half. In addition to these workload, neural network inference also contains some overhead that does not depend on the size of the input, such as function call. Such overhead can not be reduced when removing channels.

viii. (Optional) **What are the advantages and disadvantages of fine-grained pruning and channel pruning? You can discuss from the perspective of compression ratio, accuracy, latency, hardware support (*i.e.*, requiring specialized hardware accelerator), etc.**

Solution: The advantages of fine-grained pruning are that it can achieve higher compression ratio and higer accuracy easily. The disadvantages are that it requires specialized hardware support to gain actual latency reduction.

The advantages of channel pruning are that it can easily achieve lower latency on general-purpose hardware. The disadvantage is the lower compression ratio when maintaining the model accuracy.

ix. (Optional) **If you want to make your model run faster on a smartphone, which pruning method will you use? Why?**

Solution: If the mobile phone hardware supports fine-grained pruning, I will use fine-grained pruning since it can prune the model more aggressively and reduce the model size significantly. Otherwise, I will use channel pruning because it is much easier to gain speedup for channel pruning and it is faster than fine-grained pruning on general-purpose hardware.

(b) Please complete `quantization.ipynb`. Then answer the following questions.

i. After completing K-means Quantization, please answer the following questions. **If 4-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

Solution: 16.

ii. **If n-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

Solution: 2^n colors.

iii. After quantization aware training we see that even models that use 4 bit, or even 2 bit precision can still perform well. **Why do you think low precision quantization works at all?**

Solution:

1). We can think of quantization as noise. For example, if we were to truncate the decimal in a floating point value, the small fluctuation in value can be seen as noise. As seen before, deep neural networks are good at handling noisy inputs and can still infer patterns from samples with noise.

2). General structure is still maintained after quantization. For example, if we think see a 1 bit image (black or white), we can still make out the shapes and object in that image.

3). When we train with dropout we are essentially using a large ensemble of smaller models. Dropout is essentially creating a source of redundancy in nodes. If we were to run dropout during inference we wouldn't expect it to drop severely in performance. The actual computation during inference is a kind of average, rather than just information from a single source. Quantizing the values in this before averaging can be viewed as a source of noise. We know dropout is quite robust to noise, and therefore quantizing should not affect its performance much.

iv. (Optional) Please read through and complete up to question 4 in the notebook, then answer this question.

Recall that linear quantization can be represented as $r = S(q - Z)$. Linear quantization projects the floating point range $[fp_{min}, fp_{max}]$ to the quantized range $[quantized_{min}, quantized_{max}]$.

That is to say,

$$r_{\max} = S(q_{\max} - Z)$$
$$r_{\min} = S(q_{\min} - Z)$$

Substracting these two equations, we have,

$$S = r_{\max}/q_{\max}$$
$$S = (r_{\max} + r_{\min})/(q_{\max} + q_{\min})$$
$$S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$$
$$S = r_{\max}/q_{\max} - r_{\min}/q_{\min}$$

Which of these is the correct result of subtracting the two equations?

Solution: $S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$

v. (Optional) Once we determine the scaling factor S, we can directly use the relationship between $r_{\min}$ and $q_{\min}$ to calculate the zero point Z.

$$Z = \text{int}(\text{round}(r_{\min}/S - q_{\min})$$
$$Z = \text{int}(\text{round}(q_{\min} - r_{\min}/S))$$
$$Z = q_{\min} - r_{\min}/S$$
$$Z = r_{\min}/S - q_{\min}$$

Which of these are the correct zero point?

Solution: $Z = \text{int}(\text{round}(q_{\min} - r_{\min}/S))$

vi. (Optional) After finishing question 9 on the notebook, **please explain why there is no ReLU layer in the linear quantized model.**

Solution: Before deployment, Convolution and BatchNorm are fused. During activation quantization, the scale factor S and zero point Z are determined using statics of ReLU output activations. That is, $r_{\min} = 0$. Therefore, the following layer's input activation quantization will naturally clamp the input by 0, which has the same effect as ReLU.

vii. (Optional) After completing the notebook, **please compare the advantages and disadvantages of k-means-based quantization and linear quantization.** You can discuss from the perspective of accuracy, latency, hardware support, etc.

Solution: K-means-based quantization is more flexible since the quantization centroids can be arbitrary floating point values to minimize the quantization error. Therefore, K-means-based quantization can maintain higher accuracy with lower bit widths. However, k-means-based quantization only reduces the storage and still requires floating-point computation. Moreover, decoding the codebook (i.e., memory access on the lookup table) is required before general matrix multiplication on general-purpose hardware like GPUs.

Linear quantization is more hardware-friendly since all weights and activations are stored in integers, and almost all arithmetic operations are integer-based. Therefore, linear quantization can be directly exploited by modern GPUs and CPUs. However, linear quantization requires that quantization centroids are (uniformly distributed) integers, and thus it is much more difficult to maintain accuracy with lower bit-width linear quantization.

8. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Naman Jain.
- Anant Sahai.
- Patrick Mendoza.
- Sultan Daniels.
- Kevin Frans.
- Yuxi Liu.
- Olivia Watkins.
- Yujun Lin.
- Ji Lin.
- Zhijian Liu.
- Song Han.
- Liam Tan.
- Romil Bhardwaj.