

# **mistral chat**

---

**U** I am going to be feeding you Deep Learning Problems pertaining to the topics of LoRA, Soft Prompting, and Transformers. Please understand the attached homework assignment and we will walk through all of the non-coding problems one by one.

**M** Thank you for sharing the EECS 182 Deep Neural Networks homework document. I've reviewed the material and am ready to help you walk through the non-coding problems step by step. The assignment covers a range of advanced topics, including LoRA (Low-Rank Adaptation), Transformer interpretability, soft prompting, and more.

Let's tackle the problems one by one. **Where would you like to start?**

Here's a quick overview of the main topics and problems:

### **1. LoRA (Low-Rank Adaptation)**

- (a)** Addressing underfitting in LoRA-based finetuning.
- (b)** Problems with initializing both A and B to all zeros.
- (c)** Issues with Xavier uniform initialization for LoRA.
- (d)** Memory requirements for LoRA vs. full-rank adaptation.

### **2. Transformer Interpretability**

- (a)** Zero-layer Transformer: Final logits and baseline functionality.
- (b)** Equivalence of concatenation and additive views of multi-head attention.
- (c)** QK Circuit: Attention patterns and bilinear forms.
- (d)** OV Circuit: Information movement and subspace analysis.
- (e)** SVD analysis of read/write subspaces.

### **6. Soft-Prompting Language Models**

- (a)** Loss computation for soft-prompt tuning.
- (b)** Number of parameters being trained in soft-prompt tuning.
- (c)** True/False questions about soft prompting.
- (d)** Adapting MAML for soft-prompt tuning across related tasks.

**U** Lets start with problem 1. Answer problem 1, including all subproblems of 1, and concisely explain your answers.

**M** **Problem 1: LoRA (Low-Rank Adaptation)**

**(a) Addressing Underfitting in LoRA-based Finetuning**

**Question:**

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?

**Answer:**

If the model is underfitting, it means the LoRA adaptation is not flexible enough to capture the necessary patterns in the new task. To address this, you can:

*one shot correct answer* ↗ **Increase the rank  $k$**  of the LoRA matrices  $A$  and  $B$ . A higher  $k$  increases the capacity of the adaptation, allowing the model to fit more complex patterns.

**Adjust the learning rate** for  $A$  and  $B$  to ensure the updates are neither too small (slow learning) nor too large (unstable training).

**Increase the number of layers** where LoRA is applied, if currently only a subset of layers is being adapted.

**(b) Initializing  $A$  and  $B$  to All Zeros**

**Question:**

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

**Answer:**

If  $A$  and  $B$  are initialized to all zeros, the LoRA adaptation term  $W_0 + AB$  reduces to  $W_0$  at the start of training. This means:

*→ brings in undefined term  $W_0$*

The model starts with the **same output as the frozen pretrained model**, providing no new information or adaptation to the task.

The gradients for  $A$  and  $B$  will initially be zero (since the chain rule will propagate through zero matrices), **preventing any learning** from happening. This is a form of the "vanishing gradient" problem.

*→ good mention of 0 gradients*

The model will fail to adapt to the new task, as the updates to  $A$  and  $B$  will not be triggered.

**(c) Xavier Uniform Initialization for LoRA**

### Question:

Consider the following pseudocode for LoRA initialization:

Python

Copy

```
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?

### Answer:

Xavier uniform initialization is designed for linear layers where the input and output dimensions are large and the weights are randomly distributed to maintain the variance of activations.

However, for LoRA:

The matrices  $A$  and  $B$  are **low-rank** ( $k \ll \min(m, \ell)$ ), so the standard Xavier initialization may not account for the reduced dimensionality.

The product  $AB$  may have a **scaling issue**: Xavier initialization assumes the weights are used in a single linear transformation, but in LoRA, the product  $AB$  is added to the frozen weights  $W_0$ . If the scale of  $AB$  is too large or too small, it can destabilize training or make the adaptation ineffective. *→ good mention of product AB*

The initialization may not preserve the **magnitude relationship** between  $W_0$  and  $AB$ , leading to poor convergence or instability. *→ good mention of model behavior during training*

### (d) Memory Requirements for LoRA

#### Question:

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?

#### Answer:

The memory required for  $A$  is  $m \times k \times 4$  bytes.

The memory required for  $B$  is  $k \times \ell \times 4$  bytes.

**Total memory for LoRA:  $4k(m + \ell)$  bytes.**



For a full-rank adaptation matrix  $W$  of size  $m \times \ell$ , the memory required is  $m \times \ell \times 4$  bytes.

#### Comparison:

LoRA is **memory-efficient** when  $k \ll \min(m, \ell)$ , as it only stores  $4k(m + \ell)$  parameters instead of  $4m\ell$ .

For example, if  $m = \ell = 1024$  and  $k = 4$ , LoRA uses  $4 \times 4 \times (1024 + 1024) = 32,768$  bytes, while a full-rank matrix would use  $1024 \times 1024 \times 4 = 4,194,304$  bytes—a **128x reduction** in memory.

**U** Lets walk through Question 2 now. Answer problem 2, including all subproblems of 2, and concisely explain your answers.

**M Problem 2: Transformer Interpretability**

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

**(i) Mathematical Expression for Final Logits**

**Question:**

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

**Answer:**

The final logits  $\mathbf{L}$  are computed as:

one  
shot correct

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

Answer

$\mathbf{W}_E \mathbf{T}$  maps the one-hot tokens to the embedding space.



$\mathbf{W}_U$  maps the embeddings back to the vocabulary space to produce logits.

unembeddings

**(ii) Algorithm and Information Dependence**

**Question:**

In simple terms, what algorithm does this zero-layer model implement? What information does the prediction for the token at position  $t$  depend on?

**Answer:**

*2* *gut* *?*

**Algorithm:** This zero-layer model implements a lookup table or **non-contextual language model**. It directly maps each input token to a distribution over the vocabulary, without considering the context of other tokens.

**Information Dependence:** The prediction for the token at position  $t$  depends **only on the token at position  $t$  itself** (i.e., its one-hot encoding). There is no interaction or dependency on other tokens in the sequence.

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

#### (i) Equivalence of Formulations

##### Question:

Show that the "concatenation" and "additive and independent" views of multi-head attention are equivalent.

##### Answer:

**Concatenation View:** The outputs of all heads are concatenated:

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

and multiplied by a single large output matrix:

$$\mathbf{H} = \mathbf{W}_O^H \mathbf{R}^H$$

where  $\mathbf{W}_O^H \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}} \cdot H}$ .

**Additive and Independent View:** Each head's output is multiplied by its own output matrix and summed:

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

where  $\mathbf{W}_O^h \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$ .

##### Equivalence:

To make the two formulations equivalent, construct  $\mathbf{W}_O^H$  by horizontally stacking the individual  $\mathbf{W}_O^h$  matrices:

$$\mathbf{W}_O = [\mathbf{W}_O^1 \quad \mathbf{W}_O^2 \quad \dots \quad \mathbf{W}_O^H]$$



Multiplying  $\mathbf{W}_O^H$  by  $\mathbf{R}^H$  yields:

$$\mathbf{H} = \mathbf{W}_O^H \mathbf{R}^H = \sum_{h=1}^H \mathbf{W}_O^h \mathbf{r}^h$$

$\rightarrow$  doesn't formulate  $\mathbf{R}^H$  as a stack of vectors or matrices

which matches the additive and independent view.

## (ii) Advantages and Disadvantages

### Question:

What is an advantage and disadvantage of each view?

### Answer:

#### Concatenation View:

**Advantage:** Efficient for parallel computation, as all heads are processed simultaneously.

**Disadvantage:** Less interpretable, as the contributions of individual heads are not isolated.

#### Additive and Independent View:

**Advantage:** More interpretable, as each head's contribution is explicit and can be analyzed independently.

**Disadvantage:** May require more memory and computation if not optimized, as each head's output is processed separately before summation.

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

### (i) Bilinear Form for Attention Scores

#### Question:

Derive the expression for the pre-softmax attention score  $\mathbf{S}_{ij}$  as a bilinear form involving the residual stream vectors  $\mathbf{X}_i$  and  $\mathbf{X}_j$ , and a single "virtual" weight matrix  $\mathbf{W}_{QK}$ .

#### Answer:

The pre-softmax attention score is:

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

Define the "virtual" weight matrix:

A shot

I think  
correct but  
skip lots  
of steps

$$\mathbf{W}_{QK} = \mathbf{W}_Q^T \mathbf{W}_K \rightarrow \text{doesn't write Transpose}$$

Thus, the attention score can be written as:

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

## (ii) Interpretation of $\mathbf{W}_{QK}$

**Question:**

Interpret the attention patterns for the given  $\mathbf{W}_{QK}$  matrices in a toy scenario where  $d_{\text{model}} = 3$ .

**Answers:**

$$\mathbf{W}_{QK} = \mathbf{I}:$$

The attention score is  $\mathbf{S}_{ij} = \mathbf{X}_i^T \mathbf{X}_j$ , which is the **dot product** of the residual stream vectors.

This head "looks for" tokens with **similar embeddings** (high dot product). ✓

$$\mathbf{W}_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}:$$

The attention score is  $\mathbf{S}_{ij} = \mathbf{X}_{i,1} \mathbf{X}_{j,1}$ , where  $\mathbf{X}_{i,1}$  is the first dimension of  $\mathbf{X}_i$ .

This head only attends to tokens based on the **first dimension** of their embeddings. Information in the second and third dimensions is **ignored**. ✓

$$\mathbf{W}_{QK} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}:$$

The attention score is  $\mathbf{S}_{ij} = \mathbf{X}_{i,1} \mathbf{X}_{j,1} + \mathbf{X}_{i,2} \mathbf{X}_{j,2} - \mathbf{X}_{i,3} \mathbf{X}_{j,3}$ .

This head "looks for" tokens where the first two dimensions are similar, but the third dimension is **opposite** (due to the negative sign). ✓

$$\mathbf{W}_{QK} = -\mathbf{I}:$$

The attention score is  $\mathbf{S}_{ij} = -\mathbf{X}_i^T \mathbf{X}_j$ .

This head "looks for" tokens with **opposite embeddings** (low dot product). ✓

After softmax, this will result in **uniform attention** (since all scores are negative and similar in magnitude), effectively ignoring the input.

1 shot  
correct  
+ good  
explanations

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

### (i) Attention-Weighted Average

**Question:**

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

**Answer:**

The  $i$ -th column of  $\mathbf{X}^0 \mathbf{A}^T$  is:

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

This is a **weighted sum** of the source token vectors  $\mathbf{X}_j^0$ , where the weights are the attention scores  $\mathbf{A}_{ij}$ . ~~→ correct but no explanation/interpretation~~

### (ii) Final Residual Stream Expression

**Question:**

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.

**Answer:**

The final residual stream is:

$$\mathbf{X}^{\text{final}} = \mathbf{X}^0 + \sum_h^H \mathbf{W}_O^h \mathbf{W}_V^h \mathbf{X}^0 \mathbf{A}^{h,T}$$

where  $\mathbf{A}^{h,T}$  is the attention pattern for head  $h$ .

### (iii) Update Vector Lies in Column Space of $\mathbf{W}_{OV}^i$

**Question:**

Prove that the update vector added to the residual stream at destination position  $t$  by head  $i$  must lie within the column space of  $\mathbf{W}_{OV}^i$ .

**Answer:**

The update vector for head  $i$  at position  $t$  is:

$$h_i(\mathbf{X}^0)_t = (\mathbf{W}_{OV}^i \mathbf{X}^0 \mathbf{A}^{i,T})_t$$

Since  $\mathbf{W}_{OV}^i \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$ , its column space is a  $d_{\text{head}}$ -dimensional subspace of  $\mathbb{R}^{d_{\text{model}}}$ . The output  $h_i(\mathbf{X}^0)_t$  is a linear combination of the columns of  $\mathbf{W}_{OV}^i$ , so it must lie within the column space of  $\mathbf{W}_{OV}^i$ . ~~→ skips a lot of steps~~

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

## (i) Rank of $\mathbf{W}$

**Question:**  $\mathbf{W}_{OV}$

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

**Answer:**

$$\mathbf{W}_{OV} = \mathbf{W}_O \mathbf{W}_V, \text{ where } \mathbf{W}_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}} \text{ and } \mathbf{W}_V \in \mathbb{R}^{d_{\text{head}} \times d_{\text{model}}}.$$

The rank of  $\mathbf{W}_{OV}$  is at most the minimum of the ranks of  $\mathbf{W}_O$  and  $\mathbf{W}_V$ , which is  $d_{\text{head}}$ .

In typical transformers,  $d_{\text{head}} \ll d_{\text{model}}$ , so  $\mathbf{W}_{OV}$  is **low-rank**.

good explanation

## (ii) SVD Decomposition

**Question:**

Show that the output of the OV circuit can be decomposed using the SVD of  $\mathbf{W}_{OV}$ .

**Answer:**

Given the SVD  $\mathbf{W}_{OV} = \mathbf{U} \Sigma \mathbf{V}^T$ , the output for a source token  $\mathbf{X}_{\text{src}}$  is:

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

skip multiplying form steps

where  $r = \text{rank}(\mathbf{W}_{OV})$ .

## (iii) Read and Write Subspaces

**Question:**

Define the read and write subspaces.

**Answers:**

**Read Subspace:** The subspace spanned by the columns of  $\mathbf{V}$ , i.e.,  $\text{span}(\mathbf{V}_1, \dots, \mathbf{V}_r)$ . The head can only extract information from directions in  $\mathbf{X}_{\text{src}}$  that lie within this subspace.

**Write Subspace:** The subspace spanned by the columns of  $\mathbf{U}$ , i.e.,  $\text{span}(\mathbf{U}_1, \dots, \mathbf{U}_r)$ . The head can only write information to directions in the residual stream that lie within this subspace.

**Orthogonal Information:** Information in  $\mathbf{X}_{\text{src}}$  orthogonal to the read subspace is ignored by the head.

## (iv) Relation to Value Projection

**Question:**

Explain how the read subspace relates to what  $\mathbf{W}_V$  can "see" in the source token.

**Answer:**

$\mathbf{W}$  projects the source token  $\mathbf{X}_{src}$  to a  $d_{head}$ -dimensional value vector.

The read subspace  $\text{span}(\mathbf{V}_1, \dots, \mathbf{V}_r)$  is the subspace of  $\mathbf{X}_{src}$  that  $\mathbf{W}_V$  can "see" or extract information from.

Since  $r \leq d_{head}$ , the read subspace dimension is at most  $d_{head}$ .

### (v) SVD as Information Movement

**Question:**

Explain how the SVD provides a complete characterization of information movement.

**Answer:**

$\mathbf{V}$ : Determines **what information is read** from the source token (read subspace).

$\Sigma$ : Scales the **importance** of each direction in the read subspace.

$\mathbf{U}$ : Determines **where the information is written** in the destination residual stream (write subspace).

The attention pattern  $\mathbf{A}$  determines **which source tokens** this transformation is applied to and with what weights. The SVD and  $\mathbf{A}$  together fully characterize the information movement.

good explanations here

U Lets now walk through Question 5, which works with Fermi Estimation. Walk through each subpart of problem 5, using relevant plots from the PDF assignment when necessary. Briefly explain your answers.

provide some context here about Fermi:

### M Problem 5: Fermi Estimation for Large-Scale Deep Learning Models

Fermi estimation is about making rough but reasonable approximations to estimate quantities. Here, we apply it to a hypothetical GPT-6 with **100 trillion parameters**.

#### (a) GPT-like AGI: Number of Layers

**Context:**

The human brain takes about **100 steps** (based on the "hundred-step-rule") to make a snap decision.

GPT-3 has **96 layers**, which aligns well with this rule.

The brain has  $10^{15}$  synapses, and each synapse is estimated to have about **5 bits** of precision (storable in a 16-bit float).

A GPT-like AGI might need  $10^{15}$  parameters, but this problem focuses on **100 trillion parameters**.

### Key Takeaway:

The number of layers in GPT-6 is assumed to be similar to GPT-3, i.e., **~100 layers**, to match the brain's "hundred-step-rule."

→ good strategy here

### (b) Chinchilla Scaling Law

#### Context:

The Chinchilla paper provides a scaling law for optimal model size  $N$  and dataset size  $D$  given a compute budget  $C$ :

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

to make sure LLM knows important facts.

where:

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

$$A = 406.4, B = 410.7, L_0 = 1.69,$$

$$C = C_0 ND, \text{ with } C_0 = 6 \text{ FLOPs per parameter per token.}$$

#### Question:

Solve for  $N_{\text{opt}}$  and  $D_{\text{opt}}$  as a function of  $C$ .

#### Answer:

To minimize  $L$  under the constraint  $C_0 ND = C$ , we use the method of Lagrange multipliers or substitution. The optimal values are:

$$N_{\text{opt}} \propto C^{0.5}, \quad D_{\text{opt}} \propto C^{0.5}$$

The paper provides a direct fit:

doesn't show subst. into  
the chinchilla scaling  
law.

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

## Key Takeaway:

For a given compute budget  $C$ , the optimal model size and dataset size scale as the square root of  $C$ .

### (c) Compute and Dataset Size for GPT-6

#### Question:

Suppose GPT-6 has  $N = 10^{14}$  parameters. Using the Chinchilla scaling laws, how much compute  $C$  is required, and how many tokens  $D$  are needed for training?

#### Answer:

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

$$10^{14} = 0.1C^{0.5} \implies C = (10^{15})^2 = 10^{30} \text{ FLOPs}$$

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

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

✓ *One  
shot  
corr,*

## Key Takeaway:

Training GPT-6 requires  $10^{30}$  FLOPs of compute and a dataset of **1.7 quadrillion tokens**.

### (d) Dataset Size in Context

#### Question:

How many English words and books are equivalent to 1.7 quadrillion tokens?

Assume 1.4 tokens per English word.

Assume 400 words per page and 300 pages per book.

#### Answer:

**Words:**  $1.7 \times 10^{15} \text{ tokens} / 1.4 \approx 1.2 \times 10^{15} \text{ words}$ .

**Books:**  $1.2 \times 10^{15} \text{ words} / (400 \times 300) \approx 10^{10} \text{ books}$ .

#### Context:

The Library of Congress holds about **38 million books**. → Hallucination

Google Books has scanned about **40 million books**. ✓

**1.7 quadrillion tokens** is equivalent to ~**10 billion books**, which is **250x** larger than the Library of Congress.

**more around 6 bill** → **less hallucination**

### Key Takeaway:

The dataset required for GPT-6 is **orders of magnitude larger** than all existing digitized books combined.

**→ correct takeaway.**

### (e) Memory Requirement

#### Question:

How much memory is required to store GPT-6 during training?

Assume 16-bit floating-point parameters.

#### Answer:

**Memory for 1 billion parameters:**  $10^9 \times 16 \text{ bits} = 2 \text{ GB}$ .

**Memory for 100 trillion parameters:**  $10^{14} \times 16 \text{ bits} = 2 \times 10^6 \text{ GB} = 2 \text{ exabytes}$ .

**Number of H200 GPUs (100 GB VRAM each):**  $2 \times 10^6 \text{ GB} / 100 \text{ GB} = 20,000 \text{ GPUs}$ .

### Key Takeaway:

Storing GPT-6 requires **2 exabytes of memory**, which would need **20,000 H200 GPUs** just for the parameters.

### (f) Memory Cost

#### Question:

What is the annual cost to store GPT-6 on SSD and DRAM in 2025?

#### Answer:

##### **SSD Cost (2025):**

$0.00005 \text{ per MB} \cdot 2 \times 10^6 \text{ GB} = 2 \times 10^9 \text{ MB} \cdot \text{Annual cost} : 2 \times 10^9 \times 0.00005 = 100,000 \text{ \$}$ .

##### **DRAM Cost (2025):**

$0.0040 \text{ per MB} \cdot \text{Annual cost} : 2 \times 10^9 \times 0.0040 = 8,000,000 \text{ \$}$ .

## **Key Takeaway:**

Storing GPT-6 on **SSD costs**  $\$100K/year$ , while **DRAM costs**  $\$8M/year$ .

## **(g) Memory Bandwidth and Latency**

### **Question:**

What is the minimal latency for a single forward pass of GPT-6 on an H200 GPU?

H200 memory bandwidth: 4.8 TB/s.

### **Answer:**

**Memory to load:**  $2 \times 10^6 \text{ GB} = 2 \times 10^6 \times 1024^3 \text{ bytes} \approx 2.19 \times 10^{18} \text{ bytes}$ .

**Time to load:**  $2.19 \times 10^{18} \text{ bytes} / (4.8 \times 10^{12} \text{ bytes/s}) \approx 456 \text{ seconds}$ .

**Tokens per minute (autoregressive):**  $60 / 456 \approx 0.13 \text{ tokens/minute}$ .

### **Context:**

For **GPT-3 (175B parameters)**, the latency is:  $175 \times 10^9 \times 2 \text{ bytes} = 350 \text{ GB}$ . Time to load:  $350 \times 10^9 / (4.8 \times 10^{12}) \approx 0.073 \text{ seconds}$ . Tokens per minute:  $60 / 0.073 \approx 822 \text{ tokens/minute}$ .

## **Key Takeaway:**

GPT-6 would generate **0.13 tokens/minute** on a single GPU, while GPT-3 generates **822 tokens/minute**.

## **(h) Training Cost**

### **Question:**

How much does it cost to train GPT-6 using H200 GPUs?

H200 specs:

Rental price: \\$1.50/hour.

Speed: 1.98 petaFLOP/s.

1 petaFLOP-day =  $10^{15} \times 86400 = 8.64 \times 10^{19}$  FLOPs.

H200-hour per petaFLOP-day:  $8.64 \times 10^{19} / (1.98 \times 10^{15} \times 3600) \approx 12.1$  hours.

Cost per petaFLOP-day:  $12.1 \times 1.50 = \$18.15$ .

GPT-3 training cost: 3640 petaFLOP-days. Cost:  $3640 \times 18.15 \approx \$66,000$ .

GPT-6 training cost:  $10^{30}$  FLOPs =  $10^{15}$  petaFLOP-days. Cost:  $10^{15} \times 18.15 \approx \$1.8 \times 10^{16}$ .

### Context:

The **OPT-175B** training run took **3 months** but would have taken **33 days** without restarts.

Hardware failures and restarts can **2-3x** the cost.

### Key Takeaway:

Training GPT-6 would cost **\\$18 quadrillion** at current GPU rental prices, which is **infeasible**.

For context:

iPhone 1 development: \\$150M.

Airbus A380: \\$18B.

Manhattan Project: \\$24B.

Apollo Program: \\$178B.

GPT-6 training cost is **100,000x larger** than the Apollo Program.

### Wall Clock Time:

With **30 million H200s**, perfect utilization, and no interruptions:  $10^{30} \text{ FLOPs} / (30 \times 10^6 \times 1.98 \times 10^{15} \text{ FLOPs/s}) \approx 1.7 \times 10^8 \text{ seconds} \approx 5.4 \text{ years}$ .

### Key Takeaway:

Even with **30 million GPUs**, training GPT-6 would take **5.4 years** of wall clock time.

## (i) Difficulty of Large-Scale Training

### Context:

Large models often **diverge** or encounter hardware failures, requiring restarts.

The **OPT-175B** training had **90 restarts** over 3 months.

Issues included:

- Gradient explosions.

- Hardware failures (missing GPUs, CUDA errors, etc.).

- Dataset issues (e.g., unreasonably low perplexity).

### Key Takeaway:

Training large models is **extremely challenging** due to instability, hardware failures, and dataset issues.

The actual cost is likely **3x higher** due to failures and restarts.

## (j) Inference Cost

### Question:

How much does it cost to infer 1 million tokens using GPT-5 (1T parameters)?

Inference cost: 2 FLOPs per parameter per token.

H200 rental price: \\$1.50/hour.

### Answer:

**FLOPs for 1M tokens:**  $10^{12} \times 2 \times 10^6 = 2 \times 10^{18}$  FLOPs.

**Time on H200:**  $2 \times 10^{18} / (1.98 \times 10^{15}) \approx 1010$  seconds  $\approx 16.8$  minutes.

**Cost:**  $16.8 / 3600 \times$

$1.50 \approx 0.007$ .

### Context:

OpenAI charges **\\$120 per 1M tokens** for GPT-4.

To recoup a  $\backslash 1B$  training cost, OpenAI would need to sell  $: 10^9 / 120 \approx 8.3$  million \$ batches of 1M tokens.

**Equivalent essays:**  $10^6 \text{ tokens} / (1.4 \times 1000) \approx 714$  essays.

### Key Takeaway:

Inference is **cheaper** than training but still costly at scale.

### (k) Energetic Cost

#### Question:

What is the minimal energy required for one FLOP at room temperature (Landauer limit)?

Landauer limit:  $E = k_B T \ln 2$ , where  $k_B = 1.38 \times 10^{-23}$  J/K,  $T = 300$  K.

Minimal energy for one FLOP:  $32k_B T \ln 2$ .

#### Answer:

$$E = 32 \times 1.38 \times 10^{-23} \times 300 \times \ln 2 \approx 9.2 \times 10^{-21} \text{ J.}$$

**Minimal power for H200 (1980 TFLOP/s):**  $1980 \times 10^{12} \times 9.2 \times 10^{-21} \approx 1.8 \times 10^{-6}$  W.

**Actual H200 power:** 700 W.

#### Key Takeaway:

The **Landauer limit is 12 orders of magnitude lower** than the actual power consumption of GPUs.

### (l) Environmental Cost

#### Question:

What is the carbon emission cost of training GPT-3?

GPT-3 emissions: 552 tCO<sub>2</sub>.

Carbon cost: \\$112 per tCO<sub>2</sub>.

A100 training cost: \\$6M.

#### Answer:

**Carbon cost:**  $552 \times 112 = \$61,824$ .

**Total cost:** \\$6M (compute) + \\$61,824 (carbon)  $\approx \$6.06M$ .

#### Context:

For this question, there were some LLM hallucinations with a few assumptions, resulting in some miscalculations, but the overall methodology is correct.

**Beef emissions:** 50 kg CO<sub>2</sub> per kg of beef.

**Burgers equivalent to GPT-3 emissions:**

552,000 kg CO<sub>2</sub>/0.0568 kg CO<sub>2</sub> per burger ≈ 9.7 million burgers.

### Key Takeaway:

The carbon footprint of training GPT-3 is equivalent to **9.7 million beef burgers**.

U

Let's now walk through Question 6, repeating the same procedures as above.

M

### Problem 6: Soft-Prompting Language Models

Soft-prompting is a technique to adapt frozen language models to new tasks by prepending **learnable vector embeddings** (soft prompts) to the input, rather than using fixed token prompts (hard prompts). This approach allows the model to be fine-tuned without updating its original weights.

#### (a) Loss Computation for Soft-Prompt Tuning

##### Question:

For the given input sequence:

Copy

["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]

Which output tokens are used to compute the loss for training the soft prompt?

##### Answer:

The **target sequence** is shifted by one position relative to the input sequence.

The loss is computed for **all non-padding tokens** in the target sequence, which includes:

Tokens 6-50 (question),

Tokens 51-70 (chain of thought reasoning),

Token 71 (answer).

→ needed further because at test time it will be given a question

## Key Point:

The soft prompt itself is **not** part of the loss computation. Only the tokens corresponding to the question, reasoning, and answer are used to compute the cross-entropy loss.

## (b) Number of Parameters Being Trained

### Question:

How many parameters are being trained in this model?

### Answer:

The soft prompt consists of **5 learnable vectors**, each of dimension  $E$  (embedding size).

**Total trainable parameters:**  $5 \times E$ .



**One Shot**

### Key Point:

The rest of the model (e.g., transformer layers, token embeddings) remains **frozen**. Only the soft prompt vectors are updated during training.

~~→ good extra explanation~~

## (c) True/False Questions About Soft Prompting

### (i) Precomputing Representations for Prompt Tokens

#### Statement:

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

#### Answer:

True.



The soft prompt tokens are **fixed** for all inputs in a batch.

Their representations at each layer can be **precomputed once** and reused for all batch elements, saving computation.

### (ii) Soft Prompt vs. Hard Prompt Performance

#### Statement:

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.



**A hard prompt** is limited to discrete tokens from the vocabulary.

**A soft prompt** can take any value in the continuous embedding space, giving it **greater expressivity** and the ability to find better task-specific representations.

### (iii) Full Fine-Tuning vs. Soft Prompt Tuning

#### **Statement:**

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

#### **Answer:**

False.

While full fine-tuning has **higher capacity**, it is not guaranteed to outperform soft prompting in all cases.

Soft prompting can **avoid catastrophic forgetting** and may generalize better to related tasks, especially when training data is limited.

### (iv) Catastrophic Forgetting in Soft Prompting

#### **Statement:**

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.

Soft prompting **does not update the original model weights**, so it **cannot cause catastrophic forgetting** for Task B.

Each task can have its own soft prompt without interfering with others.

### (d) Adapting MAML for Soft-Prompt Tuning Across Related Tasks

#### **Question:**

How would you adapt a meta-learning approach like **MAML (Model-Agnostic Meta-Learning)** for soft-prompt tuning across a family of related tasks?

#### **Answer:**

**MAML for Soft-Prompt Tuning:**

### **Meta-Training Phase:**

Sample a batch of tasks from the family.

For each task, **initialize a soft prompt** and adapt it using gradient descent on the task's training data.

Compute the **meta-gradient** to update the **initial soft prompt** such that it can quickly adapt to any task in the family with minimal updates.

### **Meta-Testing Phase:**

For a new, unseen task, use the **meta-learned initial soft prompt** and fine-tune it on the task's training data.

The goal is to achieve **fast adaptation** with few updates.

### **Key Steps:**

**Inner Loop:** Adapt the soft prompt to each task using gradient descent.

**Outer Loop:** Update the initial soft prompt to minimize the adaptation loss across all tasks.

### **Advantages:**

Enables **few-shot learning** for new tasks.

Avoids catastrophic forgetting by keeping the base model frozen.

The meta-learned soft prompt serves as a **good initialization** for any task in the family.

### **Key Point:**

MAML is used to learn an **initial soft prompt** that is easily adaptable to any task in the family, rather than learning model weights.

good initialization, aligns with the purpose of MAML