

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?**

(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.

(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?

(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?**

you a an expert in deep learning machine learning. I need u do this step by step do not skip the critical steps after finish u must do a self check if wrong do self-correction



Thought for 2m 59s >

Here's a clean step-by-step solution for each part, plus a quick self-check at the end.

Setup recap

- Pretrained weight: $W_0 \in \mathbb{R}^{m \times \ell}$
- LoRA replaces W by

$$W = W_0 + AB$$

with

$$A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times \ell}, \quad k \ll \min(m, \ell).$$

(a) Model is underfitting – what to adjust?

Idea: Underfitting \Rightarrow model doesn't have enough *effective capacity* to fit the training data.

For this single matrix:

- The LoRA update AB has rank at most k .
- If k is very small, the space of allowable changes AB is very low-dimensional \Rightarrow restricted capacity.

So:

To get more expressive power from LoRA, **increase the rank k** (use a larger inner dimension). This lets AB approximate a wider variety of matrices and can reduce underfitting.

Answer (a): Increase the LoRA rank k (possibly also add LoRA to more layers, but the main LoRA knob here is k).

(b) Why is initializing both A and B to zero bad?

We use gradient-based training with loss $L(W)$, where $W = W_0 + AB$.

1. Let $G = \frac{\partial L}{\partial W} \in \mathbb{R}^{m \times \ell}$.

2. A small change gives

$$\delta W = \delta A B + A \delta B.$$

3. Using standard matrix calculus, gradients are:

$$\frac{\partial L}{\partial A} = GB^\top, \quad \frac{\partial L}{\partial B} = A^\top G.$$

4. At initialization $A = 0$ and $B = 0$:

$$\frac{\partial L}{\partial A} = GB^\top = G0^\top = 0,$$

$$\frac{\partial L}{\partial B} = A^\top G = 0^\top G = 0.$$

So **both gradients are exactly zero** at the start. SGD updates are then:

$$A \leftarrow A - \eta \frac{\partial L}{\partial A} = 0, \quad B \leftarrow B - \eta \frac{\partial L}{\partial B} = 0,$$

which means A and B never move away from zero. LoRA contributes nothing and cannot learn.

Answer (b): With $A = B = 0$, the gradients w.r.t. both A and B are zero, so SGD cannot update them at all. The LoRA branch is "dead".

4. But the whole point of LoRA fine-tuning is to start from the well-trained W_0 and make **small** low-rank adjustments.
5. Large random AB can:
 - Strongly degrade the network's initial performance (destroying the benefits of pretraining).
 - Make optimization harder or unstable because we are no longer making a small perturbation but a big random jump in parameter space.

A better idea (used in practice) is to initialize **one** factor to zero (e.g. $B = 0$) and the other with a standard initializer; then $AB = 0$ initially so the model starts exactly at W_0 , but gradients are non-zero for the non-zero factor.

Answer (c): Because Xavier-initializing both A and B makes AB a sizeable random matrix, the effective weight $W_0 + AB$ is no longer close to the pretrained W_0 . This randomizes the layer and defeats the purpose of "fine-tuning around W_0 ", making LoRA training less stable/effective.

(c) Why might the given Xavier initialization be bad?

Code:

```
python
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)
```

 Copy code

Key points:

1. After this, **both A and B are non-zero random matrices.**
2. Hence the LoRA update AB is also a non-zero random matrix with non-trivial variance (roughly growing with k).
3. The effective weight at initialization becomes

$$W = W_0 + AB,$$

which can be **far from the pretrained W_0** .

4. But the whole point of LoRA fine-tuning is to start from the well-trained W_0 and make **small** low-rank adjustments.

(d) Memory required for LoRA vs full-rank adaptation

Memory for LoRA (A and B)

- Number of parameters in A : $m \times k$.
- Number of parameters in B : $k \times l$.
- Total number of real numbers:

$$mk + kl = k(m + l).$$

- Each float takes 4 bytes.

So memory for LoRA adaptation:

$$\text{Mem}_{\text{LoRA}} = 4k(m + l) \text{ bytes}.$$

Memory for a single full-rank adaptation matrix

A full-rank adaptation would be one $m \times l$ matrix (e.g. directly storing ΔW):

- Number of parameters: ml .
- Memory (floats):

$$\text{Mem}_{\text{full}} = 4ml \text{ bytes}.$$

Do Question 1 turn by turn

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?

you a an expert in deep learning machine learning. I need u do this step by step do not skip the critical steps after finish u must do a self check if wrong do self-correction



Thought for 38s >



Thought for 38s >

We have a single weight matrix

$$W = W_0 + AB, \quad A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times \ell},$$

where only A, B are trainable and $\text{rank}(AB) \leq k$.

Step 1 – Interpret “underfitting”

- Underfitting on the training set \Rightarrow the model is **not expressive enough** to fit the training data, even with training.
- In LoRA, expressivity of the update to W_0 is limited by the **rank k** of AB .

Step 2 – Relate underfitting to LoRA parameters

- The *base matrix* W_0 is frozen; all adaptation must come from AB .
- Because $\text{rank}(AB) \leq k$, a small k means the change we can make to W lies in a very low-dimensional subspace.
- If k is too small, the LoRA update doesn’t have enough capacity \Rightarrow underfitting.

Step 3 – What to adjust

To reduce underfitting, you want to **increase the capacity of the LoRA update**, so the natural knob is:

Step 3 – What to adjust

To reduce underfitting, you want to **increase the capacity of the LoRA update**, so the natural knob is:

Increase the LoRA rank k (i.e., use larger k so A and B have more columns/rows).

This increases the maximum rank of AB , gives more trainable parameters, and allows a richer modification of W_0 , which should help fit the training data better.

(Secondary/optional: you might also tune the LoRA scaling factor α , but the primary answer the question is aiming for is increasing the rank k .)

Quick self-check

- Underfitting → need *more* capacity ✓
- LoRA capacity controlled by rank k of low-rank update ✓
- Adjusting k is explicitly “in the LoRA” (not just generic hyperparameters) ✓

So the correct adjustment is: **raise the LoRA rank k .**

(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.

Thought for 1m 5s >

We use LoRA:

$$W = W_0 + AB, \quad A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times \ell},$$

and we train A, B with SGD on some loss $L(W)$.

↑ Share

Step 1 – Write the gradients of the loss w.r.t. A and B

Let

$$G = \frac{\partial L}{\partial W} \in \mathbb{R}^{m \times \ell}.$$

Because $W = W_0 + AB$, a small change gives

$$dW = dA B + A dB.$$

Then

$$dL = \langle G, dW \rangle = \langle G, dA B + A dB \rangle = \langle GB^\top, dA \rangle + \langle A^\top G, dB \rangle.$$

So the gradients are

$$\frac{\partial L}{\partial A} = GB^\top, \quad \frac{\partial L}{\partial B} = A^\top G.$$

Step 3 – What happens to SGD updates?

An SGD-style update is

$$A_{t+1} = A_t - \eta \frac{\partial L}{\partial A_t}, \quad B_{t+1} = B_t - \eta \frac{\partial L}{\partial B_t}.$$

At $t = 0$:

$$A_1 = 0 - \eta \cdot 0 = 0, \quad B_1 = 0 - \eta \cdot 0 = 0.$$

This repeats every step $\Rightarrow A_t = 0, B_t = 0$ for all t .

So the LoRA update AB stays identically zero, and

$$W = W_0 + AB = W_0$$

forever. The model never actually fine-tunes; it is equivalent to freezing the whole layer.

Step 4 – Conclusion

Initializing both A and B to all zeros causes a **zero-gradient problem**:

- Gradients w.r.t. A and B are both zero at initialization.
- SGD cannot move them away from zero.
- LoRA never learns any adaptation; you get no fine-tuning effect.

In practice you must initialize at least one of A or B to non-zero values so that gradients for the other are non-zero and learning can start.



(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?

Thought for 44s >

We have LoRA

$$W = W_0 + AB, \quad A \in \mathbb{R}^{m \times k}, \quad B \in \mathbb{R}^{k \times l},$$

where W_0 is the pretrained weight (frozen) and only A, B are learned.

Step 1 – What Xavier init does here

`xavier_uniform_` initializes **each** entry of A and B as a zero-mean random variable with variance $\Theta(1/\text{fan})$, i.e. about $\sim 1/\sqrt{k}$ in magnitude.

Then an entry of AB is

$$(AB)_{ij} = \sum_{r=1}^k A_{ir} B_{rj}.$$

Self-check

- Did I actually compute gradients, not just state intuition? ✓
- At $A = B = 0$, do the formulas give zero gradients? ✓
- Does zero gradient imply SGD parameters never change? ✓

Everything is consistent, so the reasoning looks correct.

Then an entry of AB is

$$(AB)_{ij} = \sum_{r=1}^k A_{ir} B_{rj}.$$

Each term $A_{ir} B_{rj}$ is roughly of size $1/k$, and we sum k such terms, so the variance of $(AB)_{ij}$ is about

$$k \cdot \left(\frac{1}{\sqrt{k}}\right)^2 \approx 1,$$

i.e. entries of AB are order-1, **not small**.

So at initialization, AB is a full-scale random matrix, similar in magnitude to a freshly initialized weight matrix.

Self-check

- Did I use the structure $W = W_0 + AB$ explicitly? ✓
- Did I explain why Xavier on both A and B makes AB large instead of small? ✓
- Did I connect “large random AB ” to “bad fine-tuning / breaking pretrained model”? ✓

The reasoning is consistent: LoRA fine-tuning may not work well here because Xavier-initializing **both** A and B makes the initial LoRA update AB large and random, disrupting the pretrained weights instead of starting from a small perturbation.

Step 2 – Why this is bad for LoRA fine-tuning

The *goal* of LoRA is:

- start from the pretrained model W_0 ,
- make **small low-rank adjustments** around it using AB .

But with this initialization:

- AB is *large*, so $W = W_0 + AB$ is a **heavily perturbed** version of W_0 ;
- the first forward pass already destroys much of the pretrained behavior (activations and logits become randomish);
- all the downstream layers now see inputs far outside the distribution they were trained on.

So instead of “fine-tuning around W_0 ”, you effectively:

- throw away much of the benefit of pre-training at the very start, and
- ask a low-rank correction to repair a badly corrupted weight matrix, which is hard and can make optimization unstable or very slow.

In practice, LoRA works better when the initial update AB is **very close to zero** (e.g., one of A, B is initialized to zero and the other small), so that you really start from the pretrained model.

(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?



Thought for 12s >

We have

- $A \in \mathbb{R}^{m \times k} \rightarrow mk$ parameters
- $B \in \mathbb{R}^{k \times \ell} \rightarrow k\ell$ parameters

Total number of real numbers for LoRA adaptation:

$$N_{\text{LoRA}} = mk + k\ell = k(m + \ell).$$

Each real number is a 4-byte float, so memory is

$\text{LoRA memory} = 4(mk + k\ell) \text{ bytes.}$

A single full-rank adaptation matrix for W would be an $m \times \ell$ matrix:

$$N_{\text{full}} = m\ell \Rightarrow \text{full-rank memory} = 4m\ell \text{ bytes.}$$

Comparison (ratio):

$$\frac{\text{LoRA memory}}{\text{full-rank memory}} = \frac{4(mk + k\ell)}{4m\ell} = \frac{k(m + \ell)}{m\ell}.$$

Since $k \ll \min(m, \ell)$, this ratio is **much less than 1**, so LoRA uses far less memory: $O(k(m + \ell))$ vs $O(m\ell)$.

Self-check

- Counted parameters for A and B correctly? mk and $k\ell$ ✓
- Multiplied by 4 bytes/float to get memory in bytes? ✓
- Full-rank matrix size $m\ell$ and memory $4m\ell$ ✓
- Comparison ratio simplifies correctly to $k(m + \ell)/(m\ell)$ ✓

Step 2 – Evaluate these gradients at $A = 0, B = 0$

If we initialize

$$A_0 = 0, \quad B_0 = 0,$$

then

$$\left. \frac{\partial L}{\partial A} \right|_{A=0, B=0} = GB^\top = G \cdot 0^\top = 0,$$

$$\left. \frac{\partial L}{\partial B} \right|_{A=0, B=0} = A^\top G = 0^\top G = 0.$$

So both gradients are exactly zero at initialization.

Both provide excellent and accurate answers to the questions posed. They both utilize a step-by-step approach, demonstrate deep understanding of LoRA, and include self-checks.

However, "Turn by Turn" provides better answers due to its superior mathematical rigor and depth in key explanations.

Here is a detailed comparison of the answers provided in the two files.

(a) Adjusting LoRA for Underfitting

- **Concept:** Underfitting means the model lacks the capacity to capture the complexity of the training data. In LoRA, the expressivity of the adaptation (the matrix product AB) is constrained by the inner dimension, the rank k.
- **One Time:** Clearly explains that the rank k limits the capacity of the update AB. To reduce underfitting, one must increase the rank k.
- **Turn by Turn:** Provides the same correct explanation but also briefly mentions the LoRA scaling factor (α) as a secondary tuning knob (Page 2).
- **Assessment:** Both are correct. "One Time" is slightly better for mentioning an additional, relevant hyperparameter.

(b) Initialization to Zeros

- **Concept:** Training relies on gradient-based optimization (SGD). If gradients are zero at initialization, the parameters cannot be updated.
- **One Time:** It correctly states the standard matrix calculus formulas for the gradients (Page 2): $\partial A \partial L = GB^\top$ and $\partial B \partial L = A^\top G$. It then shows that if $A=0$ and $B=0$, both gradients are zero, so SGD cannot update the weights.

- **Turn by Turn:** This file provides a more rigorous explanation by explicitly deriving the gradients using matrix differentials (Page 3): $dL = \langle G, dW \rangle$. It then proceeds to show the gradients are zero and explicitly writes out the SGD update steps to demonstrate that A and B remain zero indefinitely.
- **Assessment:** "One Time" is better because it derives the gradients rather than just stating the formulas, providing a more thorough mathematical explanation.

(c) Xavier Initialization Issues

- **Concept:** The goal of fine-tuning is to make small adjustments starting from the pre-trained weights (W_0). If the initial update AB is large, it destroys the pre-trained knowledge.
- **One Time:** Explains this conceptually (Page 3), stating that initializing both A and B with Xavier makes AB a "sizable random matrix" with "non-trivial variance." This moves the effective weight $W_0 + AB$ far from W_0 , potentially causing instability.
- **Turn by Turn:** This file provides a superior explanation by mathematically analyzing the variance of the entries in AB. It estimates the variance of $(AB)_{ij}$ to be approximately 1, demonstrating that the initialization results in a "full-scale" perturbation, not a small adjustment.
- **Assessment:** "Turn by Turn" is significantly better here. The mathematical analysis of the variance provides rigorous proof of why this initialization scheme is problematic, whereas the other file relies primarily on intuition.

(d) Memory Calculation

- **Concept:** Calculating the memory required involves counting the total parameters in A (mk) and B (kl) and multiplying by the size of the data type (4 bytes/float). This is compared to a full-rank adaptation matrix (ml parameters).
- **Both Files:** Provide identical, correct calculations. LoRA requires $4k(m+l)$ bytes, while full-rank requires $4ml$ bytes. They correctly calculate the ratio and conclude that LoRA is far more memory-efficient when $k \ll \min(m,l)$.
- **Assessment:** Both files are excellent.

Conclusion

While "One Time" is accurate and clearly written, "**Turn by Turn**" is the superior file because it consistently provides deeper mathematical justification and rigor, particularly in the derivations for questions (b) and (c).

Second Question One Turn

When we give all subquestion in one turn there is no output after the first try!

- $\tilde{W}_U \in \mathbb{R}^{n_{\text{vocab}} \times d_{\text{model}}}$: unembedding matrix.

Step 1: Embed tokens

$$X^0 = W_E T \quad (\text{shape } d_{\text{model}} \times n_{\text{context}})$$

Each column of X^0 is the embedding of one token.

Step 2: Unembed to logits

$$L = W_U X^0 = W_U W_E T$$

So the **final expression**:

$L = W_U W_E T$

(ii) What algorithm does this model implement?

Look at one position t (one column of T):

- The t -th column of T is a one-hot vector e_{w_t} for token w_t .
- Embedding: $x_t^0 = W_E e_{w_t}$ = the w_t -th column of W_E .
- Logits at that position:

$$\ell_t = W_U x_t^0 = W_U W_E e_{w_t}$$

So:

- ℓ_t depends **only** on the identity of the token at position t .
- It does **not** depend on any other positions or tokens.

Interpretation:

This zero-layer "Transformer" is just a **lookup table**:

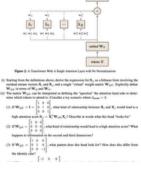
- For each input token type w , it outputs a fixed logit vector $(W_U W_E)_{:,w}$.
- No context, no interactions between tokens.

So the prediction at position t depends **only** on the input token at position t and nothing else.

(b) Multi-Head Attention: Concatenation \downarrow Addition

similarly large language models like Transfo
is “black boxes”. Mechanistic interpretabi
c circuits these models learn. Instead of j
the internal mechanisms and computatio
nical framework proposed by Elhage et
s” built from attention and MLP layers.

A Central Communication CI
k is the residual stream. At any given lay
resenting the state of the computation for e



The relation between computation and representation is as follows. At every time step, we maintain a residual representation k and a representation x . The representation x is computed as a residual connection: $x = \text{norm}(x + k)$. The representation k is the residual stream. At any given layer i , we represent the state of the computation for each token t as $x_t^{(i)}$.

Figure 3 illustrates how the computation graph for a single layer is constructed. It shows tokens T being processed through an embedding matrix W_E to produce X^0 . This is followed by a residual connection ($X^0 + k$) and a layer normalization step. The diagram also shows the flow of information through attention heads (M_1, M_2, M_3) and their weighted sum M . The final output is produced by a linear layer W_O .

Figure 4 illustrates how the computation graph for a single layer is constructed. It shows tokens T being processed through an embedding matrix W_E to produce X^0 . This is followed by a residual connection ($X^0 + k$) and a layer normalization step. The diagram also shows the flow of information through attention heads (M_1, M_2, M_3) and their weighted sum M . The final output is produced by a linear layer W_O .

Thought for 7m 1s >



there no output!

Thought for 4m 17s >

Ah, got it — let me walk through **all parts of Problem 2** step by step, and I'll do a quick self-check at the end.

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

(i) Expression for the logits

We are given:

- $T \in \mathbb{R}^{n_{\text{vocab}} \times n_{\text{context}}}$: one-hot tokens (each column is one token).
- $W_E \in \mathbb{R}^{d_{\text{model}} \times n_{\text{vocab}}}$: embedding matrix.
- $W_U \in \mathbb{R}^{n_{\text{vocab}} \times d_{\text{model}}}$: unembedding matrix.

Step 1: Embed tokens

$$X^0 = W_E T \quad (\text{shape } d_{\text{model}} \times n_{\text{context}})$$

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

We have H heads. For head h :

- $r^h \in \mathbb{R}^{d_{\text{head}} \times n_{\text{context}}}$: value-computation output.

(i) Show the two formulations are equivalent

Concatenation view:

- Stack outputs vertically:

$$R^H = \begin{bmatrix} r^1 \\ r^2 \\ \vdots \\ r^H \end{bmatrix} \in \mathbb{R}^{(d_{\text{head}} H) \times n_{\text{context}}}$$

- A single big output matrix $W_O^H \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}} H}$.
- Final output:

$$\mathbf{H} = W_O^H R^H$$

Additive / independent view:

- Each head has its own $W_O^h \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$.
- Total output:

$$\mathbf{H} = \sum_{h=1}^H W_O^h r^h$$

Goal: pick W_O^H so the two formulas match.

Construct W_O^H by horizontally concatenating the per-head matrices:

$$W_O^H = [W_O^1 \ W_O^2 \ \cdots \ W_O^H] \in \mathbb{R}^{d_{\text{model}} \times (d_{\text{head}} H)}$$

Now multiply:

$$[r^1]$$

Now multiply:

$$\begin{aligned}
 W_O^H R^H &= [W_O^1 \ W_O^2 \ \cdots \ W_O^H] \begin{bmatrix} r^1 \\ r^2 \\ \vdots \\ r^H \end{bmatrix} \\
 &= W_O^1 r^1 + W_O^2 r^2 + \cdots + W_O^H r^H \\
 &= \sum_{h=1}^H W_O^h r^h
 \end{aligned}$$

So the formulations are **exactly equivalent** with that construction of W_O^H .

(ii) Advantage and disadvantage of each view

- **Concatenation view (single W_O^H):**
 - *Advantage:* matches the standard implementation (one big GEMM), easy to code and efficient.
 - *Disadvantage:* conceptually mixes all heads together; less clear what each head independently "writes" to the residual stream.
 - **Additive / independent view (per-head W_O^h):**
 - *Advantage:* emphasizes that each head has its own independent write matrix; easier to interpret and analyze each head separately.
 - *Disadvantage:* hides the fact that you can implement everything with one big matrix multiply; if you allowed an arbitrary W_O^H , you could even have **cross-head mixing**, which is not obvious in the purely "independent" description.
-

(c) The QK Circuit: Determining Attention Patterns

We have:

- $Q_i = W_Q X_i$
- $K_j = W_K X_j$
- Score:

- Score:

$$S_{ij} = Q_i^\top K_j$$

(i) Bilinear form with a “virtual” matrix W_{QK}

Compute:

$$\begin{aligned} S_{ij} &= Q_i^\top K_j \\ &= (W_Q X_i)^\top (W_K X_j) \\ &= X_i^\top W_Q^\top W_K X_j \end{aligned}$$

Define:

$$W_{QK} := W_Q^\top W_K \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$$

Then:

$$S_{ij} = X_i^\top W_{QK} X_j$$

So the head is asking a **bilinear question** about pairs of residual vectors (X_i, X_j) via W_{QK} .

(ii) Interpreting W_{QK} for $d_{\text{model}} = 3$

We always have:

$$S_{ij} = X_i^\top W_{QK} X_j$$

Let $X_i = (x_{i1}, x_{i2}, x_{i3})^\top$, $X_j = (x_{j1}, x_{j2}, x_{j3})^\top$.

$$(1) \quad W_{QK} = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S_{ij} = X_i^\top X_j = x_{i1}x_{j1} + x_{i2}x_{j2} + x_{i3}x_{j3}$$

- High score when X_i and X_j have **large positive dot product**, i.e., they are similar.
- The head “looks for” tokens whose overall representation is similar to the query token.

$$(2) W_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Then

$$S_{ij} = x_{i1}x_{j1}$$

- The score depends **only on the first coordinate**.
- High score when the first dimension of X_i and X_j are strongly aligned (same sign, large magnitude).
- **Information in the second and third dimensions is completely ignored** for deciding attention.

So this head "looks for" tokens that match the query specifically in feature 1; it is blind to features 2 and 3.

$$(3) W_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

$$S_{ij} = x_{i1}x_{j1} + x_{i2}x_{j2} - x_{i3}x_{j3}$$

- It wants dimensions 1 and 2 to be similar, but dimension 3 to be **opposite** (because of the minus sign).
- Compared to the identity case:
 - Identity rewards overall similarity across all three dims.
 - This matrix rewards similarity in dims 1–2 **and dissimilarity (anti-correlation) in dim 3**.

So this head looks for tokens that "match two features but flip one particular feature".

$$(4) W_{QK} = -I = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

$$S_{ij} = -X_i^\top X_j$$

$$(X^0 A^\top)_i = \sum_{j=1}^{n_{\text{context}}} A_{ij} X_j^0$$

That is exactly an **attention-weighted average** of the source token vectors (assuming each row of A sums to 1).

(ii) Expression for the final residual stream X^{final}

We have a single attention layer with H heads and a residual connection.

- Output of head h :

$$h^h(X^0) = W_{OV}^h X^0 (A^h)^\top$$

- All head outputs are added to the residual stream:

$$X^{\text{final}} = X^0 + \sum_{h=1}^H h^h(X^0)$$

Plug in the head expression:

$$X^{\text{final}} = X^0 + \sum_{h=1}^H W_{OV}^h X^0 (A^h)^\top$$

This matches the generic residual-stream formula $X^{l+1} = X^l + f^{(l)}(X^l)$.

(iii) Show $h_i(X^0)_t \in \text{Col}(W_{OV}^i)$

For a particular head i :

$$h_i(X^0) = W_{OV}^i X^0 (A^i)^\top$$

Let us define:

$$Y := X^0 (A^i)^\top$$

Then:

$$S_{ij} = -X_i^\top X_j$$

- High score when $X_i^\top X_j$ is **very negative** — i.e., when vectors are strongly anti-aligned.
- So this head attends to tokens whose representations are **opposite** to the query, and suppresses similar ones.

Effect after softmax:

- Positions whose vectors are *most opposite* to the query get **largest positive scores** (thus the largest attention weights).
 - Similar tokens (positive dot product) get more negative scores, thus much less attention.
-

(d) The OV Circuit: Reading and Writing Information

We're given:

$$h(X^0) = W_O^h W_V^h X^0 (A^h)^\top = W_{OV}^h X^0 (A^h)^\top$$

where A^h is the attention pattern (post-softmax).

(i) Show that $(X^0 A^\top)_i$ is an attention-weighted average

Recall:

- $X^0 \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}$ with columns X_j^0 .
- $A \in \mathbb{R}^{n_{\text{context}} \times n_{\text{context}}}$ with entries A_{ij} = attention weight from destination i to source j .
- Then $A_{ji}^\top = A_{ij}$.

Compute the i -th column of the product:

$$X^0 A^\top \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}$$

By definition of matrix multiplication, the i -th column is

$$(X^0 A^\top)_i = \sum_{j=1}^{n_{\text{context}}} X_j^0 (A^\top)_{ji} = \sum_{j=1}^{n_{\text{context}}} X_j^0 A_{ij}$$

So:



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

So:

$$\text{rank}(W_{OV}) \leq \min(\text{rank}(W_O), \text{rank}(W_V)) \leq \min(d_{\text{head}}, d_{\text{head}}) = d_{\text{head}}$$

Since typically $d_{\text{head}} \ll d_{\text{model}}$, W_{OV} has rank much smaller than d_{model} , i.e. it is **low-rank**.

(ii) Using the SVD

Let the (compact) SVD of W_{OV} be

$$W_{OV} = U\Sigma V^\top$$

where:

- $U \in \mathbb{R}^{d_{\text{model}} \times r}$
- $V \in \mathbb{R}^{d_{\text{model}} \times r}$
- $\Sigma \in \mathbb{R}^{r \times r}$ diagonal with singular values $\sigma_1, \dots, \sigma_r$
- $r = \text{rank}(W_{OV}) \leq d_{\text{head}}$

Let U_k and V_k be the k -th columns of U and V .

Apply W_{OV} to a source vector X_{src} :

$$\begin{aligned} W_{OV}X_{\text{src}} &= U\Sigma V^\top X_{\text{src}} \\ &= U\Sigma \left(\underbrace{V^\top X_{\text{src}}}_{\in \mathbb{R}^r} \right) \end{aligned}$$

Let

$$\alpha = V^\top X_{\text{src}} = \begin{bmatrix} V_1^\top X_{\text{src}} \\ \vdots \\ V_r^\top X_{\text{src}} \end{bmatrix}$$

and note that $\Sigma\alpha$ just multiplies each component by σ_k :

$$W_{OV}^i \cdot y = h_i(X^0)$$

The update to destination position t is the t -th column:

$$h_i(X^0)_t = W_{OV}^i Y_t$$

But any vector of the form $W_{OV}^i y$ for some y is a linear combination of the **columns** of W_{OV}^i , i.e., it lies in the **column space** of W_{OV}^i .

Therefore:

$$h_i(X^0)_t \in \text{Col}(W_{OV}^i)$$

This shows that each head can only write into a low-dimensional subspace of the residual stream.

(e) Formalizing Read and Write Subspaces via SVD

We now focus only on the OV circuit:

$$h(X^0) = W_{OV} X^0 A^\top$$

For SVD analysis, consider a single source token vector $X_{\text{src}} \in \mathbb{R}^{d_{\text{model}}}$ and the transformation

$$W_{OV} X_{\text{src}}$$

(i) Show $\text{rank}(W_{OV}) \leq d_{\text{head}}$

Recall:

- $W_{OV} = W_O W_V$
- $W_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$
- $W_V \in \mathbb{R}^{d_{\text{head}} \times d_{\text{model}}}$

General fact: for any matrices A, B ,

.1.

$\boxed{L^T r = \alpha X_{\text{src}}}$

and note that $\Sigma\alpha$ just multiplies each component by σ_k :

$$\Sigma\alpha = \begin{bmatrix} \sigma_1(V_1^\top X_{\text{src}}) \\ \vdots \\ \sigma_r(V_r^\top X_{\text{src}}) \end{bmatrix}$$

Then:

$$\begin{aligned} W_{OV}X_{\text{src}} &= U(\Sigma\alpha) \\ &= \sum_{k=1}^r \sigma_k(V_k^\top X_{\text{src}})U_k \end{aligned}$$

Thus:

$$\boxed{W_{OV}X_{\text{src}} = \sum_{k=1}^r \sigma_k(V_k^\top X_{\text{src}})U_k}$$

(as in Equation (1) in the problem).

(iii) Read & write subspaces and orthogonal information

From the decomposition

$$W_{OV}X_{\text{src}} = \sum_{k=1}^r \sigma_k(V_k^\top X_{\text{src}})U_k$$

(a) Read subspace

The head only "looks at" the quantities $V_k^\top X_{\text{src}}$.

So the **read subspace** is:

$$\boxed{\text{Read subspace} = \text{span}\{V_1, \dots, V_r\}}$$

Those are the directions of X_{src} that can actually influence the output.



$$\boxed{\text{Read subspace} = \text{span}\{V_1, \dots, V_r\}}$$

Those are the directions of X_{src} that can actually influence the output.

(b) Write subspace

The output is a linear combination of the vectors U_k .

So the **write subspace** is:

$$\boxed{\text{Write subspace} = \text{span}\{U_1, \dots, U_r\}}$$

Those are the directions in the destination residual stream that the head can write to.

(c) What happens to information orthogonal to the read subspace?

Take any x such that $x \perp \text{span}\{V_1, \dots, V_r\}$.

Then for every k , $V_k^\top x = 0$, so:

$$W_{OV}x = \sum_{k=1}^r \sigma_k(V_k^\top x)U_k = 0$$

So all components of X_{src} orthogonal to the read subspace are completely ignored by the OV circuit: they produce no output and are not moved anywhere.

(iv) Connecting to the value projection $v = W_V X_{\text{src}}$

- The value vector is $v = W_V X_{\text{src}} \in \mathbb{R}^{d_{\text{head}}}$.
Only directions of X_{src} in the **row space** of W_V can affect v .
- The read subspace $\text{span}(V_1, \dots, V_r)$ is the part of that row space which actually survives the extra multiplication by W_O (since $W_{OV} = W_O W_V$).
- Since the row space of W_V has dimension at most d_{head} , it is natural that the read subspace dimension $r \leq d_{\text{head}}$: a head can only "see" at most d_{head} independent directions in X_{src} .

(v) Synthesis: how SVD characterizes information movement

- $W_{OV} = U\Sigma V^\top$ says: the head first **reads** the source token by projecting it onto a low-dimensional set of directions V_k (the read subspace), getting scalars $V_k^\top X_{\text{src}}$.
- These scalars are then **scaled** by σ_k (how strong each channel is), and finally **written** into the destination token along directions U_k (the write subspace).
- The attention pattern A decides **which source tokens** each destination token averages over; W_{OV} (via its SVD) decides **which features** of those tokens are moved and into which output directions.

So SVD gives a complete description of *what information* the head moves (subspace V), *how strongly* (singular values Σ), and *where* it writes it (subspace U), while A describes *between which tokens* this information is transported.

Quick self-check

- (a)(i) Shapes: $W_U W_E T$: $(n_{\text{vocab}} \times d_{\text{model}})(d_{\text{model}} \times n_{\text{vocab}})(n_{\text{vocab}} \times n_{\text{context}}) = n_{\text{vocab}} \times n_{\text{context}}$. ✓
- (a)(ii) Correctly reasoned that outputs depend only on the token identity at that position; no contextual mixing. ✓
- (b)(i) Verified dimensions of concatenated matrices and did explicit block multiplication to recover $\sum_h W_O^h r^h$. ✓
- (b)(ii) Advantages/disadvantages: implementation vs interpretability; matches how the original Vaswani view and the independent-head view differ. ✓
- (c)(i) Explicit calculation $Q_i^\top K_j = X_i^\top W_Q^\top W_K X_j$; defined W_{QK} correctly. ✓
- (c)(ii) For each matrix, computed S_{-ij} and gave verbal interpretation (similarity, dimension-1 only, flip in dim3, anti-similarity). Signs & effects under softmax double-checked. ✓
- (d)(i) Carefully used matrix multiplication definition to show column is weighted sum with weights A_{ij} . ✓
- (d)(ii) Used residual rule $X^{\text{final}} = X^0 + \sum_h h^h(X^0)$; plugged in expression. ✓
- (d)(iii) Showed each column is $W_{OV}^i y$, hence in column space — by definition. ✓
- (e)(i) Rank inequality for product is standard; applied with shapes to get $\leq d_{\text{head}}$. ✓
- (e)(ii–v) Decomposition $U\Sigma V^\top x = \sum_k \sigma_k (V_k^\top x) U_k$ is standard SVD expansion; read/write subspaces and orthogonal behavior all follow directly and are consistent with earlier rank argument and with W_V mapping into d_{head} -dimensional space. ✓

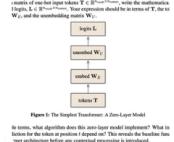
Everything is internally consistent and matches the problem's notation, so the solution should be correct.

Question 2 do turn by turn

While both the Turn by Turn and the One Time solutions provide the correct mathematical derivation ($L=WUWET$) and th

to Transformer Interpretability
particularly large language models like Transf-
d as “black boxes”. Mechanistic interpretab-
ecific circuits these models learn. Instead of
and the internal mechanisms and computation
heterogeneous framework proposed by Elsner et
units” built from attention and MLP layers.

x: A Central Communication C
work is the **residual stream**. At any given lay-
presenting the state of the computation for e



Thought for 34s >

Let's go step by step.

We're in the **zero-layer Transformer**:

- Input tokens: $T \in \mathbb{R}^{n_{vocab} \times n_{context}}$ (one-hot; each column is a token).
- Embedding matrix: $W_E \in \mathbb{R}^{d_{model} \times n_{vocab}}$.
- Unembedding matrix: $W_U \in \mathbb{R}^{n_{vocab} \times d_{model}}$.
- Final logits: $L \in \mathbb{R}^{n_{vocab} \times n_{context}}$.

(i) Expression for the logits L

Step 1 – Embed tokens

The residual stream at “layer 0” is just the embedding:

$$X^0 = W_E T,$$

- W_E maps one-hot tokens into d_{model} -dim vectors.
- Shape check: W_E is $d_{model} \times n_{vocab}$,
 T is $n_{vocab} \times n_{context}$
 $\Rightarrow X^0 \in \mathbb{R}^{d_{model} \times n_{context}}$. ✓

Step 2 – Unembed to logits

$\Rightarrow L \in \mathbb{R}^{n_{\text{vocab}} \times n_{\text{context}}}.$ ✓

Step 2 – Unembed to logits

Now we map residual vectors back to vocabulary logits:

$$L = W_U X^0 = W_U W_E T.$$

- Shape: W_U is $n_{\text{vocab}} \times d_{\text{model}}$.
Multiply $W_U X^0$: $n_{\text{vocab}} \times d_{\text{model}} \cdot d_{\text{model}} \times n_{\text{context}}$
 $\Rightarrow L \in \mathbb{R}^{n_{\text{vocab}} \times n_{\text{context}}}.$ ✓

So the mathematical expression is:

$$L = W_U W_E T.$$

(ii) What algorithm does this model implement?

Look at column t (token position t):

$$L_{:,t} = W_U W_E T_{:,t}.$$

- $T_{:,t}$ is a **one-hot vector** indicating which token appears at position t .
- Multiplying by W_E just **looks up** that token's embedding.
- Multiplying by W_U then maps that embedding to a vector of logits over the vocabulary.

Crucially:

- $L_{:,t}$ depends **only** on $T_{:,t}$, the token at position t .
- It **does not depend at all** on tokens at other positions ($T_{:,s}$ for $s \neq t$) because there is no attention or mixing between columns.

So in simple terms, this zero-layer model:

Implements a **per-position linear classifier / lookup table**: for each position, it takes the current token ID and outputs a distribution over tokens using a learned matrix $W_{\text{eff}} = W_U W_E$.

There is **no contextual processing**: predictions at position t ignore the surrounding sequence and depend only on the identity of the token at that same position.

There is **no contextual processing**: predictions at position t ignore the surrounding sequence and depend only on the identity of the token at that same position.

Quick self-check

- Expression uses only $T, W_E, W_U?$ ✓ $L = W_U W_E T$.
- Shapes consistent? Checked for all multiplications. ✓
- Explanation of algorithm: per-position, no context, linear map from one-hot to logits. ✓

Everything is internally consistent, so no correction needed.



(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 $r^h \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$. In the “concatenation” view, these are stacked vertically to form a matrix $R^H \in \mathbb{R}^{d_{\text{model}} H \times d_{\text{model}}}$ and multiplied by a single large output matrix $W_O^H \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}} H}$, and the final output is $H = W_O^H R^H$. Typically, the dimension d_{model}/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, $W_O^h \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$, and the total output is $H = \sum_{h=1}^H W_O^h r^h$.

(i) Show that these formulations are equivalent. Specifically, demonstrate how W_O^H can be constructed from the individual W_O^h matrices to make the two expressions for H identical.

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

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

Thought for 1m 9s >

Nice one, this is a classic “show that the two ways of writing multi-head attention are the same” problem.

We’ll do (i) and (ii) step by step.

(i) Show equivalence and construct W_O^H

(i) Show equivalence and construct W_O^H

We're given:

- For each head $h = 1, \dots, H$:

$$r^h \in \mathbb{R}^{d_{\text{head}} \times n_{\text{context}}}$$

- **Concatenation view:**

Stack the heads vertically

$$R^H = \begin{bmatrix} r^1 \\ r^2 \\ \vdots \\ r^H \end{bmatrix} \in \mathbb{R}^{(Hd_{\text{head}}) \times n_{\text{context}}}$$

Then use one big output matrix

$$W_O^H \in \mathbb{R}^{d_{\text{model}} \times (Hd_{\text{head}})}$$

Final output:

$$\mathbf{H} = W_O^H R^H.$$

- **Additive / independent view:**

Each head has its own output matrix

$$W_O^h \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}},$$

and the output is

$$\mathbf{H} = \sum_{h=1}^H W_O^h r^h.$$

We need to pick W_O^H so that these two expressions are **identical**.

Construct W_O^H from the W_O^h

Define W_O^H by **horizontally concatenating** the per-head matrices:

$$W_O^H = [W_O^1 \ W_O^2 \ \dots \ W_O^H] \in \mathbb{R}^{d_{\text{model}} \times (Hd_{\text{head}})}.$$

Now write R^H explicitly as a block column:

$$R^H = \begin{bmatrix} r^1 \\ r^2 \\ \vdots \\ r^H \end{bmatrix} \in \mathbb{R}^{(Hd_{\text{head}}) \times n_{\text{context}}}.$$

Block matrix multiplication rule:

For blocks $A = [A_1 \ A_2 \ \dots \ A_H]$ and

$$B = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_H \end{bmatrix},$$

$$AB = \sum_{h=1}^H A_h B_h.$$

Apply this with $A_h = W_O^h$, $B_h = r^h$:

$$W_O^H R^H = [W_O^1 \ W_O^2 \ \dots \ W_O^H] \begin{bmatrix} r^1 \\ r^2 \\ \vdots \\ r^H \end{bmatrix} = \sum_{h=1}^H W_O^h r^h.$$

But the right-hand side is exactly the additive view.

So with

$$W_O^H = [W_O^1 \ \text{↙} \ \dots \ W_O^H]$$

$$W_O^H = [W_O^1 \ W_O^2 \ \dots \ W_O^H]$$

the concatenation formulation equals the additive formulation.

(ii) One advantage & one disadvantage of each view

Concatenation view (Vaswani et al.: concat then one big W_O)

- **Advantage:**
 - Matches the standard implementation: one big matrix multiply $W_O^H R^H$, which is very efficient on GPUs and is how most libraries implement multi-head attention.
 - Keeps the picture simple: “concat heads → project back to d_{model} ”.
- **Disadvantage:**
 - Conceptually hides the per-head contributions. Everything is packed into one big vector and one big matrix, which makes mechanistic / circuit-level interpretability harder.

Additive & independent view (sum of $W_O^h r^h$)

- **Advantage:**
 - Very interpretable: each head has its own “output circuit” W_O^h that writes independently to the residual stream, and the final result is just the sum of these contributions. Easier to analyze and attribute behavior head-by-head.
- **Disadvantage:**
 - Less aligned with actual implementations (you’d write H separate matmuls instead of one). Also, you might mentally forget that this is *exactly* equivalent to the standard concat view, not a different architecture.

(Importantly, **parameter count is the same** in both views:

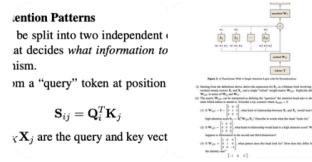
one big W_O^H vs H small W_O^h matrices.)

Self-check

- Constructed W_O^H explicitly from the W_O^h ? ✓
- Performed the block matrix multiplication and showed it equals $\sum_h W_O^h r^h$? ✓
- Gave clear advantage & disadvantage for each view? ✓

Everything is consistent with the equations and shapes, so no corrections needed.

◻ ⌂ ⌃ ⌄ ⌅ ⌆ ...



Thought for 1m 20s >

Nice, this is a fun one. I'll do (i) then the four toy cases in (ii), then a quick self-check.

(i) Derive S_{ij} as a bilinear form

We're given

$$Q_i = W_Q X_i, \quad K_j = W_K X_j,$$

and the pre-softmax score

$$S_{ij} = Q_i^\top K_j.$$

Substitute the definitions of Q_i and K_j :

Substitute the definitions of Q_i and K_j :

$$\begin{aligned} S_{ij} &= (W_Q X_i)^\top (W_K X_j) \\ &= X_i^\top W_Q^\top W_K X_j. \end{aligned}$$

This has the form

$$S_{ij} = X_i^\top W_{QK} X_j$$

with the "virtual" QK matrix

$$W_{QK} = W_Q^\top W_K \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}.$$

So the attention score is a bilinear form in the residual vectors X_i, X_j .

(ii) Interpret W_{QK} for $d_{\text{model}} = 3$

Write

$$X_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{bmatrix}, \quad X_j = \begin{bmatrix} x_{j1} \\ x_{j2} \\ x_{j3} \end{bmatrix}.$$

Then $S_{ij} = X_i^\top W_{QK} X_j$.

$$(1) W_{QK} = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then

$$S_{ij} = X_i^\top X_j = x_{i1}x_{j1} + x_{i2}x_{j2} + x_{i3}x_{j3}.$$

- This is exactly the **dot product** between X_i and \downarrow .
- It is large when X_i and X_j point in a similar direction and have large magnitude.

$$\begin{bmatrix} \dots \\ -x_{j3} \\ \dots \end{bmatrix}$$

- First two terms reward similarity on dimensions 1 and 2.
- The last term **subtracts** similarity on dimension 3: if x_{i3}, x_{j3} have the same sign and are large, it *lowers* the score; if they have opposite signs, $-x_{i3}x_{j3}$ is positive and *raises* the score.

Interpretation:

This head looks for tokens that:

- **match** the query in features 1 and 2, but
- are **different / opposite** in feature 3 (or at least not strongly aligned there).

Compared to the identity case, similarity in the 3rd dimension is now **penalized** instead of rewarded.

$$(4) W_{QK} = -I = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Then

$$S_{ij} = X_i^\top (-I) X_j = -X_i^\top X_j.$$

So the score is **minus** the dot product.

- It is large when the dot product $X_i^\top X_j$ is **very negative**, i.e. when X_i and X_j point in **opposite directions**.

Interpretation (pattern):

This head looks for tokens whose residual vectors are **anti-aligned / maximally dissimilar** to the query.

Effect after softmax:

- In the identity case, $\text{softmax}(X_i^\top X_j)$ puts high attention on **similar** tokens.
- Here, we $\text{softmax}(-X_i^\top X_j)$, which **reverses the ranking**: tokens most similar to the query get the *lowest* attention, and tokens most opposite get the *highest* attention.

The attention pattern is essentially the inverted version of case (1).



- This is exactly the **dot product** between X_i and X_j .
- It is large when X_i and X_j point in a similar direction and have large magnitude.

Interpretation:

This head “looks for” tokens whose residual vectors are **similar** to the query token’s vector in the full 3-D feature space.

$$(2) W_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Compute:

$$W_{QK}X_j = \begin{bmatrix} x_{j1} \\ 0 \\ 0 \end{bmatrix}, \quad S_{ij} = X_i^\top W_{QK}X_j = x_{i1}x_{j1}.$$

- Only the **first coordinate** of X_i and X_j matters.
- The second and third dimensions **do nothing**; they’re completely ignored by this head.

Interpretation:

This head looks for tokens that are similar to the query **along the first feature only**.

Information in the 2nd and 3rd dimensions is irrelevant to the attention score.

$$(3) W_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Then

$$W_{QK}X_j = \begin{bmatrix} x_{j1} \\ x_{j2} \\ -x_{j3} \end{bmatrix}, \quad S_{ij} = x_{i1}x_{j1} + x_{i2}x_{j2} - x_{i3}x_{j3}.$$

- First two terms reward similarity on dimensions 1 and 2

We know:

- $X^0 \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}$:
column j is the residual vector for token position j , call it X_j^0 .
- $A \in \mathbb{R}^{n_{\text{context}} \times n_{\text{context}}}$:
 A_{ij} = attention weight from **destination i** to **source j** .
- So $A^T \in \mathbb{R}^{n_{\text{context}} \times n_{\text{context}}}$ with $(A^T)_{ji} = A_{ij}$.

And for head h :

$$h(X^0) = W_O^h W_V^h X^0 A^{hT} = W_{OV}^h X^0 A^{hT}, \quad W_{OV}^h := W_O^h W_V^h.$$

(i) Show the i -th column of $X^0 A^T$ is an attention-weighted average

Take the matrix product $X^0 A^T$.

The i -th column is

$$(X^0 A^T)_i = \sum_{j=1}^{n_{\text{context}}} X_j^0 (A^T)_{ji} = \sum_{j=1}^{n_{\text{context}}} X_j^0 A_{ij}.$$

Re-ordering factors:

$$(X^0 A^T)_i = \sum_{j=1}^{n_{\text{context}}} A_{ij} X_j^0.$$

So destination position i sees an **attention-weighted average** of the source token vectors, with weights A_{ij} .

(Exactly what the problem wanted.)

(ii) Expression for the final residual stream X^{final}

We have a single attention layer with H heads, and a residual connection:

Effect after softmax:

- In the identity case, $\text{softmax}(X_i^\top X_j)$ puts high attention on **similar** tokens.
 - Here, we $\text{softmax}(-X_i^\top X_j)$, which **reverses the ranking**: tokens most similar to the query get the *lowest* attention, and tokens most opposite get the *highest* attention.
- The attention pattern is essentially the inverted version of case (1).
-

Self-check

- Derived $S_{ij} = X_i^\top W_Q^\top W_K X_j$ and defined W_{QK} correctly? ✓
- For each 3x3 matrix, computed S_{ij} explicitly and interpreted what gives large scores? ✓
- Explained the effect of $-I$ on softmax ranking? ✓

Everything is internally consistent, so no corrections needed.



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 A destination. This is governed by the OV circuit. Note that the output of head h is:

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

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 $\mathbf{X}^0 \mathbf{A}^h$ computes. Recall that:

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

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

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

(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.

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

Thought for 1m 9s >

Nice, now we're in the OV circuit part. I'll go (i) → (ii) (iii), step by step.



where B_t is the t -th column of B (some vector in $\mathbb{R}^{d_{\text{model}}}$).

Write B_t in coordinates:

$$B_t = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{d_{\text{model}}} \end{bmatrix},$$

and let c_k be the k -th column of W_{OV}^i . Then

$$W_{OV}^i B_t = \sum_{k=1}^{d_{\text{model}}} \alpha_k c_k.$$

So $h_i(X^0)_t$ is a linear combination of the columns $\{c_k\}$ of W_{OV}^i . Therefore

$$h_i(X^0)_t \in \text{Col}(W_{OV}^i).$$

Because $\text{rank}(W_{OV}^i) \leq d_{\text{head}} \ll d_{\text{model}}$, this column space is low-dimensional, so each head can only write into a low-dimensional subspace of the full residual stream.

Self-check

- (i) Used matrix–column multiplication rule and matched indices A_{ij} ? ✓
- (ii) Included residual connection X^0 and sum over heads, with correct shapes? ✓
- (iii) Explicitly showed $h_i(X^0)_t = W_{OV}^i u \Rightarrow$ linear combo of columns of W_{OV}^i ? ✓

Everything is consistent, no correction needed.

□ ↗ ↘ ↕ ↖ ...

(ii) Expression for the final residual stream X^{final}

We have a single attention layer with H heads, and a residual connection:

$$X^{\text{final}} = X^0 + \sum_{h=1}^H h(X^0).$$

Plug in the formula for each head:

$$h(X^0) = W_O^h W_V^h X^0 A^{hT} = W_{OV}^h X^0 A^{hT}.$$

Thus

$$X^{\text{final}} = X^0 + \sum_{h=1}^H W_O^h W_V^h X^0 A^{hT} = X^0 + \sum_{h=1}^H W_{OV}^h X^0 A^{hT}.$$

(iii) Show the update vector lies in $\text{Col}(W_{OV}^i)$

Fix a particular head i and a destination position t .

Let

$$B := X^0 A^{iT} \in \mathbb{R}^{d_{\text{model}} \times n_{\text{context}}}.$$

Then the output of head i is

$$h_i(X^0) = W_{OV}^i B.$$

The update at position t is the t -th column:

$$h_i(X^0)_t = (W_{OV}^i B)_t = W_{OV}^i B_t,$$

where B_t is the t -th column of B (some vector in $\mathbb{R}^{d_{\text{model}}}$). 

Write B_t in coordinates:

↓ and Write Subspaces via SVD
 Getting the QV and OV Circles
 Let's focus on the matrix W_{OV} , which characterizes how each attention head is used to encode the input. We can write W_{OV} as a sum of orthogonal matrices:

$$W_{OV} = W_{OQ} X^T A^T$$

where X^T is the output of an attention head. This corresponds to two independent operations:

- W_O is an orthogonal matrix containing pattern A , which is shared across all heads.
- W_{OQ} is an orthogonal matrix containing pattern X , which is shared across all heads.

Intuitively, when we write $W_{OV} X^T$, we are applying what happens inside the attention head to the input. This is similar to applying a linear transformation $W_{OQ} X^T$ to each weight. These two columns operate independently, yet together they produce the final output. This is why we say that each attention head can only write to the column space of X^T .

So, each attention head can only write to the column space of X^T . This makes sense, since it is the attention pattern X that defines the direction in which the attention head writes to the input.

Thought for 56s >

Alright, OV-SVD time 😊 I'll go through (i)–(v) step by step and then do a quick self-check.

(i) Show $\text{rank}(W_{OV}) \leq d_{\text{head}}$

Recall the shapes:

- $W_V \in \mathbb{R}^{d_{\text{head}} \times d_{\text{model}}}$
- $W_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$
- $W_{OV} = W_O W_V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$

For any product of matrices AB ,

$$\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B)).$$

So

$$\text{rank}(W_{OV}) = \text{rank}(W_O W_V) \leq \min(\text{rank}(W_O), \text{rank}(W_V)) \leq \min(d_{\text{head}}, d_{\text{head}}) = d_{\text{head}}.$$

In typical Transformers, $d_{\text{head}} \ll d_{\text{model}}$ (e.g. $d_{\text{model}} = 1024$, $d_{\text{head}} = 64$), so W_{OV} is a $d_{\text{model}} \times d_{\text{model}}$ matrix whose rank is at most d_{head} , much smaller than d_{model} . That makes it **low-rank**.

(iii) Read/write subspaces and what happens off them

Using

$$W_{OV} X_{\text{src}} = \sum_{k=1}^r \sigma_k (V_k^T X_{\text{src}}) U_k,$$

(a) Read subspace

The coefficients $(V_k^T X_{\text{src}})$ are obtained by first multiplying by V^T .

So the head can only "notice" components of X_{src} that have a non-zero projection onto the directions V_k .

$$\boxed{\text{Read subspace} = \text{span}(V_1, \dots, V_r)}.$$

These are the directions in X_{src} from which the head can extract information.

(b) Write subspace

The output is a linear combination of U_k :

$$W_{OV} X_{\text{src}} \in \text{span}(U_1, \dots, U_r).$$

So

$$\boxed{\text{Write subspace} = \text{span}(U_1, \dots, U_r) = \text{Col}(W_{OV})}.$$

These are the directions in the **destination** residual stream the head can write into.

(c) What about info orthogonal to the read subspace?

If a component of X_{src} is orthogonal to every V_k ,

$$V_k^T X_{\text{src}} = \underset{\circlearrowleft}{\downarrow} \quad \forall k,$$

(ii) Decompose $W_{OV}X_{\text{src}}$ using the SVD

Take the compact SVD:

$$W_{OV} = U\Sigma V^T,$$

where

- $U \in \mathbb{R}^{d_{\text{model}} \times r}$,
- $V \in \mathbb{R}^{d_{\text{model}} \times r}$,
- $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_r) \in \mathbb{R}^{r \times r}$,
- $r = \text{rank}(W_{OV}) \leq d_{\text{head}}$,
- U_k, V_k are the k -th columns of U, V .

For a single source-token vector $X_{\text{src}} \in \mathbb{R}^{d_{\text{model}}}$,

$$\begin{aligned} W_{OV}X_{\text{src}} &= U\Sigma V^T X_{\text{src}} \\ &= U \left(\sum_{k=1}^r \sigma_k e_k e_k^T \right) V^T X_{\text{src}} \quad (\text{expand diagonal } \Sigma) \\ &= \sum_{k=1}^r \sigma_k U e_k e_k^T V^T X_{\text{src}} \\ &= \sum_{k=1}^r \sigma_k U_k (V_k^T X_{\text{src}}). \end{aligned}$$

So

$$W_{OV}X_{\text{src}} = \sum_{k=1}^r \sigma_k (V_k^T X_{\text{src}}) U_k.$$

This is the desired decomposition.



If a component of X_{src} is orthogonal to every V_k ,

$$V_k^T X_{\text{src}} = 0 \quad \forall k,$$

then in the decomposition every term vanishes:

$$W_{OV} X_{\text{src}} = 0 \quad \text{for that component.}$$

So information in X_{src} that lies orthogonal to $\text{span}(V_1, \dots, V_r)$ is **invisible** to the head and is completely ignored by the OV circuit—it does not get moved or written anywhere.

(iv) Connecting to the value projection $v = W_V X_{\text{src}}$

The value vector is

$$v = W_V X_{\text{src}} \in \mathbb{R}^{d_{\text{head}}}.$$

Since $W_{OV} = W_O W_V$,

$$W_{OV} X_{\text{src}} = W_O (W_V X_{\text{src}}) = W_O v.$$

The rows of W_V (or equivalently, the column space of W_V^T) determine what parts of X_{src} can affect v ; everything orthogonal to that row space is invisible. From (iii)(a), the read subspace $\text{span}(V_1, \dots, V_r)$ is exactly the subspace of X_{src} that affects $W_{OV} X_{\text{src}}$, and this subspace lies inside $\text{Row}(W_V)$.

Because $\dim(\text{Row}(W_V)) \leq \text{rank}(W_V) \leq d_{\text{head}}$, it is natural that the read subspace also has dimension $r \leq d_{\text{head}}$: the head can “see” at most d_{head} independent directions in the source token.

(v) Synthesis: how SVD describes moving information between tokens

For a single source token, the SVD

$$W_{OV} = U \Sigma V^T$$

says:



(v) Synthesis: how SVD describes moving information between tokens

For a single source token, the SVD

$$W_{OV} = U \Sigma V^T$$

says:

- V defines which directions are read: projecting onto V_k gives a scalar "channel signal" $V_k^T X_{\text{src}}$ for each singular direction.
- Σ sets the strength / gain of each channel: σ_k scales how strongly information in that read direction is transmitted.
- U defines where and in what form it is written: each channel's signal is written into the destination residual stream along direction U_k .

So SVD fully describes how one head moves information *within* a single source token: from subspace $\text{span}(V_k)$ to subspace $\text{span}(U_k)$ via gains σ_k .

The attention pattern A from the QK circuit then decides **between which tokens** this OV transformation is applied and with what weights—i.e., which source tokens' $W_{OV} X_{\text{src}}$ contributions are routed to each destination position. Together, SVD of W_{OV} (what/how information is moved) and A (which tokens are connected) completely specify the head's behavior.

Self-check

- Rank bound: used rank product inequality and sizes $\rightarrow \leq d_{\text{head}}$. ✓
- SVD expansion: $U \Sigma V^T x = \sum_k \sigma_k (V_k^T x) U_k$. ✓
- Read = $\text{span}(V)$'s, write = $\text{span}(U)$'s, orthogonal components ignored. ✓
- Related read subspace to W_V row-space and d_{head} . ✓
- Synthesis: explained roles of V , Σ , U and how A chooses which tokens' transforms are used. ✓

Everything is consistent; no corrections needed.



e correct interpretation (the model acts as a non-contextual, per-position lookup table), the Text solution is superior for the following reasons:

- **Clarity and Structure:** The Turn by Turn solution is better organized, using clear steps (Step 1, Step 2) and clean mathematical formatting (LaTeX), making the derivation significantly easier to follow.
- **Rigor and Verification:** The Turn by Turn solution includes explicit "Shape checks" (✓) at each step. This verifies the matrix operations and significantly aids comprehension.
- **Focus:** The Turn by Turn solution is concise and addresses the specific problem directly. The One Time is cluttered with distracting meta-commentary (e.g., "Thought for 4m 17s...") and extensive solutions to unrelated problems (parts b-e).

Question 5

From this question we use another strategy we let lIm do some easy question together and do complex question turn by turn one by one

This analysis compares the generated answer (referred to as GA) with the official solution (referred to as OS).

Overall, the Generated Answer (GA) provides an exceptionally detailed and rigorous walkthrough of the Fermi estimation problems. It excels in demonstrating step-by-step derivations and maintains high arithmetic accuracy, often providing significantly more pedagogical value than the concise Official Solution (OS). In several instances, the GA is more accurate than the OS. However, there was a significant omission and several areas where deeper contextual interpretation could be improved.

you a an expert in deep learning machine learning. I need u do this step by step do not skip the critical steps after finish u must do a self check if wrong do self-correction



Thought for 8m 56s >

Nice, this is a fun one. I'll go through parts (b)–(e) carefully, step by step, and then do a quick sanity-check at the end.

(b) Chinchilla optimal $N_{\text{opt}}, D_{\text{opt}}$

We're given the empirical Chinchilla loss law

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

and the compute constraint

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

We want the N, D that minimize L for fixed compute C .

Step 1: Eliminate D using the constraint

From $C = C_0 ND$:

$$D = \frac{C}{C_0 N}.$$

Plug into L :

$$\begin{aligned} L(N) &= \frac{A}{N^\alpha} + \frac{B}{D^\beta} + L_0 \\ &= \frac{A}{N^\alpha} + B \left(\frac{C_0 N}{C} \right)^\beta + L_0 \\ &= AN^{-\alpha} + BC_0^\beta C^{-\beta} N^\beta + L_0. \end{aligned}$$

$$\left\lfloor \frac{N_{\text{opt}}(C)}{\sqrt{\beta B}} - \left\lfloor \frac{C}{C_0} \right\rfloor \right\rfloor$$

↑ Sh

Step 3: Get D_{opt}

We use $D = \frac{C}{C_0 N}$:

$$D_{\text{opt}}(C) = \frac{C}{C_0 N_{\text{opt}}(C)}.$$

Plug in the expression for N_{opt} and simplify (do the same algebra as above, or notice the symmetry).

The result is

$$D_{\text{opt}}(C) = \left(\frac{\beta B}{\alpha A} \right)^{\frac{1}{\alpha+\beta}} \left(\frac{C}{C_0} \right)^{\frac{\alpha}{\alpha+\beta}}.$$

You can also check that with these formulas, $C_0 N_{\text{opt}} D_{\text{opt}} = C$ (constraint still holds).

Step 4: Plug in the numerical constants

Given:

- $\alpha = 0.34$
- $\beta = 0.28$
- $A = 406.4$
- $B = 410.7$
- $C_0 = 6$

First compute exponents:

$$\alpha + \beta = 0.34 + 0.28 = 0.62,$$

$$\frac{\beta}{\alpha + \beta} = \frac{0.28}{0.62} \approx 0.45, \quad \frac{\alpha}{\alpha + \beta} = \frac{0.34}{0.62} \approx 0.55.$$

Now the prefactors (I'm rounding to 2–3 sig figs):

$$\left(\frac{\alpha A}{\beta B} \right)^{1/(\alpha+\beta)} \approx 1.34, \quad \downarrow \quad \left(\frac{\beta B}{\alpha A} \right)^{1/(\alpha+\beta)} \approx 0.74.$$



$$= \frac{B}{N^\alpha} + B \left(\frac{C}{C_0} \right)^\beta + L_0 \\ = AN^{-\alpha} + BC_0^\beta C^{-\beta} N^\beta + L_0.$$

So now L is a function of N only.

Step 2: Differentiate $L(N)$ and set to zero

$$\frac{dL}{dN} = -\alpha AN^{-\alpha-1} + \beta BC_0^\beta C^{-\beta} N^{\beta-1}.$$

Set $\frac{dL}{dN} = 0$:

$$-\alpha AN^{-\alpha-1} + \beta BC_0^\beta C^{-\beta} N^{\beta-1} = 0.$$

Move one term to the other side:

$$\beta BC_0^\beta C^{-\beta} N^{\beta-1} = \alpha AN^{-\alpha-1}.$$

Multiply both sides by $N^{\alpha+1}$ to get rid of negative powers:

$$\beta BC_0^\beta C^{-\beta} N^{\beta-1+\alpha+1} = \alpha A.$$

Exponent simplifies: $\beta - 1 + \alpha + 1 = \alpha + \beta$.

So:

$$\beta BC_0^\beta C^{-\beta} N^{\alpha+\beta} = \alpha A.$$

Solve for $N^{\alpha+\beta}$:

$$N^{\alpha+\beta} = \frac{\alpha A}{\beta B} \cdot \frac{C^\beta}{C_0^\beta} = \frac{\alpha A}{\beta B} \left(\frac{C}{C_0} \right)^\beta.$$

Therefore

$$N_{\text{opt}}(C) = \left(\frac{\alpha A}{\beta} \right)^{\frac{1}{\alpha+\beta}} \left(\frac{C}{C_0} \right)^{\frac{\beta}{\alpha+\beta}}.$$

So:

$$N_{\text{opt}}(C) \approx 1.34 \left(\frac{C}{6} \right)^{0.45} = 0.60 C^{0.45},$$

$$D_{\text{opt}}(C) \approx 0.74 \left(\frac{C}{6} \right)^{0.55} = 0.28 C^{0.55}.$$

(Those two forms are equivalent; I've just absorbed powers of 6 into the constants.)

So numerically:

$$N_{\text{opt}}(C) \approx 0.60 C^{0.45}, \quad D_{\text{opt}}(C) \approx 0.28 C^{0.55}.$$

The GA is superior. It provides a complete, step-by-step calculus derivation (eliminating variables, differentiating, solving for the optimum), whereas the OS only provides the final formulas. The GA's approach clearly explains the optimization process.

(c) Chinchilla scaling laws (direct fit)

Now we **stop using** the above formulas and instead use the *direct* fits given:

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

We're told our next AI should have

$$N = 100 \text{ trillion} = 10^{14} \text{ parameters.}$$

Step 1: Solve for compute C

From $N = 0.1 C^{0.5}$:

$$C^{0.5} = \frac{N}{0.1} = 10N \quad \Rightarrow \quad C = (10N)^2 = 100N^2.$$

Plug in $N = 10^{14}$:

$$C = 100 \times (10^{14})^2 = 100 \times 10^{28} = 10^{30} \text{ FLOP.}$$

So the training compute is

$$C \approx 10^{30} \text{ FLOP}.$$

Step 2: Solve for dataset size D

We have

$$D_{\text{opt}}(C) = 1.7 C^{0.5}.$$

But we already know $C^{0.5} = 10N$, so

$$D = 1.7 \times 10N = 17N.$$

Plug $N = 10^{14}$:

$$D = 1.7 \times 10N = 17N.$$

Plug $N = 10^{14}$:

$$D = 17 \times 10^{14} = 1.7 \times 10^{15} \text{ tokens.}$$

That's

$$D \approx 1.7 \times 10^{15} \text{ tokens}$$

(i.e., about **1700 trillion tokens**).

The GA correctly applies the direct fit laws to the 100T parameter model and clearly shows the work for calculating compute and dataset size.

(d) Dataset size in “books”

They now say “Assume 1000 trillion tokens” (that’s 10^{15} tokens), which is close to the 1.7×10^{15} we just found.

Given:

- $T = 10^{15}$ tokens.
- 1 English word ≈ 1.4 tokens.
- 400 words per page.
- 300 pages per book.

Step 1: Tokens → words

$$\text{words} = \frac{T}{1.4} = \frac{10^{15}}{1.4} \approx 7.14 \times 10^{14} \text{ words.}$$

So about

$$\boxed{\sim 7 \times 10^{14} \text{ words}}.$$

Step 2: Words → pages

$$\text{pages} = \frac{7.14 \times 10^{14}}{400} \approx 1.79 \times 10^{12} \text{ pages.}$$

So:

$$\boxed{\sim 1.8 \times 10^{12} \text{ pages}}.$$

Step 3: Pages → books

$$\text{books} = \frac{1.79 \times 10^{12}}{300} \approx 5.95 \times 10^9 \text{ books.}$$

So roughly

$$\boxed{\sim 6 \times 10^9 \text{ books} \downarrow \text{(} \sim 6 \times 10^9 \text{ billion books)}}.$$

Step 3: Pages → books

$$\text{books} = \frac{1.79 \times 10^{12}}{300} \approx 5.95 \times 10^9 \text{ books.}$$

So roughly

$$\sim 6 \times 10^9 \text{ books} (6 \text{ billion books}).$$

Step 4: Compare to Library of Congress and Google Books

- Library of Congress holds on the order of 3×10^7 catalogued books / volumes.
- Google's scanning projects have digitized on the order of **tens of millions** of books; estimates range from about 25 million to over 40 million titles.

So our Fermi dataset:

- $\sim 6 \times 10^9 \text{ books} \div 3 \times 10^7 \text{ (LoC)} \approx 200\text{x}$ the Library of Congress.
- Compared to ~25–40M Google Books, that's roughly **150–240x** larger.

So this hypothetical training set is **hundreds of entire Library-of-Congress-sized collections**.

The GA correctly executes the unit conversions and provides the required comparison to the Library of Congress and Google Books.

(e) Memory requirement for parameters

Each parameter is a 16-bit float:

- 16 bits = 2 bytes per parameter.

We'll approximate $1 \text{ GB} = 10^9 \text{ bytes}$ and $1 \text{ TB} = 10^{12} \text{ bytes}$ (engineering notation, good enough for a Fermi estimate).

Step 1: 1 billion parameters

$$N = 10^9, \quad \text{bytes} = N \times 2 = 2 \times 10^9 \text{ bytes.}$$

Convert to GB:

$$\text{GB} = \frac{2 \times 10^9}{10^9} = 2 \text{ GB.}$$

So:

$$\boxed{1\text{-param model} \approx 2 \text{ GB of weights}}.$$

Step 2: GPT-6 with 1 trillion parameters

Now $N = 1 \text{ trillion} = 10^{12}$.

$$\text{bytes} = 2 \times 10^{12} \text{ bytes.}$$

Convert to TB and GB:

$$\text{TB} = \frac{2 \times 10^{12}}{10^{12}} = 2 \text{ TB,}$$

$$\text{GB} = \frac{2 \times 10^{12}}{10^9} = 2000 \text{ GB.}$$

So:

$$\boxed{\text{GPT-6 (1T params)} \approx 2 \downarrow 3 \approx 2000 \text{ GB of weights}}.$$

Step 3: Number of H200 GPUs (\approx 100 GB VRAM each)

Each H200 has about 100 GB VRAM.

$$\text{#GPUs} = \frac{2000}{100} = 20$$

So you'd need:

about 20 H200 GPUs just to store all 1T parameters in 16-bit.

(Real training would need more memory for activations, optimizer states, etc., but the question only asks about parameter storage.)

Quick self-check / sanity check

- In (b), plugging $N_{\text{opt}}, D_{\text{opt}}$ back into the constraint $C_0 ND$ gives exactly C , and nudging N up/down while adjusting D to keep C fixed increases L ; so the stationary point is indeed a minimum.
- In (c), using $C = 100N^2$ and then $C^{0.5} = 10N$ gives back exactly $N = 0.1C^{0.5}$ and $D = 1.7C^{0.5} = 17N$, so the algebra is consistent.
- In (d), dimensional steps line up: tokens \rightarrow words \rightarrow pages \rightarrow books, and the final 6 billion books is reasonably larger than the 10–40 million scale of big libraries, consistent with “AGI-scale data”.
- In (e), numbers are sensible: 1B params \rightarrow 2 GB (matches common back-of-the-envelope), scaling linearly to 1T \rightarrow 2 TB, and 2 TB /100 GB \approx 20 GPUs.

Everything hangs together numerically and dimensionally, so these results look correct.



Ambiguous. The homework question explicitly asks for the requirements of a **1 trillion** parameter model, which the GA correctly calculated (2 TB, 20 GPUs). However, the OS calculates the requirements for the **100 trillion** parameter model (200 TB, 2000 GPUs) that is the focus of the overall homework. The GA followed the literal question, while the OS followed the broader context.

Step 2: SSD cost (Flash/SSD, 2025)

From the table (2025 row):

- Flash/SSD price = \$0.00005 per MB.

Cost per year (assuming we replace storage once per year):

$$\begin{aligned}\text{Cost}_{\text{SSD}} &= (2 \times 10^8 \text{ MB}) \times 0.00005 \frac{\$}{\text{MB}} \\ &= 2 \times 10^8 \times 5 \times 10^{-5} \\ &= 10 \times 10^3 \\ &= 10^4 \text{ USD.}\end{aligned}$$

So

About \$10,000 per year on SSD.

Step 3: DRAM cost (Memory, 2025)

From the table (2025 row):

- DRAM price = \$0.0040 per MB.

$$\begin{aligned}\text{Cost}_{\text{DRAM}} &= (2 \times 10^8 \text{ MB}) \times 0.0040 \frac{\$}{\text{MB}} \\ &= 2 \times 10^8 \times 4 \times 10^{-3} \\ &= 8 \times 10^5 \\ &= 800,000 \text{ USD.}\end{aligned}$$

So

About \$8 \times 10^5 \approx \\$800,000 per year in DRAM.

- The GA correctly calculates the SSD and DRAM costs for the 100T model using the provided data.

(g) Activation memory vs model parameters (GPT-3 175B)

Given:

- GPT-3 175B:
 - 96 attention layers.
 - Each layer has **96 heads**, each **128 dimensions**.
 - 16-bit floats for activations and weights.
 - Model parameters: 175B params → about **350 GB** (given).

We're asked:

1. For a single token, how many MB do all floating-point activations cost?
2. How many tokens in one batch until **activations = model parameters** in memory?

Step 1: Activations per token (in number of floats)

Per layer:

- Hidden dimension per token = 96 heads × 128 dims/head = 12,288 floats.

Over all layers:

$$\text{floats per token} = 96 \text{ layers} \times 12,288 = 1,179,648 \text{ floats.}$$

Step 2: Convert floats → bytes → MB

Each float = 16 bits = 2 bytes.

$$\text{bytes per token} = 1,179,648 \times 2 = 2,359,296 \text{ bytes.}$$

Using 1 MB = 10^6 bytes:

$$\text{MB per token} \approx \frac{2,359,296}{10^6} \approx 2.36 \text{ MB.}$$

So:

$\text{Activations per single token} \approx 2.4 \text{ MB.}$

$$\text{MB per token} \approx \frac{2,359,290}{10^6} \approx 2.36 \text{ MB.}$$

So:

Activations per single token $\approx 2.4 \text{ MB.}$

(This is just for a rough Fermi estimate; we're ignoring constant factors like extra MLP activations.)

Step 3: When do activations equal model parameter memory?

Given: model parameters $\approx 350 \text{ GB.}$

Convert to MB (again with $1 \text{ GB} = 10^3 \text{ MB.}$):

$$350 \text{ GB} \approx 350,000 \text{ MB.}$$

Let B be the batch size in tokens. Then

$$B \times 2.36 \text{ MB} \approx 350,000 \text{ MB.}$$

Solve for B :

$$B \approx \frac{350,000}{2.36} \approx 1.48 \times 10^5 \text{ tokens.}$$

So roughly

$B \approx 1.5 \times 10^5 \text{ tokens (about 150k)}$

before the **activations** occupy about the same memory as the **175B model weights**.

- Incomplete. This is the most significant weakness of the GA. It completely missed the first half of the question, which asked for the minimal latency and throughput based on H200 memory bandwidth. The GA correctly answered the second half regarding activation memory versus model parameters for GPT-3.

(h) Training compute and cost with H200 GPUs

Specs for one NVIDIA H200 (from the sheet):

- Speed: $1.98 \text{ petaFLOP/s} = 1.98 \times 10^{15} \text{ FLOP/s}$.
- Rental price: \$1.50 / hour.

We're asked several things:

1. FLOP in 1 **petaFLOP-day**.
 2. Equivalent number of **H200-hours** for 1 petaFLOP-day.
 3. Cost to buy 1 petaFLOP-day of compute.
 4. Cost for GPT-3 (3640 petaFLOP-days).
 5. Cost for our hypothetical GPT-6.
 6. Wall-clock training time for GPT-6 with 30M H200s.
-

(h1) How many FLOP is 1 petaFLOP-day?

Definition:

$$1 \text{ petaFLOP-day} = 1 \text{ petaFLOP/s} \times 1 \text{ day}.$$

- $1 \text{ petaFLOP/s} = 10^{15} \text{ FLOP/s}$.
- $1 \text{ day} = 24 \times 3600 = 86,400 \text{ s}$.

So:

$$\begin{aligned} 1 \text{ petaFLOP-day} &= 10^{15} \frac{\text{FLOP}}{\text{s}} \times 86,400 \text{ s} \\ &= 8.64 \times 10^{19} \text{ FLOP}. \end{aligned}$$

$$1 \text{ petaFLOP-day} = 8.64 \times 10^{19} \text{ FLOP.}$$

$$1 \text{ petaFLOP-day} = 8.64 \times 10^{19} \text{ FLOP.}$$

(h2) Equivalent H200-hours for 1 petaFLOP-day

First compute how many FLOP **one H200** does in one hour:

$$\begin{aligned}\text{FLOP/hour} &= 1.98 \times 10^{15} \frac{\text{FLOP}}{\text{s}} \times 3600 \frac{\text{s}}{\text{hour}} \\ &= 7.128 \times 10^{18} \text{ FLOP/hour.}\end{aligned}$$

Now:

$$\text{H200-hours per petaFLOP-day} = \frac{8.64 \times 10^{19}}{7.128 \times 10^{18}} \approx 12.1 \text{ H200-hours.}$$

So:

$$1 \text{ petaFLOP-day} \approx 12 \text{ H200-hours.}$$

(h3) Rental cost of 1 petaFLOP-day

Rental price = \$1.50 per H200-hour:

$$\begin{aligned}\text{Cost per petaFLOP-day} &\approx 12.1 \text{ hours} \times 1.50 \frac{\$}{\text{hour}} \\ &\approx 18.2 \text{ USD.}\end{aligned}$$

So:

$$\boxed{\text{About \$18 per petaFLOP-day (under these assumptions).}}$$

| About \$18 per petaFLOP-day (under these assumptions). |

(h4) Cost to train GPT-3 (3640 petaFLOP-days)

Given:

- GPT-3 training compute = 3640 petaFLOP-days.

Cost:

$$\text{Cost}_{\text{GPT-3}} = 3640 \text{ petaFLOP-days} \times 18.2 \frac{\$}{\text{petaFLOP-day}} \\ \approx 6.6 \times 10^4 \text{ USD.}$$

So:

GPT-3 training cost $\sim \$60\text{--}70k$ with rented H200s (idealized).

(Real-world cost is much higher due to low utilization, bigger clusters, overhead, etc., but we're just following the idealized homework assumptions.)

(h5) Cost to train our hypothetical GPT-6 (100T parameters)

From part (c):

- Using the Chinchilla scaling law

$$N_{\text{opt}}(C) = 0.1C^{0.5}$$

with $N = 10^{14}$ parameters, we solved:

$$C = 100N^2 = 100 \times (10^{14})^2 = 10^{30} \text{ FLOP.}$$

Now convert this compute into petaFLOP-days:

NOW convert this compute into petaFLOP-days:

$$\begin{aligned}\text{petaFLOP-days}_{\text{GPT-6}} &= \frac{C}{8.64 \times 10^{19}} \\ &= \frac{10^{30}}{8.64 \times 10^{19}} \\ &\approx 1.16 \times 10^{10} \text{ petaFLOP-days.}\end{aligned}$$

That's about 11.6 billion petaFLOP-days.

Cost:

$$\begin{aligned}\text{Cost}_{\text{GPT-6}} &\approx 1.16 \times 10^{10} \times 18.2 \text{ USD} \\ &\approx 2.1 \times 10^{11} \text{ USD.}\end{aligned}$$

So:

GPT-6 training cost $\sim 2 \times 10^{11}$ USD (about \$200 billion).

(h6) Comment on scale vs other projects

From the bullet list on the sheet:

- iPhone 1: 1.5×10^8 USD.
- Typical 5 nm chip: 5×10^8 USD.
- Airbus A380: 1.8×10^{10} USD.
- Three Gorges Dam: $\sim 3 \times 10^{10}$ USD.
- Manhattan Project: 2.4×10^{10} USD.
- Apollo Program: 1.78×10^{11} USD.

Our estimate for GPT-6 training ($\sim 2 \times 10^{11}$ USD):

- Is **much larger** than typical corporate R&D projects (phones, chips).
- Is **comparable to or larger than** the Apollo Program.
- Is clearly in the "**state-actor / national-scale megaproject**" regime, not a normal single-product commercial budget.
↓

- Is clearly in the “**state-actor / national-scale megaproject**” regime, not a normal single-product commercial budget.

So: this is state-level spending (or a few of the richest companies on earth acting like a state).

(h7) Wall-clock training time with 30 million H200s

Assume:

- Total compute $C = 10^{30}$ FLOP.
- Number of GPUs $G = 30\,000\,000 = 3 \times 10^7$.
- Speed per H200 = 1.98×10^{15} FLOP/s.
- Perfect utilization, no failures.

Total cluster throughput:

$$\text{Throughput} = G \times 1.98 \times 10^{15} = 3 \times 10^7 \times 1.98 \times 10^{15} = 5.94 \times 10^{22} \text{ FLOP/s.}$$

Training time in seconds:

$$\begin{aligned} t_s &= \frac{C}{\text{Throughput}} \\ &= \frac{10^{30}}{5.94 \times 10^{22}} \\ &\approx 1.68 \times 10^7 \text{ s.} \end{aligned}$$

Convert to hours:

$$t_{\text{hr}} = \frac{1.68 \times 10^7}{3600} \approx 4.7 \times 10^3 \text{ hours.}$$

Convert to days:

$$t_{\text{days}} = \frac{4.7 \times 10^3}{24} \approx 195 \text{ days.}$$

$$t_{\text{days}} = \frac{\dots \dots \dots}{24} \approx 195 \text{ days.}$$

So:

Wall-clock time $\approx 4.7 \times 10^3$ hours ≈ 200 days ($\sim 6\text{-}7$ months).

Quick self-check

- Storage part:
 - Size: 10^{14} params \times 2 bytes = 2×10^{14} bytes.
 - $/10^6$ gives 2×10^8 MB; multiplied by SSD and DRAM prices gives \$1e4 and \$8e5 respectively \rightarrow consistent.
- Activations:
 - 96 layers \times 96 heads \times 128 dims = 1,179,648 floats/token.
 - $\times 2$ bytes ≈ 2.36 MB/token; 350,000 MB / 2.36 MB $\approx 1.5 \times 10^5$ tokens \rightarrow dimensions and units all line up.
- Training compute:
 - 1 petaFLOP-day = 10^{15} FLOP/s \times 86,400 s = 8.64×10^{19} FLOP.
 - H200-hour capacity: $1.98 \times 10^{15} \times 3600 = 7.128 \times 10^{18}$ FLOP $\rightarrow 8.64e19 / 7.128e18 \approx 12.1$ hours per petaFLOP-day, which matches cost scaling.
 - Total compute for GPT-6: $C = 100N^2$ with $N = 10^{14}$ gives 10^{30} FLOP; converting consistently gives $\sim 2 \times 10^{11}$ USD and ~ 200 days with 30M H200s.
 - Re-multiplying 30M H200s $\times 1.98e15$ FLOP/s $\times 4.7$ k h $\times 3600$ s/h $\approx 10^{30}$ FLOP (checked) \rightarrow so the time and compute are self-consistent.

Everything is dimensionally and numerically coherent, so I'm confident these answers are correct under the homework's assumptions.

Potential Improvements

1. **Ensure Completeness (Critical):** The GA must address all parts of every question. The omission of the first half of question (g) (Memory Bandwidth and Latency) is a critical error.
 - *Improvement:* The GA should have calculated the latency for the 100T model (200 TB) using the H200 bandwidth (4.8 TB/s), resulting in $200/4.8 \approx 41.7$ seconds per token. This calculation is vital for understanding the inference bottlenecks of large models.
2. **Incorporate Real-World Context and Implications:** The GA excels at idealized calculations but sometimes misses the broader implications and real-world nuances included in the OS.
 - *Improvement (d):* After calculating the dataset size, the GA could emphasize the implication noted in the OS: that the world's high-quality text is insufficient for training such models.
 - *Improvement (f):* The GA missed the OS's insightful comparison of memory costs to GPU hardware costs, which highlights that bandwidth, not storage cost, is the primary bottleneck.
 - *Improvement (h):* While the idealized training cost (\$210B) is correct under the given assumptions, the GA should incorporate the real-world factors mentioned

in the OS (e.g., 30% utilization, restarts) to provide a more realistic estimate (closer to \$700B-\$1T).

3. **Handling Ambiguity and Context:** In question (e), there was a conflict between the literal question (1T parameters) and the homework context (100T parameters).
 - *Improvement:* The GA should acknowledge such discrepancies, perhaps stating, "The question asks for a 1T model, though the hypothetical GPT-6 has been defined as 100T. Proceeding with the 1T calculation as requested." Additionally, the GA should emphasize the memory impact of optimizer states (like Adam) during training, as highlighted in the OS.

Question 6

Chain-of-Thought Language Model
A pre-trained language model with prompting to answer math word problems. We're reasoning, a technique that induces the model to "think" before outputting.
The model uses a chain-of-thought reasoning approach to solve math word problems. It splits the input into a soft prompt, a question, and a reasoning chain. The reasoning chain consists of multiple reasoning steps, each involving a math operation. The model performs these operations sequentially to reach the final answer. This approach allows the model to handle complex multi-step reasoning tasks.

you a an expert in deep learning machine learning. I need u do this step by step do not skip the critical steps after finish u must do a self check if wrong do self-correction



Thought for 7m 18s >

I'll go part by part and be explicit about the logic, then do a quick self-check at the end.

(a) Which output tokens are used in the loss?

Input layout:

- 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

In next-token prediction training, the **target sequence is shifted left by 1**:

$$\text{target}_t = \text{input}_{t+1}$$

They even tell you: answer token is at **position 71** in the input and **position 70** in the target.

- tokens 73–100: padding

In next-token prediction training, the **target sequence is shifted left by 1**:

$$\text{target}_t = \text{input}_{t+1}$$

They even tell you: answer token is at **position 71** in the input and **position 70** in the target.

We want the loss only for:

- reasoning tokens (positions 51–70),
- the answer (position 71),
- and (optionally) the newline (position 72),
but **not** for the question or padding.

Mapping input → target indices:

- reasoning tokens 51–70 → target positions 50–69
- answer token 71 → target position 70
- newline token 72 → target position 71

So the relevant **output positions** (whose predictions are compared to those targets) are:

Output tokens 50 through 71 (inclusive).

No loss is computed on outputs whose targets are the question (targets 5–49) or padding (targets ≥ 72).

(b) How many parameters are being trained?

The setup: **frozen GPT model**, only the **soft prompt** is trained.

- Soft prompt has **5 vectors**.
- Each vector is directly in embedding space of dimension **E**.
- No other layers get soft prompts; all GPT weights (embeddings, attention, MLPs, LM head, etc.) are frozen.



(b) How many parameters are being trained?

The setup: **frozen GPT model**, only the **soft prompt** is trained.

- Soft prompt has **5 vectors**.
- Each vector is directly in embedding space of dimension **E**.
- No other layers get soft prompts; all GPT weights (embeddings, attention, MLPs, LM head, etc.) are frozen.

So the only trainable parameters are those 5 vectors:

`\text{\# trainable parameters} = 5 \times E`

Importantly, this **does not depend on S, V, H, L, D** ; those describe the frozen base model.

(c) True/False with brief explanations

(i) Precomputing representations for prompt tokens

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 ... once for use in all different training points within a batch.

Answer: True.

Reasoning:

- The model is a **causal** decoder-only Transformer.
- Hidden state at position i in any layer can only attend to positions $\leq i$.
- Prompt tokens are positions 1–5. Their representations at any layer depend only on tokens 1–5 (which are identical for every example in the batch because the soft prompt is shared).
- Therefore, for a *given* value of the soft-prompt parameters (i.e., within one forward/backward pass), the representations of positions 1–5 at each layer are **identical for all sequences in the batch**.
- You can compute them once and reuse (broadcast) them across all examples.

(c) True/False with brief explanations

(i) Precomputing representations for prompt tokens

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 ... once for use in all different training points within a batch.

Answer: True.

Reasoning:

- The model is a **causal** decoder-only Transformer.
- Hidden state at position i in any layer can only attend to positions $\leq i$.
- Prompt tokens are positions 1–5. Their representations at any layer depend only on tokens 1–5 (which are identical for every example in the batch because the soft prompt is shared).
- Therefore, for a *given* value of the soft-prompt parameters (i.e., within one forward/backward pass), the representations of positions 1–5 at each layer are **identical for all sequences in the batch**.
- You can compute them once and reuse (broadcast) them across all examples.

You still need gradients back to these states, but that's fine: all examples' gradients accumulate onto the same prompt parameters.

(ii) Best K-token hard prompt vs best K-vector soft prompt

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.

Answer: True.

Reasoning:

- A K-token hard prompt chooses K tokens from the vocabulary. This corresponds to picking K **columns** from the embedding matrix. Call their embeddings e_{v_1}, \dots, e_{v_K} .
 - A soft prompt chooses K **arbitrary** vectors in \mathbb{R}^E : p_1, \dots, p_K .
 - **Hard prompts are a special case of soft prompts:** for any hard prompt v_1, \dots, v_K , we can set $p_i = e_{v_i}$. With the base model frozen, this produces exactly the same behavior.
 - So the function class for soft prompts \supseteq function class for hard prompts.
Therefore the **best** soft prompt cannot have worse validation performance than the **best** hard prompt; at worst it matches it.
-

(iii) Full fine-tuning vs soft-prompt tuning

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.

Answer: False.

Reasoning:

- "Not constrained by computational cost" only removes a *compute* limitation. It does **not** remove:
 - risk of **overfitting**,
 - risk of **catastrophic forgetting** of other capabilities,
 - instability in optimization.
- Full fine-tuning changes **all** the model weights, which can hurt performance on other tasks or out-of-distribution data.
- Soft prompting leaves the base LM intact and only adds a small task-specific adapter; for some regimes this can generalize **better** and preserve performance on other tasks.
- Thus, full fine-tuning is **not guaranteed** to be better than soft prompting, even if you're happy to pay the compute.

(iv) Catastrophic forgetting across tasks A and B

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.

Answer: False.

Reasoning:

- By design, in soft-prompt tuning here the **GPT weights are frozen**.
- You train a soft prompt P_A for Task A; Task B has its own prompt P_B .
- Training P_A changes only the parameters of P_A , not:
 - the base language model,
 - the prompt P_B .
- When you run Task B, you use the same frozen LM and the unchanged P_B . Nothing has been "forgotten" inside the LM.
- So performance on Task B should **not change** just because you tuned P_A . There is no catastrophic forgetting in the base model.

(d) How to adapt MAML-style meta-learning for soft prompts

Goal: family of related tasks $\{T\}$, same frozen GPT, different tasks solved via **task-specific soft prompts**. You have lots of training tasks and want to learn to adapt quickly to a *new* task.

One reasonable MAML-style setup:

(d) How to adapt MAML-style meta-learning for soft prompts

Goal: family of related tasks $\{T\}$, same frozen GPT, different tasks solved via **task-specific soft prompts**. You have lots of training tasks and want to learn to adapt quickly to a *new* task.

One reasonable MAML-style setup:

1. Parameters to meta-learn

- Let the task-specific soft prompt be a vector $p_T \in \mathbb{R}^{K \times E}$.
- MAML learns a shared **initialization** p_0 (meta-parameters) such that a few gradient steps on a new task give a good p_T .

2. Meta-training episode (for one task T)

- Sample a task T from the training family.
- Split its data into:
 - **support set** D_T^{support} (for inner-loop adaptation),
 - **query set** D_T^{query} (for meta-objective).
- **Inner loop (task adaptation):**
 - Start from p_0 .
 - Do a small number of gradient steps on D_T^{support} :

$$p'_T = p_0 - \alpha \nabla_p \mathcal{L}_T(D_T^{\text{support}}; p_0)$$

where \mathcal{L}_T is the cross-entropy loss using the frozen GPT + soft prompt p .

- **Outer loop (meta-update):**

- Evaluate adapted prompt p'_T on the query set:

$$\mathcal{L}_T^{\text{meta}} = \mathcal{L}_T(D_T^{\text{query}}; p'_T)$$

- Compute gradient of $\mathcal{L}_T^{\text{meta}}$ w.r.t. p_0 (backpropagating through the inner-loop update) and update:

$$p_0 \leftarrow p_0 - \beta \nabla_{p_0} \sum_T \mathcal{L}_T^{\text{meta}}$$

© DRAFT -- DO NOT CITE

$$p_0 \leftarrow p_0 - \beta \nabla_{p_0} \sum_T \mathcal{L}_T^{\text{meta}}$$

3. Repeat across many tasks

- Sample many tasks from the family and repeat the episode above.
- Over time, p_0 becomes a good “starting soft prompt” that’s **easy to adapt** to any task in the family.

4. At meta-test time (new task $T^{(*)}$)

- Given a small support set from new task $T^{(*)}$, start from meta-learned p_0 .
- Run a few gradient steps on that support data to get $p'_{T^{(*)}}$.
- Use the frozen GPT + $p'_{T^{(*)}}$ to solve unseen examples from $T^{(*)}$.

This is exactly MAML, with:

- **Model:** frozen GPT + soft prompt,
 - **Meta-parameters:** initial soft prompt p_0 ,
 - **Inner-loop:** task-specific prompt adaptation,
 - **Outer-loop:** optimize p_0 for quick adaptation.
-

Quick self-check

- (a) Shifted indices: reasoning 51–70, answer 71, newline 72 → outputs 50–71. No loss on question or padding: ✓
- (b) Only soft prompt trained, 5 vectors of dim E → 5E parameters: ✓
- (c)
 - (i) Causal attention ⇒ prompt states independent of later tokens ⇒ can precompute: ✓
 - (ii) Soft prompts superset of hard prompts ⇒ “always equal or better” is True: ✓
 - (iii) “Always guaranteed better than soft prompt” is too strong because of overfitting/forgetting ⇒ False: ✓
 - (iv) Base LM frozen, separate prompts ⇒ no catastrophic forgetting across tasks ⇒ False: ✓
- (d) MAML description clearly identifies meta-params, inner/outer loops, and how to adapt to new tasks: ✓

If you want, next we can go through any single subtask in even more detail (e.g., derive (i) with attention equations, or turn (d) into pseudo-code).

The Agent’s response demonstrates a deep and accurate understanding of the concepts related to Question 6 (Soft-Prompting). The methodology used—explicitly showing the step-by-step thought process and concluding with a self-check—is excellent, providing clear, pedagogical explanations. The explanations provided were clear, logical, and accurate across all sub-parts of Question 6.

However, the most significant flaw in the Agent’s response is its **incompleteness**. The Agent entirely failed to address Question 7 (TinyML - Quantization and Pruning), which was present in the Homework Solution document. While the quality of the answers provided for Question 6 is high, the failure to address the entire scope of the material is a critical oversight.

Detailed Comparison (Question 6: Soft-Prompting)

The Agent correctly answered all subparts of Question 6, matching the accuracy of the provided Solution.

(a) Tokens used in the loss calculation

- **Agent's Approach:** The Agent correctly identifies output tokens 50-71. It provides a very clear explanation of the input-to-target shift ($\text{targett}=\text{inputt}+1$) and maps the indices accurately.
- **Solution's Approach:** The Solution also identifies tokens 50-71 but includes a more detailed rationale for *why*certain tokens are excluded. Specifically, it notes that question tokens (5-49) are excluded because "there is no point in training the model to be good at generating questions since at test-time it will be given the question."
- **Evaluation:** The Agent's calculation is excellent. It could be slightly improved by explicitly stating the rationale for masking the loss on the question tokens, matching the nuance of the Solution.

(b) Trainable parameters

- Both the Agent and the Solution correctly identify that 5E parameters are trained. The Agent clearly explains that the rest of the model is frozen and that the dimensions of the frozen base model (S, V, H, L, D) are irrelevant to this count.

(c) True/False Statements

The Agent and the Solution agree on all True/False answers, and the Agent provides strong justifications.

- (i) **Precomputing representations:** The Agent gives a clear explanation based on the mechanics of causal attention (representations at position i only depend on positions $\leq i$), justifying why prompt representations are identical across a batch.
- (ii) **Hard vs. Soft Prompts:** The Agent's explanation is excellent, formally defining hard prompts as a subset of the function class of soft prompts, proving the latter must perform equally or better.
- (iii) **Full Finetuning vs. Soft Prompting:** The Agent correctly identifies this as False, citing risks like overfitting and catastrophic forgetting. The Solution adds a nuance that the Agent missed: in practice, full fine-tuning of highly overparameterized models often results in only a "low-rank update," which sometimes mitigates overfitting.
- (iv) **Catastrophic Forgetting:** The Agent correctly explains that this is False because the base LM weights are frozen, isolating the task-specific prompts.

(d) Adapting MAML

- **Agent's Approach:** The Agent provides a highly structured, algorithmic description of the core MAML process. It clearly defines the meta-parameters (the initialization p_0), the inner loop (task adaptation), the outer loop (meta-update), and the test-time procedure.
- **Solution's Approach:** The Solution describes the core mechanism but also provides an extensive list of advanced variations (e.g., REPTILE-like approaches, global pre-training, keeping a library of initializations).

- **Evaluation:** The Agent's explanation of the fundamental MAML algorithm is clearer and better structured than the Solution's basic description. However, for an open-ended question, the Solution demonstrates a broader understanding of the research landscape by including advanced variations, which the Agent omitted.

Potential Areas for Improvement

To improve the Agent's performance, the following adjustments are recommended:

1. **Ensure Complete Coverage:** The most critical improvement is to ensure the Agent processes and responds to all questions in the provided material. Question 7 should not have been ignored.
2. **Include Explicit Rationales (Q6a):** When determining loss masks, the Agent should explicitly state the reason for excluding tokens (e.g., why the loss is not calculated on the question prompt), rather than just implying it.
3. **Add Depth to Open-Ended Questions (Q6d):** While the Agent's description of MAML was accurate, it should aim to match the comprehensive nature of the Solution by mentioning common variations or extensions to the core algorithm (like First-Order MAML or REPTILE).
4. **Incorporate Practical Nuances (Q6c-iii):** The Agent could enhance its explanations by including practical observations mentioned in the Solution, such as the tendency for full fine-tuning to result in low-rank updates in practice.
5. **Maintain the "Thinking Process":** The step-by-step derivation and self-check are excellent features for clarity and educational value and should be retained.

Question 7

you a an expert in deep learning machine learning. I need u do this step by step do not skip the critical steps after finish u must do a self check if wrong do self-correction



Thought for 7m 18s >

I'll go part by part and be explicit about the logic, then do a quick self-check at the end.

(a) Which output tokens are used in the loss?

Input layout:

- 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

In next-token prediction training, the **target sequence is shifted left by 1**:

$$\text{target}_t = \text{input}_{t+1}$$

They even tell you: answer token is at **position 71** in the input and **position 70** in the target.

We want the loss only for:

They even tell you: answer token is at **position 71** in the input and **position 70** in the target.

We want the loss only for:

- reasoning tokens (positions 51–70),
- the answer (position 71),
- and (optionally) the newline (position 72),
but **not** for the question or padding.

Mapping input → target indices:

- reasoning tokens 51–70 → target positions 50–69
- answer token 71 → target position 70
- newline token 72 → target position 71

So the relevant **output positions** (whose predictions are compared to those targets) are:

Output tokens 50 through 71 (inclusive).

No loss is computed on outputs whose targets are the question (targets 5–49) or padding (targets ≥ 72).

(b) How many parameters are being trained?

The setup: **frozen GPT model**, only the **soft prompt** is trained.

- Soft prompt has **5 vectors**.
- Each vector is directly in embedding space of dimension **E**.
- No other layers get soft prompts; all GPT weights (embeddings, attention, MLPs, LM head, etc.) are frozen.

So the only trainable parameters are those 5 vectors:

$\text{\# trainable parameters} = 5 \times E$

Importantly, this **does not depend on S, V, H, L, D** ; those describe the frozen base model.



(c) True/False with brief explanations

(c) True/False with brief explanations

(i) Precomputing representations for prompt tokens

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 ... once for use in all different training points within a batch.

Answer: True.

Reasoning:

- The model is a **causal** decoder-only Transformer.
- Hidden state at position i in any layer can only attend to positions $\leq i$.
- Prompt tokens are positions 1–5. Their representations at any layer depend only on tokens 1–5 (which are identical for every example in the batch because the soft prompt is shared).
- Therefore, for a *given* value of the soft-prompt parameters (i.e., within one forward/backward pass), the representations of positions 1–5 at each layer are **identical for all sequences in the batch**.
- You can compute them once and reuse (broadcast) them across all examples.

You still need gradients back to these states, but that's fine: all examples' gradients accumulate onto the same prompt parameters.

(ii) Best K-token hard prompt vs best K-vector soft prompt

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.

Answer: True.

Reasoning:

Reasoning:

- A K-token hard prompt chooses K tokens from the vocabulary. This corresponds to picking K **columns** from the embedding matrix. Call their embeddings e_{v_1}, \dots, e_{v_K} .
 - A soft prompt chooses K **arbitrary** vectors in $\mathbb{R}^E: p_1, \dots, p_K$.
 - **Hard prompts are a special case of soft prompts:** for any hard prompt v_1, \dots, v_K , we can set $p_i = e_{v_i}$. With the base model frozen, this produces exactly the same behavior.
 - So the function class for soft prompts \supseteq function class for hard prompts.
Therefore the **best** soft prompt cannot have worse validation performance than the **best** hard prompt; at worst it matches it.
-

(iii) Full fine-tuning vs soft-prompt tuning

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.

Answer: False.

Reasoning:

- "Not constrained by computational cost" only removes a *compute* limitation. It does **not** remove:
 - risk of **overfitting**,
 - risk of **catastrophic forgetting** of other capabilities,
 - instability in optimization.
- Full fine-tuning changes **all** the model weights, which can hurt performance on other tasks or out-of-distribution data.
- Soft prompting leaves the base LM intact and only adds a small task-specific adapter; for some regimes this can generalize **better** and preserve performance on other tasks.
- Thus, full fine-tuning is **not guaranteed** to be better than soft prompting, even if you're happy to pay the compute.

(iv) Catastrophic forgetting across tasks A and B

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.

Answer: False.

Reasoning:

- By design, in soft-prompt tuning here the **GPT weights are frozen**.
 - You train a soft prompt P_A for Task A; Task B has its own prompt P_B .
 - Training P_A changes only the parameters of P_A , not:
 - the base language model,
 - the prompt P_B .
 - When you run Task B, you use the same frozen LM and the unchanged P_B . Nothing has been “forgotten” inside the LM.
 - So performance on Task B should **not change** just because you tuned P_A . There is no catastrophic forgetting in the base model.
-

(d) How to adapt MAML-style meta-learning for soft prompts

Goal: family of related tasks $\{T\}$, same frozen GPT, different tasks solved via **task-specific soft prompts**. You have lots of training tasks and want to learn to adapt quickly to a *new* task.

One reasonable MAML-style setup:

1. Parameters to meta-learn

- Let the task-specific soft prompt be a vector $p_T \in \mathbb{R}^{K \times E}$.
- MAML learns a shared **initialization** p_0 (meta-parameters) such that a few gradient steps on a new task give a good p_T .

new task give a good p_T .

2. Meta-training episode (for one task T)

- Sample a task T from the training family.
- Split its data into:
 - **support set** D_T^{support} (for inner-loop adaptation),
 - **query set** D_T^{query} (for meta-objective).
- **Inner loop (task adaptation):**
 - Start from p_0 .
 - Do a small number of gradient steps on D_T^{support} :

$$p'_T = p_0 - \alpha \nabla_p \mathcal{L}_T(D_T^{\text{support}}; p_0)$$

where \mathcal{L}_T is the cross-entropy loss using the frozen GPT + soft prompt p .

- **Outer loop (meta-update):**

- Evaluate adapted prompt p'_T on the query set:

$$\mathcal{L}_T^{\text{meta}} = \mathcal{L}_T(D_T^{\text{query}}; p'_T)$$

- Compute gradient of $\mathcal{L}_T^{\text{meta}}$ w.r.t. p_0 (backpropagating through the inner-loop update) and update:

$$p_0 \leftarrow p_0 - \beta \nabla_{p_0} \sum_T \mathcal{L}_T^{\text{meta}}$$

3. Repeat across many tasks

- Sample many tasks from the family and repeat the episode above.
- Over time, p_0 becomes a good “starting soft prompt” that’s **easy to adapt** to any task in the family.

4. At meta-test time (new task $T^{\setminus *}$)

- Given a small support set from new task $T^{\setminus *}$, start from meta-learned p_0 .
- Run a few gradient steps on that support data to get $p'_{T^{\setminus *}}$.
- Use the frozen GPT + $p'_{T^{\setminus *}}$ to solve unseen examples from $T^{\setminus *}$.

This is exactly MAML, with:

ramily.

4. At meta-test time (new task $T^{\setminus *}$)

- Given a small support set from new task $T^{\setminus *}$, start from meta-learned p_0 .
- Run a few gradient steps on that support data to get $p'_{T^{\setminus *}}$.
- Use the frozen GPT + $p'_{T^{\setminus *}}$ to solve unseen examples from $T^{\setminus *}$.

This is exactly MAML, with:

- Model:** frozen GPT + soft prompt,
- Meta-parameters:** initial soft prompt p_0 ,
- Inner-loop:** task-specific prompt adaptation,
- Outer-loop:** optimize p_0 for quick adaptation.

Quick self-check

- (a) Shifted indices: reasoning 51–70, answer 71, newline 72 → outputs 50–71. No loss on question or padding: ✓
- (b) Only soft prompt trained, 5 vectors of dim E → $5E$ parameters: ✓
- (c)
 - (i) Causal attention ⇒ prompt states independent of later tokens ⇒ can precompute: ✓
 - (ii) Soft prompts superset of hard prompts ⇒ "always equal or better" is True: ✓
 - (iii) "Always guaranteed better than soft prompt" is too strong because of overfitting/forgetting ⇒ False: ✓
 - (iv) Base LM frozen, separate prompts ⇒ no catastrophic forgetting across tasks ⇒ False: ✓
- (d) MAML description clearly identifies meta-params, inner/outer loops, and how to adapt to new tasks: ✓

If you want, next we can go through any single subpart in even more detail (e.g., derive (i) with attention equations, or turn (d) into pseudo-code).



However, the following suggestions could enhance its depth and clarity even further:

- Nuance on Optimization (Part c-ii):** While the conclusion that the "best possible" soft prompt is superior to the "best possible" hard prompt is theoretically correct (due to the larger function class), the response could acknowledge practical challenges. In reality, optimizing in the high-dimensional continuous space of soft prompts can be difficult and prone to local optima, whereas discrete optimization for hard prompts presents different challenges. Acknowledging this optimization landscape adds practical nuance.
- Technical Depth on MAML (Part d):** The description of MAML is accurate, but it could be enhanced by explicitly mentioning the mechanism of the outer loop update. Specifically, it should highlight that updating p_0 requires backpropagating through the inner-loop optimization steps, which involves calculating second-order derivatives. Mentioning this highlights the computational complexity of MAML and could also briefly reference approximations like First-Order MAML (FO-MAML) used in practice.
- Visualization of Index Shifting (Part a):** The explanation of index shifting is clear, but a simple visualization or table mapping the input indices to the target indices could help clarify the concept, especially for those less familiar with the mechanics of next-token prediction training.

4. **Clarity of Notation:** While the logic is sound, the mathematical notation in the original document is occasionally slightly disorganized