

Homework 3 - Grok AI LLM Conversation

Bruno Vieira
3038484295

Week: 09/19/25 - 09/26/25

Conversation

1)

Me:

I will give you a 3-part machine learning / math question about a dense layer and RMS norms. Do not compute anything yet; only outline the steps and the variables you will treat as random and which as fixed. I will give you the question soon.

Grok:

Understood! Please provide the 3-part machine learning/math question about a dense layer and RMS norms, and I'll outline the steps to solve it along with specifying which variables I'll treat as random and which as fixed. To ensure clarity, I'll wait for your question and then:

- Identify the key components of each part of the question.
- Outline the steps required to address each part systematically.
- Specify which variables are treated as random (e.g., inputs, weights, or noise terms) and which are fixed (e.g., layer dimensions, constants, or hyperparameters).
- Avoid any computations until you provide specific values or further instructions.

Please go ahead and share the question!

1) (a)

Me:

Consider a dense layer with input $x \in \mathbb{R}^{d_1}$, weights $W \in \mathbb{R}^{d_2 \times d_1}$, where W is initialized with i.i.d. standard Gaussian entries, and entries of x are sampled from an i.i.d. unit Gaussian. What is the expected squared RMS norm of the output features $y = Wx$? How does this scale with d_1 or d_2 ? What constant should we multiply W by to ensure that the expected squared RMS norm of Wx is 1, regardless of d_1 and d_2 ?

Hint: Consider a simplified dense layer with a single output feature, $W \in \mathbb{R}^{1 \times d_1}$. What is the distribution of the scalar $y = Wx$? What is its variance?

Grok:

Part 1: Expected Squared RMS Norm of $y = Wx$

The RMS norm of the output vector $y \in \mathbb{R}^{d_2}$ is defined as

$$\|y\|_{\text{RMS}} = \sqrt{\frac{1}{d_2} \sum_{j=1}^{d_2} y_j^2}.$$

Thus, the squared RMS norm is

$$\|y\|_{\text{RMS}}^2 = \frac{1}{d_2} \sum_{j=1}^{d_2} y_j^2.$$

To find the expected value $E\left[\frac{1}{d_2} \sum_{j=1}^{d_2} y_j^2\right]$:

$$y_j = \sum_{k=1}^{d_1} W_{jk} x_k.$$

Since $W_{jk} \sim N(0, 1)$ and $x_k \sim N(0, 1)$ i.i.d. and independent, each term $W_{jk}x_k$ has variance

$$\text{Var}(W_{jk}x_k) = E[W_{jk}^2]E[x_k^2] = 1.$$

The terms are independent across k , so

$$\text{Var}(y_j) = \sum_{k=1}^{d_1} 1 = d_1.$$

Since y_j is a sum of independent Gaussian products, $y_j \sim N(0, d_1)$, and

$$E[y_j^2] = \text{Var}(y_j) = d_1.$$

Therefore,

$$E \left[\frac{1}{d_2} \sum_{j=1}^{d_2} y_j^2 \right] = \frac{1}{d_2} \sum_{j=1}^{d_2} E[y_j^2] = d_1.$$

Part 2: Scaling with d_1 or d_2

The expectation scales linearly with d_1 and is independent of d_2 .

Part 3: Constant to Multiply W By

Let $W' = cW$, so $y' = W'x = cy$. Then

$$\|y'\|_{\text{RMS}}^2 = c^2 \|y\|_{\text{RMS}}^2.$$

To ensure $E[\|y'\|_{\text{RMS}}^2] = 1$, set

$$c^2 d_1 = 1 \quad \Rightarrow \quad c = \frac{1}{\sqrt{d_1}}.$$

Me:

Good work, can you expand on why $E[\text{squared RMS of } y'] = c^2 E[\text{squared RMS of } y]$?

Grok:

To explain why

$$E[\|y'\|_{\text{RMS}}^2] = c^2 E[\|y\|_{\text{RMS}}^2],$$

let's break it down step-by-step.

Definitions and Setup

The output of the dense layer is $y = Wx$, with $x \in \mathbb{R}^{d_1}$ and $W \in \mathbb{R}^{d_2 \times d_1}$, with i.i.d. Gaussian entries. The RMS norm is

$$\|y\|_{\text{RMS}} = \sqrt{\frac{1}{d_2} \sum_{j=1}^{d_2} y_j^2}, \quad \|y\|_{\text{RMS}}^2 = \frac{1}{d_2} \sum_{j=1}^{d_2} y_j^2.$$

Consider a scaled weight matrix $W' = cW$, giving $y' = W'x = cy$. Then each component scales as $y'_j = cy_j$, and the squared RMS norm is

$$\|y'\|_{\text{RMS}}^2 = \frac{1}{d_2} \sum_{j=1}^{d_2} (y'_j)^2 = \frac{1}{d_2} \sum_{j=1}^{d_2} (cy_j)^2 = c^2 \|y\|_{\text{RMS}}^2.$$

Taking expectations:

$$E[\|y'\|_{\text{RMS}}^2] = E[c^2\|y\|_{\text{RMS}}^2] = c^2 E[\|y\|_{\text{RMS}}^2],$$

since c^2 is a constant.

Verification with Single Output Hint

For $W \in \mathbb{R}^{1 \times d_1}$, $y = \sum_{k=1}^{d_1} W_k x_k$, $\text{Var}(y) = d_1$, so $E[y^2] = d_1$. Scaling gives $y' = cy$, hence $E[(y')^2] = c^2 d_1$, consistent with the multi-output case.

Conclusion

Scaling the weights by c scales the output vector linearly, and the squared RMS norm introduces a factor of c^2 . The expectation preserves this scaling, making

$$E[\|y'\|_{\text{RMS}}^2] = c^2 E[\|y\|_{\text{RMS}}^2].$$

1) (b)

Me:

Good work, now on to the next question: We will now consider how to ensure that RMS norms of features update at a constant rate during training, again regardless of layer width. Assume we are using the SignGD optimizer (which is a simplified version of Adam). Unlike at initialization, where we assume weights and inputs are independent, the updates to weights during training are very much correlated with the inputs. For simplicity, assume that the minibatch training input $x_i \in \mathbb{R}^{d_1}$ are sampled from an i.i.d. unit Gaussian, and the raw gradient $\nabla_W f(W)$ is an outer product of the input and a constant backpropagated vector $g_i \in \mathbb{R}^{d_2}$.

$$W_{t+1} \leftarrow W_t + \eta \text{sign}(x_i g_i^T).$$

What is the expected RMS norm squared of the (unscaled) change in features $y' = \text{sign}(x_i g_i^T) x_i$? How does this scale with d_1 or d_2 ? What constant should we multiply the update by to ensure that the expected RMS norm of y' is 1, regardless of d_1 and d_2 ?

Grok:

Part 1: Expected Squared RMS Norm of $y' = \text{sign}(g_i x_i^T) x_i$

To compute the expected squared RMS norm

$$E[\|y'\|_{\text{RMS}}^2],$$

where the expectation is over the random input $x_i \sim N(0, I_{d_1})$ (i.i.d. unit Gaussian components), and $g_i \in \mathbb{R}^{d_2}$ is a fixed (constant) vector with all nonzero components:

The raw gradient is

$$\nabla_W f(W) = g_i x_i^T \in \mathbb{R}^{d_2 \times d_1}, \quad (g_i x_i^T)_{jk} = g_{i,j} x_{i,k}.$$

The sign is applied elementwise:

$$\text{sign}(g_{i,j} x_{i,k}) = \text{sign}(g_{i,j}) \text{sign}(x_{i,k}),$$

since $\text{sign}(ab) = \text{sign}(a)\text{sign}(b)$ for $a, b \in \mathbb{R}$ (assuming no zeros, which have probability zero for Gaussians).

Thus,

$$y'_j = \sum_{k=1}^{d_1} \text{sign}(g_{i,j} x_{i,k}) x_{i,k} = \text{sign}(g_{i,j}) \sum_{k=1}^{d_1} \text{sign}(x_{i,k}) x_{i,k}.$$

Note that $\text{sign}(x_{i,k}) x_{i,k} = |x_{i,k}|$. Let

$$z = \sum_{k=1}^{d_1} |x_{i,k}|,$$

then $y'_j = \text{sign}(g_{i,j}) z$. Since all $g_{i,j} \neq 0$, $\text{sign}(g_{i,j})^2 = 1$ for all j .

The squared L_2 norm is

$$\|y'\|_2^2 = \sum_{j=1}^{d_2} y_j'^2 = \sum_{j=1}^{d_2} \text{sign}(g_{i,j})^2 z^2 = d_2 z^2.$$

The squared RMS norm is

$$\|y'\|_{\text{RMS}}^2 = \frac{1}{d_2} \|y'\|_2^2 = z^2.$$

Therefore,

$$E[\|y'\|_{\text{RMS}}^2] = E[z^2].$$

Each $|x_{i,k}|$ is half-normal:

$$E[|x_{i,k}|] = \sqrt{\frac{2}{\pi}}, \quad E[|x_{i,k}|^2] = 1 \quad (\text{since } |x|^2 = x^2).$$

Since the $|x_{i,k}|$ are i.i.d.,

$$E[z^2] = \text{Var}(z) + (E[z])^2 = d_1 \left(1 - \frac{2}{\pi}\right) + d_1^2 \frac{2}{\pi} = d_1 + \frac{2}{\pi} d_1 (d_1 - 1).$$

Alternatively, directly:

$$E[z^2] = E \left[\left(\sum_k |x_k| \right)^2 \right] = \sum_k E[|x_k|^2] + \sum_{k \neq l} E[|x_k|] E[|x_l|] = d_1 \cdot 1 + d_1(d_1 - 1) \cdot \frac{2}{\pi}.$$

The expected squared RMS norm is

$$d_1 + \frac{2}{\pi} d_1(d_1 - 1).$$

Part 2: Scaling with d_1 or d_2

From the expression above:

- The expectation scales quadratically with d_1 (the leading term is $\frac{2}{\pi} d_1^2$ for large d_1).
- It does not depend on d_2 (constant with respect to d_2). This follows because z depends only on the input dimension d_1 , and the averaging in the RMS norm cancels the d_2 dependence when all components of g_i are nonzero.

Part 3: Constant to Multiply the Update By for Expected RMS Norm of 1

Let the scaled update be $y'' = cy'$, where c is the constant multiplier on the update.

The RMS norm scales linearly:

$$\|y''\|_{\text{RMS}} = |c| \|y'\|_{\text{RMS}} \quad (\text{assuming } c > 0).$$

From above, $\|y'\|_{\text{RMS}} = \sqrt{z^2} = z$ (since $z > 0$), so

$$E[\|y''\|_{\text{RMS}}] = cE[z].$$

$$E[z] = d_1 \sqrt{\frac{2}{\pi}}.$$

To ensure $E[\|y''\|_{\text{RMS}}] = 1$ regardless of d_1 and d_2 , set

$$cE[z] = 1 \quad \Rightarrow \quad c = \frac{1}{d_1 \sqrt{\frac{2}{\pi}}} = \frac{\sqrt{\pi/2}}{d_1}.$$

Note that this is independent of d_2 , as expected from Part 2.

1) (c)

Me:

That looks good as well good job! The final question of the subpart is: You may notice that the above update rule only depends on d_1 . Why is this the case?

Grok:

The update rule for the expected squared RMS norm of the change in features

$$y' = \text{sign}(g_i x_i^T) x_i$$

depends only on d_1 and not on d_2 . Let's explore why this is the case by analyzing the mathematical structure of the problem.

Key Components:

- The input $x_i \in \mathbb{R}^{d_1}$ has i.i.d. unit Gaussian entries, $x_{i,k} \sim N(0, 1)$.
- The backpropagated vector $g_i \in \mathbb{R}^{d_2}$ is a fixed (constant) vector with all nonzero components.
- The raw gradient is

$$\nabla_W f(W) = g_i x_i^T \in \mathbb{R}^{d_2 \times d_1},$$

and the update is based on its sign:

$$\text{sign}(g_i x_i^T).$$

- The change in features is

$$y' = \text{sign}(g_i x_i^T) x_i \in \mathbb{R}^{d_1}.$$

- The squared RMS norm is

$$\|y'\|_{\text{RMS}}^2 = \frac{1}{d_1} \sum_{j=1}^{d_1} y_j'^2,$$

and we compute its expectation over the randomness in x_i .

Derivation of y' and RMS Norm:

For each component of y' :

$$y'_j = \sum_{k=1}^{d_1} \text{sign}(g_{i,j}x_{i,k})x_{i,k}.$$

Since

$$\text{sign}(g_{i,j}x_{i,k}) = \text{sign}(g_{i,j})\text{sign}(x_{i,k}),$$

and $g_{i,j}$ is a fixed nonzero constant, we get:

$$y'_j = \text{sign}(g_{i,j}) \sum_{k=1}^{d_1} |x_{i,k}|.$$

Define

$$z = \sum_{k=1}^{d_1} |x_{i,k}|,$$

which depends only on the input dimension d_1 and the random variables $x_{i,k}$.

Thus,

$$y'_j = \text{sign}(g_{i,j})z, \quad \text{and since } \text{sign}(g_{i,j})^2 = 1, \quad y_j'^2 = z^2.$$

The squared L_2 norm is:

$$\|y'\|_2^2 = \sum_{j=1}^{d_2} y_j'^2 = \sum_{j=1}^{d_2} z^2 = d_2 z^2.$$

The squared RMS norm is:

$$\|y'\|_{\text{RMS}}^2 = \frac{1}{d_2} \|y'\|_2^2 = \frac{1}{d_2} \cdot d_2 z^2 = z^2.$$

The expected squared RMS norm is:

$$E[\|y'\|_{\text{RMS}}^2] = E[z^2] = E \left[\left(\sum_{k=1}^{d_1} |x_{i,k}| \right)^2 \right].$$

Since $z = \sum_{k=1}^{d_1} |x_{i,k}|$, and $|x_{i,k}| \sim N(0, 1)$ half-normal, the expectation $E[z^2]$ depends only on d_1 , the dimension of the input.

Why No Dependence on d_2 :

- **Role of g_i :** The vector g_i affects only the sign of each component y'_j . The magnitude $y_j'^2 = z^2$ is independent of $g_{i,j}$'s specific values, as long as they are nonzero. The dimension d_2 influences the number of output components but not their individual magnitudes.
- **Averaging in RMS Norm:** The RMS norm computes the mean of squared components across d_2 . Each $y_j'^2 = z^2$, so summing over d_2 and dividing by d_2 cancels the d_2 dependence.
- **Randomness in x_i :** The variable z is a function of d_1 alone. Its moments (like $E[z^2]$) depend only on d_1 .
- **Structure of the Update:** The update $\text{sign}(g_i x_i^T) x_i$ produces components y'_j that all share the same random variable z , scaled by deterministic signs. This ensures RMS averaging eliminates d_2 dependence.

Intuition:

The lack of d_2 dependence arises because the RMS norm normalizes by d_2 . Each component y'_j has the same magnitude z , so the average squared magnitude across d_2 is simply z^2 , which depends only on d_1 . The backpropagated vector g_i only determines the direction (via signs), not the magnitude.

Verification:

Compute:

$$z = \sum_{k=1}^{d_1} |x_{i,k}|, \quad E[|x_{i,k}|] = \sqrt{\frac{2}{\pi}}, \quad E[|x_{i,k}|^2] = E[x_{i,k}^2] = 1$$

$$E[z^2] = \sum_{k=1}^{d_1} E[|x_{i,k}|^2] + \sum_{k \neq l} E[|x_{i,k}|] E[|x_{i,l}|] = d_1 \cdot 1 + d_1(d_1 - 1) \cdot \frac{2}{\pi},$$

consistent with

$$E[\|y'\|_{\text{RMS}}^2] = d_1 + \frac{2}{\pi} d_1(d_1 - 1).$$

Conclusion:

The expected squared RMS norm depends only on d_1 because the magnitude of the update is determined by z , a function of the input dimension d_1 . The output dimension d_2 is canceled out by RMS averaging, and the fixed vector g_i only contributes signs, not magnitudes, to y' .

Me:

Great work, next time make sure your work is less messy. Less is more sometimes. For example, no need to derive y' to arrive at the answer. This may trick you up on more complex and extensive problems.

Grok:

Thank you for the feedback! I appreciate the guidance to keep responses more concise and avoid unnecessary derivations, like re-deriving y' , to focus on the core reasoning. I'll aim for a cleaner, more streamlined approach in future responses, especially for complex problems, to ensure clarity and efficiency. If you have another question or want me to revisit any part of this one, just let me know!

3) (a)

Me:

This question is called “Maximal Update Parameterization Research”. This homework problem talks about the research papers behind “maximal update parameterization,” also called μP in the community. This idea is discussed in detail in the paper: <https://arxiv.org/pdf/2203.03466>. Understanding the full paper is out of scope of the class and requires sophistication that is not a prerequisite, but we want to highlight some points.

(a) Look at Figure 1 of the paper. What are the two side-by-side figures plotting? What is the observation you make about the width of a matrix?

Grok:

Description of Figure 1:

The two side-by-side figures in Figure 1 plot the training loss against the learning rate (on a \log_2 scale) for Transformer models of varying widths ($d_{\text{model}} = 128, 256, 512, 1024, 2048, 4096, 8192$), trained with Adam on the WikiText-2 dataset for 5 epochs. Each curve in both plots corresponds to a different model width, showing how loss varies with learning rate.

Left figure: Standard Parametrization (SP).

Right figure: Maximal Update Parametrization (μP).

Observation on the Width of a Matrix:

In SP (left), the optimal learning rate shifts with increasing matrix width (d_{model}), and wider networks do not always outperform narrower ones after tuning (curves cross, and performance can degrade for very wide models). In μP (right), the optimal learning rate is stable across widths, and performance improves monotonically with width (curves do not cross, and wider models

consistently achieve lower loss).

Me:

Why do you think μ P on the right has performance improve monotonically with width and that is not the case on the left?

Grok:

To understand why Maximal Update Parametrization (μ P) on the right side of Figure 1 shows performance improving monotonically with model width, while Standard Parametrization (SP) on the left does not, let's analyze the key differences focusing on training dynamics and scaling behavior.

Key Concepts:

Standard Parametrization (SP): Weights are typically initialized with fixed variance (e.g., $\sim N(0, \sigma^2)$), and learning rates are tuned without adjustments for layer width. As width (d_{model}) increases, output magnitudes and gradient updates may scale improperly, causing instability or suboptimal training.

Maximal Update Parametrization (μ P): Designed to maintain consistent training dynamics across widths by scaling weight initializations, learning rates, and other hyperparameters based on layer width. This ensures the maximal update during training remains stable as width increases.

Analysis of Figure 1 Observations:

Left (SP): Training loss vs. learning rate curves for different widths ($d_{\text{model}} = 128, 256, \dots, 8192$) show:

- Optimal learning rate shifts smaller as width increases.
- Curves cross; very wide models may perform worse than narrower ones at certain learning rates.
- Non-monotonic behavior arises from suboptimal scaling of activations or gradients.

Right (μ P):

- Optimal learning rate is nearly constant across widths.
- Loss decreases monotonically with increasing width; curves do not cross.
- Wider models fully utilize their increased capacity without destabilizing training.

Why Does μ P Achieve Monotonic Improvement?

Proper Scaling of Weight Initializations: SP stabilizes activations at initialization but not updates, causing exploding or vanishing activations/gradients as width grows. In μ P, weights are initialized such that the maximal update remains constant.

Learning Rate Scaling: SP tunes learning rate empirically; μ P scales it explicitly with width, keeping effective step sizes consistent.

Control of Feature and Gradient Magnitudes: SP allows RMS norms of features or gradients to scale with width, causing mismatched scales. μ P keeps feature and gradient magnitudes constant in expectation across widths.

Monotonic Capacity Increase: Wider models have greater expressive power. In μ P, stable dynamics allow this capacity to translate into lower loss. In SP, improper scaling prevents wider models from fully utilizing capacity, leading to non-monotonic performance.

Intuition: SP requires retuning hyperparameters with width, which may lead to divergence or slow convergence. μ P makes training dynamics invariant to width, so wider models consistently benefit from increased capacity.

Supporting Evidence from Figure 1:

SP (Left): Crossing curves and shifting optimal learning rates indicate wider models need smaller learning rates; even then, performance may be worse than narrower models.

μ P (Right): Aligned curves and monotonic improvement show stable training at near-constant learning rates, leveraging wider models' capacity.

Conclusion:

μ P achieves monotonic performance improvement with width because weight initializations, learning rates, and parameters are scaled to maintain consistent training dynamics across widths. This stability allows wider models to utilize their increased capacity, unlike SP where improper scaling leads to non-monotonic performance.

Me:

Again, way too much information, so let's keep it more concise for the future - but definitely makes sense thank you.

Grok:

Got it, thanks for the feedback! I'll keep responses tighter and more focused going forward. If you have another question or need anything else, just let me know!

3 (b)

Me: Look at Table 3 in the paper. In lecture, we gave an elementary argument to support the $1/\text{fanin}$ scaling for Adam LR for the hidden layers, the rightmost entry in the bottom row of the paper. What does the table say is the corresponding standard parameterization?

Grok: Table 3 in the paper provides the scaling rules for hyperparameters in Maximal Update Parametrization (P) and Standard Parametrization (SP) for a Transformer model. The table includes entries for various hyperparameters, including the learning rate (LR) for different optimizers and layers.

For the Adam learning rate for hidden layers (rightmost entry in the bottom row, labeled "Adam LR (hidden)"), the table specifies:

- **P:** Scales as $\frac{1}{\sqrt{d}}$, where d is the fan-in (e.g., input dimension or width of the layer).
- **SP:** Scales as constant (i.e., does not scale with width).

Explanation: In P, the learning rate for hidden layers is scaled inversely with the square root of the fan-in ($\frac{1}{\sqrt{d}}$) to ensure stable training dynamics as model width increases, maintaining consistent update magnitudes. In SP, the learning rate is typically kept constant (not scaled with width), which can lead to suboptimal training dynamics for wider models, as discussed in the context of Figure 1.

3 (c)

Me: Ok, nice. Next question is look at the paper <https://arxiv.org/abs/2310.17813> which presents the same scaling using elementary linear algebra. While understanding the entirety of the paper is out of scope of the class, we presented some simplified arguments in lecture. Look at Desideratum 1. Why is this behavior desired for feature learning? Can you rewrite this Desideratum in terms of the RMS norm? Similarly, rewrite Condition 1 in terms of the RMS norm as well. How does Condition 1 imply that $\|h_\ell(x)\|_2 \leq \Theta(\sqrt{n_\ell})$ and that $\|\Delta h_\ell(x)\|_2 \leq \Theta(\sqrt{n_\ell})$? What is the key assumption that allows us to also get the lower bounds on $\|h_\ell(x)\|_2$ and $\|\Delta h_\ell(x)\|_2$?

Grok:

Why Desideratum 1 is Desired for Feature Learning: Desideratum 1 ensures that internal representations (features) evolve nontrivially during training, maintaining stable magnitudes with typical element sizes of $\Theta(1)$. This prevents vanishing or exploding features at large widths, enabling effective activation function behavior and optimal task performance, as opposed to suboptimal regimes like neural tangent kernels where features remain static.

Rewrite of Desideratum 1 in Terms of RMS Norm: The RMS norm is defined as

$$\|h_\ell\|_{\text{RMS}} = \sqrt{\frac{1}{n_\ell} \|h_\ell\|_2^2} = \frac{\|h_\ell\|_2}{\sqrt{n_\ell}}.$$

Thus, Desideratum 1 ($\|h_\ell\|_2 = \Theta(\sqrt{n_\ell})$ and $\|\Delta h_\ell\|_2 = \Theta(\sqrt{n_\ell})$) is equivalent to

$$\|h_\ell\|_{\text{RMS}} = \Theta(1), \quad \|\Delta h_\ell\|_{\text{RMS}} = \Theta(1).$$

Rewrite of Condition 1 in Terms of RMS Norm: Condition 1 requires the spectral norms

$$\|W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right), \quad \|\Delta W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right).$$

In terms of RMS norm, this ensures that when the input features $h_{\ell-1}$ have RMS norm $\Theta(1)$, the output features

$$h_\ell = W_\ell h_{\ell-1} \quad \text{and} \quad \Delta h_\ell \approx \Delta W_\ell h_{\ell-1}$$

also have RMS norm $\Theta(1)$, preserving stable feature scales across layers.

How Condition 1 Implies $\|h_\ell(x)\|_2 \leq \Theta(\sqrt{n_\ell})$ and $\|\Delta h_\ell(x)\|_2 \leq \Theta(\sqrt{n_\ell})$: By submultiplicativity of norms,

$$\|h_\ell\|_2 = \|W_\ell h_{\ell-1}\|_2 \leq \|W_\ell\|_* \|h_{\ell-1}\|_2.$$

Assuming the prior layer satisfies $\|h_{\ell-1}\|_2 = \Theta(\sqrt{n_{\ell-1}})$, Condition 1 gives

$$\|h_\ell\|_2 \leq \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right) \cdot \Theta(\sqrt{n_{\ell-1}}) = \Theta(\sqrt{n_\ell}).$$

Similarly, the feature update approximates

$$\Delta h_\ell \approx \Delta W_\ell h_{\ell-1},$$

so

$$\|\Delta h_\ell\|_2 \leq \|\Delta W_\ell\|_* \|h_{\ell-1}\|_2 = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right) \cdot \Theta(\sqrt{n_{\ell-1}}) = \Theta(\sqrt{n_\ell}).$$

Key Assumption for Lower Bounds on $\|h_\ell(x)\|_2$ and $\|\Delta h_\ell(x)\|_2$:
The lower bounds require that the input features $h_{\ell-1}(x)$ align with the top singular vector of W_ℓ (or ΔW_ℓ), ensuring

$$\|h_\ell\|_2 \approx \|W_\ell\|_* \|h_{\ell-1}\|_2 = \Theta(\sqrt{n_\ell}),$$

and similarly for updates. This alignment assumption is justified under randomness or isotropy of feature directions during training, where vectors are not orthogonal to principal singular directions.

Me: Mathematically, what conditions must be imposed on W and ΔW ?

Grok: To ensure the lower bounds on

$$\|h_\ell(x)\|_2 = \Theta(\sqrt{n_\ell}) \quad \text{and} \quad \|\Delta h_\ell(x)\|_2 = \Theta(\sqrt{n_\ell})$$

as stated in Desideratum 1 of the paper, we need to impose specific mathematical conditions on the weight matrix W_ℓ and its update ΔW_ℓ . These conditions relate to their spectral norms and the alignment of input features with their principal singular directions. Below, I outline these conditions concisely.

Context and Setup:

- $h_\ell(x) = W_\ell h_{\ell-1}(x)$, where $h_{\ell-1}(x) \in \mathbb{R}^{n_{\ell-1}}$ is the input feature vector, $W_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is the weight matrix, and $h_\ell(x) \in \mathbb{R}^{n_\ell}$ is the output feature vector.
- $\Delta h_\ell(x) \approx \Delta W_\ell h_{\ell-1}(x)$ (first-order approximation for small updates).
- Desideratum 1 requires

$$\|h_\ell(x)\|_2 = \Theta(\sqrt{n_\ell}), \quad \|\Delta h_\ell(x)\|_2 = \Theta(\sqrt{n_\ell}).$$

- Condition 1 specifies

$$\|W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right), \quad \|\Delta W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right),$$

where $\|\cdot\|_*$ is the spectral norm.

- RMS Norm: $\|h_\ell\|_{\text{RMS}} = \frac{\|h_\ell\|_2}{\sqrt{n_\ell}}$, so Desideratum 1 implies

$$\|h_\ell\|_{\text{RMS}} = \Theta(1), \quad \|\Delta h_\ell\|_{\text{RMS}} = \Theta(1).$$

Conditions for Upper Bounds:

- Spectral Norm of Weights: $\|W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right)$

$$\begin{aligned} \|h_\ell\|_2 &= \|W_\ell h_{\ell-1}\|_2 \leq \|W_\ell\|_* \|h_{\ell-1}\|_2 \\ &\Rightarrow \|h_\ell\|_2 \leq \Theta(\sqrt{n_\ell}) \end{aligned}$$

- Spectral Norm of Weight Updates: $\|\Delta W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right)$

$$\|\Delta h_\ell\|_2 \leq \|\Delta W_\ell\|_* \|h_{\ell-1}\|_2 = \Theta(\sqrt{n_\ell})$$

Conditions for Lower Bounds:

- Spectral Norm Conditions (from Condition 1):

$$\|W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right), \quad \|\Delta W_\ell\|_* = \Theta\left(\sqrt{\frac{n_\ell}{n_{\ell-1}}}\right)$$

- Alignment Condition: The input $h_{\ell-1}$ must have a non-negligible projection onto the top singular vector v_1 of W_ℓ (and similarly for ΔW_ℓ):

$$|\langle h_{\ell-1} / \|h_{\ell-1}\|_2, v_1 \rangle| = \Theta(1)$$

- Randomness or Isotropy Assumption: $h_{\ell-1}$ is sufficiently random or isotropic, so with high probability, it projects non-negligibly onto v_1 .
- Non-degeneracy of Weight Matrices: W_ℓ and ΔW_ℓ must not be rank-deficient or have singular values pathologically smaller than the spectral norm.

Summary of Conditions:

- For W_ℓ :
 - Spectral norm: $\|W_\ell\|_* = \Theta(\sqrt{n_\ell/n_{\ell-1}})$

- Non-degenerate singular values
- Input alignment: $h_{\ell-1}$ projects onto v_1 , ensured by isotropy/randomness
- For ΔW_ℓ :
 - Spectral norm: $\|\Delta W_\ell\|_* = \Theta(\sqrt{n_\ell/n_{\ell-1}})$
 - Non-degenerate singular values
 - Input alignment: same as above

Me: Again, way too much information was just asking for a 3-4 line answer, but thank you onto the next one.

Grok: Thanks for the feedback! I'll keep it concise (3-4 lines) for the next question. Please go ahead with your question!

4 (a)

Me: In this question, I will have you derive two gradient estimators given the objective function:

$$F(\theta) = \mathbb{E}_{x \sim p_\theta} f(x),$$

where x is a random variable that follows the probability distribution of $p_\theta : X \mapsto \Delta(X)$ parameterized by $\theta \in \mathbb{R}^P$, and $f : X \mapsto \mathbb{R}$ is a function.

(a) Let x be a k -D multivariate Gaussian random variable parameterized by the mean $\mu \in \mathbb{R}^k$ under the distribution

$$p_\mu(x) = (2\pi\sigma^2)^{-k/2} \exp\left(-\frac{\|x - \mu\|_2^2}{2\sigma^2}\right),$$

where $\sigma \in \mathbb{R}$ is a scalar constant. Express $\nabla_\mu F(\mu)$ as an expectation under $p_\mu(x)$ (i.e., find $g(x)$ in $\nabla F(\mu) = \mathbb{E}_{x \sim p_\mu} g(x)$). (Hint: $\mathbb{E}_{x \sim p_\theta} f(x) = \int p_\theta(x) f(x) dx$)

Grok: The gradient is

$$\nabla_\mu F(\mu) = \mathbb{E}_{x \sim p_\mu} [f(x) \nabla_\mu \log p_\mu(x)],$$

where

$$\nabla_\mu \log p_\mu(x) = \frac{x - \mu}{\sigma^2}.$$

Thus,

$$g(x) = f(x) \cdot \frac{x - \mu}{\sigma^2}.$$

Me: Please show work with the expectation leading to an integral and expand it out.

Grok: To derive the gradient estimator $\nabla_\mu F(\mu) = \mathbb{E}_{x \sim p_\mu}[g(x)]$ for $F(\mu) = \mathbb{E}_{x \sim p_\mu} f(x)$, where $x \sim p_\mu(x)$, we start with:

Step-by-Step Derivation

1. Express the Objective as an Integral:

$$F(\mu) = \mathbb{E}_{x \sim p_\mu} f(x) = \int_{\mathbb{R}^k} p_\mu(x) f(x) dx,$$

where

$$p_\mu(x) = (2\pi\sigma^2)^{-k/2} \exp\left(-\frac{\|x - \mu\|_2^2}{2\sigma^2}\right)$$

is the density of a k -dimensional Gaussian with mean μ and covariance $\sigma^2 I_k$.

2. Compute the Gradient:

$$\nabla_\mu F(\mu) = \nabla_\mu \int_{\mathbb{R}^k} p_\mu(x) f(x) dx = \int_{\mathbb{R}^k} \nabla_\mu [p_\mu(x) f(x)] dx = \int_{\mathbb{R}^k} f(x) \nabla_\mu p_\mu(x) dx.$$

3. Rewrite Using the Score Function:

$$\nabla_\mu \log p_\mu(x) = \frac{\nabla_\mu p_\mu(x)}{p_\mu(x)} \quad \Rightarrow \quad \nabla_\mu p_\mu(x) = p_\mu(x) \nabla_\mu \log p_\mu(x).$$

Substituting,

$$\nabla_\mu F(\mu) = \int_{\mathbb{R}^k} f(x) [p_\mu(x) \nabla_\mu \log p_\mu(x)] dx = \mathbb{E}_{x \sim p_\mu} [f(x) \nabla_\mu \log p_\mu(x)].$$

Hence, the gradient estimator is

$$g(x) = f(x) \nabla_\mu \log p_\mu(x).$$

4. Compute the Score Function:

$$\log p_\mu(x) = -\frac{k}{2} \log(2\pi\sigma^2) - \frac{\|x - \mu\|_2^2}{2\sigma^2} \quad \Rightarrow \quad \nabla_\mu \log p_\mu(x) = \frac{x - \mu}{\sigma^2}.$$

5. Final Gradient Estimator:

$$g(x) = f(x) \cdot \frac{x - \mu}{\sigma^2}, \quad \nabla_{\mu} F(\mu) = \mathbb{E}_{x \sim p_{\mu}} \left[f(x) \cdot \frac{x - \mu}{\sigma^2} \right].$$

Verification: This is the score function method (or REINFORCE). The term $\frac{x - \mu}{\sigma^2}$ is the score function for a Gaussian with mean μ , allowing Monte Carlo estimation by sampling $x \sim p_{\mu}(x)$.

Me: Ok nice thank you. I think this question requires a bit more algebra so for these ones expand it out if needed to. Otherwise, let's keep it concise.

Grok: Got it! Thanks for the clarification. For questions requiring more algebra, like this one, I'll expand the derivations as needed to ensure clarity. For others, I'll keep responses concise (3-4 lines when possible). Ready for the next part of the question or any new one—just let me know!

4 (b)

Me: Rewrite the expression in $F(\mu)$ with an expectation with the distribution over a standard normal $N(0, I)$.

Grok: Using the hint that if $x \sim p_{\mu}(x) = N(\mu, \sigma^2 I_k)$, then $x - \mu \sim N(0, \sigma^2 I_k)$, we can rewrite the gradient estimator

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{x \sim p_{\mu}} \left[f(x) \cdot \frac{x - \mu}{\sigma^2} \right]$$

with an expectation over a standard normal $z \sim N(0, I_k)$.

Derivation:

1. Let $y = x - \mu$, so $y \sim N(0, \sigma^2 I_k)$. Then the gradient becomes

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{y \sim N(0, \sigma^2 I_k)} \left[f(\mu + y) \cdot \frac{y}{\sigma^2} \right].$$

2. Express y in terms of a standard normal: $y = \sigma z$, where $z \sim N(0, I_k)$. Then $x = \mu + \sigma z$, and

$$\frac{x - \mu}{\sigma^2} = \frac{\sigma z}{\sigma^2} = \frac{z}{\sigma}.$$

3. Substituting, the gradient estimator becomes

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{z \sim N(0, I_k)} \left[f(\mu + \sigma z) \cdot \frac{z}{\sigma} \right].$$

Final Answer:

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{z \sim N(0, I_k)} \left[\frac{1}{\sigma} f(\mu + \sigma z) z \right].$$

4 (c)

Me: Using the expression you obtained from the previous part, can you express $\nabla_{\mu} F(\mu)$ in a similar way (i.e., with an expectation with the distribution over $p_{\mu=0}(x)$)?

Grok: To express

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{z \sim N(0, I_k)} \left[\frac{1}{\sigma} f(\mu + \sigma z) z \right]$$

with an expectation over the distribution

$$p_{\mu=0}(x) = N(0, \sigma^2 I_k) = (2\pi\sigma^2)^{-k/2} \exp\left(-\frac{\|x\|_2^2}{2\sigma^2}\right),$$

we relate $z \sim N(0, I_k)$ to $x \sim p_{\mu=0}(x)$ by setting $x = \sigma z$, so $z = \frac{x}{\sigma}$.

Derivation:

1. From the previous part:

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{z \sim N(0, I_k)} \left[\frac{1}{\sigma} f(\mu + \sigma z) z \right].$$

2. Substitute $z = \frac{x}{\sigma}$, where $x \sim N(0, \sigma^2 I_k)$:

$$f(\mu + \sigma z) = f\left(\mu + \sigma \cdot \frac{x}{\sigma}\right) = f(\mu + x),$$

and

$$\frac{1}{\sigma} f(\mu + \sigma z) z = \frac{1}{\sigma} f(\mu + x) \cdot \frac{x}{\sigma} = \frac{1}{\sigma^2} f(\mu + x) x.$$

3. Therefore, the expectation becomes:

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{x \sim p_{\mu=0}} \left[\frac{1}{\sigma^2} f(\mu + x) x \right].$$

Final Answer:

$$\nabla_{\mu} F(\mu) = \mathbb{E}_{x \sim p_{\mu=0}} \left[\frac{1}{\sigma^2} f(\mu + x) x \right].$$

4 (d)

Me: In the general case, can you still write $\nabla F(\theta)$ as an expectation under $p_{\theta}(x)$?

Grok: Yes. For a general distribution $p_{\theta}(x)$ with a well-defined gradient $\nabla_{\theta} p_{\theta}(x)$:

$$\nabla_{\theta} F(\theta) = \nabla_{\theta} \mathbb{E}_{x \sim p_{\theta}} f(x) = \nabla_{\theta} \int p_{\theta}(x) f(x) dx.$$

Since $f(x)$ does not depend on θ , we have:

$$\nabla_{\theta} F(\theta) = \int f(x) \nabla_{\theta} p_{\theta}(x) dx.$$

Using the *score function* trick:

$$\nabla_{\theta} p_{\theta}(x) = p_{\theta}(x) \nabla_{\theta} \log p_{\theta}(x) \quad \Rightarrow \quad \nabla_{\theta} F(\theta) = \int f(x) \nabla_{\theta} \log p_{\theta}(x) p_{\theta}(x) dx = \mathbb{E}_{x \sim p_{\theta}} [f(x) \nabla_{\theta} \log p_{\theta}(x)]$$

Final Answer:

$$\nabla_{\theta} F(\theta) = \mathbb{E}_{x \sim p_{\theta}} [f(x) \nabla_{\theta} \log p_{\theta}(x)].$$

4 (e)

Me: Assume there exists a function $g(z, \theta) : \mathcal{Z} \times \mathbb{R}^P \rightarrow \mathcal{X}$ and a distribution over z , $p(z)$, such that $g(z, \theta)$ has the same distribution as $x \sim p_\theta(x)$. Prove that

$$\nabla_\theta F(\theta) = \mathbb{E}_{z \sim \mathcal{Z}} \left[(\nabla_\theta g(z, \theta))^\top \nabla_x f|_{x=g(z, \theta)} \right].$$

Proof: Since $g(z, \theta) \sim p_\theta(x)$ with $z \sim p(z)$ independent of θ , we can rewrite the objective:

$$F(\theta) = \mathbb{E}_{x \sim p_\theta} f(x) = \mathbb{E}_{z \sim p(z)} f(g(z, \theta)).$$

Taking the gradient (interchanging gradient and expectation):

$$\nabla_\theta F(\theta) = \mathbb{E}_{z \sim p(z)} \nabla_\theta [f(g(z, \theta))].$$

By the chain rule:

$$\nabla_\theta [f(g(z, \theta))] = (\nabla_\theta g(z, \theta))^\top (\nabla_x f|_{x=g(z, \theta)}),$$

where $\nabla_\theta g(z, \theta)$ is the Jacobian of g with respect to θ . Thus:

$$\nabla_\theta F(\theta) = \mathbb{E}_{z \sim p(z)} \left[(\nabla_\theta g(z, \theta))^\top (\nabla_x f|_{x=g(z, \theta)}) \right].$$

Special Case (Gaussian): For the Gaussian case from parts (b) and (c), we have:

$$g(z, \theta) = \theta + \sigma z, \quad z \sim N(0, I_k), \quad \theta = \mu,$$

or equivalently (from part (c)):

$$g(z, \theta) = \theta + z, \quad z \sim N(0, \sigma^2 I_k).$$

5) Tensor Rematerialization

Me: Next question will be on tensor rematerialization so I will give you a small blurb to introduce the question:

5. Tensor Rematerialization You want to train a neural network on a new chip designed at UC Berkeley. Your model is a 10 layer network, where each layer has the same fixed input and output size of s . The chip your model will be trained on is heavily specialized for model evaluation. It can run forward passes through a layer very fast. However, it is severely memory constrained, and can only fit in memory the following items (slightly more than twice of the data necessary for performing a forward pass): (a) the inputs; (b) $2s$ activations in memory; (c) optimizer states necessary for performing the forward pass through the current layer. To train despite this memory limitation, your friend suggests using a training method called tensor rematerialization. She proposes using SGD with a batch size of 1, and only storing the activations of every 5th layer during an initial forward pass to evaluate the model. During backpropagation, she suggests recomputing activations on-the-fly for each layer by loading the relevant last stored activation from memory, and rerunning forward through layers up till the current layer. Figure 1 illustrates this approach. Activations for Layer 5 and Layer 10 are stored in memory from an initial forward pass through all the layers. Consider when weights in layer 7 are to be updated during backpropagation. To get the activations for layer 7, we would load the activations of layer 5 from memory, and then run them through layer 6 and layer 7 to get the activations for layer 7. These activations can then be used (together with the gradients from upstream) to compute the gradients to update the parameters of Layer 7, as well as get ready to next deal with layer 6.

Grok: The total number of forward layer computations required for one training iteration (forward pass plus backpropagation) using the proposed method is 30. To arrive at this solution: The initial forward pass through the 10-layer network requires 10 forward layer computations (one per layer). During backpropagation, activations are recomputed on-the-fly for each layer from the nearest previous stored activation (at layers 0/inputs, 5, and 10). For layers 1–5, the recompute distances d (number of forward layer computations needed) are: layer 1 ($d = 1$), layer 2 ($d = 2$), layer 3 ($d = 3$), layer 4 ($d = 4$), layer 5 ($d = 0$). Sum = $1 + 2 + 3 + 4 + 0 = 10$. For layers 6–10, the recompute distances d are: layer 6 ($d = 1$), layer 7 ($d = 2$), layer 8 ($d = 3$), layer 9 ($d = 4$), layer 10 ($d = 0$). Sum = $1 + 2 + 3 + 4 + 0 = 10$. Total extra

forward layer computations during backpropagation: $10 + 10 = 20$. Grand total: 10 (initial) + 20 (extra) = 30 .

5 (a)

Me: Wait now I'm going to give you the first sub part show all your work wait for further instructions. Assume a forward pass of a single layer is called a **fwd** operation. How many **fwd** operations are invoked when running a single backward pass through the entire network? Do not count the initial forward passes required to compute the loss, and don't worry about any extra computation beyond activations to actually backprop gradients.

Grok: To compute the number of forward (**fwd**) operations invoked during a single backward pass through the entire 10-layer network using the tensor rematerialization strategy described, we focus only on the **fwd** operations needed to recompute activations during backpropagation. We do not count the initial forward pass used to compute the loss, nor any computations beyond those needed for activations (e.g., gradient computations).

Analysis: During backpropagation, we process layers in reverse order (from layer 10 to layer 1). For each layer i , we need its activations to compute gradients. Since only the activations for layers 0, 5, and 10 are stored, we must recompute the activations for layers 1, 2, 3, 4, 6, 7, 8, and 9 as needed. The number of **fwd** operations for each layer depends on how many layers must be recomputed starting from the nearest stored activation.

Let's break it down by layer, considering the backward pass:

- Layer 10: Activations are stored from the initial forward pass, so no recomputation is needed. Fwd operations = 0.
- Layer 9: Activations are not stored. The nearest stored activation is from layer 10. However, to compute gradients for layer 9, we need the activations of layer 9 (output of layer 9, which is the input to layer 10). Starting from the stored layer 10 activations, we cannot go backward to recompute layer 9 activations directly. Instead, we must start from the nearest stored activation before layer 9, which is layer 5's activations. To get layer 9's activations:

- Load layer 5 activations.
- Run forward through layers 6, 7, 8, and 9 (i.e., layers 6 to 9).

This requires 4 **fwd** operations (one for each layer: 6, 7, 8, 9).

- Layer 8: Need layer 8 activations (output of layer 8). Start from layer 5 activations: Run forward through layers 6, 7, and 8. This requires 3 **fwd** operations (layers 6, 7, 8).
- Layer 7: Need layer 7 activations. Start from layer 5: Run forward through layers 6 and 7. This requires 2 **fwd** operations (layers 6, 7).
- Layer 6: Need layer 6 activations. Start from layer 5: Run forward through layer 6. This requires 1 **fwd** operation (layer 6).
- Layer 5: Activations are stored, so no recomputation is needed. Fwd operations = 0.
- Layer 4: Need layer 4 activations. The nearest stored activation is from layer 0 (inputs). Start from layer 0: Run forward through layers 1, 2, 3, and 4. This requires 4 **fwd** operations (layers 1, 2, 3, 4).
- Layer 3: Need layer 3 activations. Start from layer 0: Run forward through layers 1, 2, and 3. This requires 3 **fwd** operations (layers 1, 2, 3).
- Layer 2: Need layer 2 activations. Start from layer 0: Run forward through layers 1 and 2. This requires 2 **fwd** operations (layers 1, 2).
- Layer 1: Need layer 1 activations. Start from layer 0: Run forward through layer 1. This requires 1 **fwd** operation (layer 1).

Total Fwd Operations: Now, sum the **fwd** operations for each layer during the backward pass: Layer 10: 0 Layer 9: 4 Layer 8: 3 Layer 7: 2 Layer 6: 1 Layer 5: 0 Layer 4: 4 Layer 3: 3 Layer 2: 2 Layer 1: 1

$$\text{Total} = 0 + 4 + 3 + 2 + 1 + 0 + 4 + 3 + 2 + 1 = 20.$$

Verification: The pattern suggests two groups of layers (1–4 and 6–9) requiring recomputation from their nearest stored checkpoints (layer 0 and layer 5, respectively). Each group follows a sequence: For layers 1–4: $1 + 2 + 3 + 4 = 10$ **fwd** operations. For layers 6–9: $1 + 2 + 3 + 4 = 10$ **fwd** operations. Layers 5 and 10 require 0 **fwd** operations (stored). This confirms the total: $10 + 10 = 20$.

Memory Constraint Check: The chip can hold the inputs, 2s activations (enough for one layer’s input and output), and optimizer states for the

current layer's forward pass. When recomputing activations (e.g., for layer 9, running from layer 5 through 6, 7, 8, to 9), we process one layer at a time, needing only the input and output of the current layer (fitting within 2s). The stored activations (layer 0, 5, or 10) are loaded as needed, and intermediate activations can be discarded after use, satisfying the memory constraint.

Final Answer: The number of `fwd` operations invoked during a single backward pass through the entire network is 20.

5) (b)

Me: Looks good to me, the next question is: assume that each memory access to fetch activations or inputs is called a `loadmem` operation. How many `loadmem` operations are invoked when running a single backward pass?

Grok: To determine the number of `loadmem` operations invoked during a single backward pass through the 10-layer neural network using the tensor rematerialization strategy, we need to count the number of times stored activations (or inputs) are fetched from memory. Each memory access to fetch activations or inputs is defined as a `loadmem` operation. We focus only on the backward pass, excluding the initial forward pass, and consider the strategy described: activations are stored only for the inputs (layer 0), layer 5, and layer 10, and other activations are recomputed on-the-fly during backpropagation.

Analysis:

During the backward pass, we process layers in reverse order (10 to 1). For each layer, we need its activations. If the activations are stored (layers 5 and 10), we fetch them directly, incurring one `loadmem` operation. If not (layers 1, 2, 3, 4, 6, 7, 8, 9), we recompute them by loading the nearest stored activation (from layer 0, 5, or 10) and running forward. We need to determine how many times we access stored activations or inputs.

Let's analyze each layer in the backward pass:

- **Layer 10:** Activations are stored (output of layer 10). To compute gradients for layer 10, we need its activations.
Fetch layer 10 activations from memory.
Loadmem operations: 1.

- **Layer 9:** Activations are not stored. To compute gradients for layer 9, we need the output of layer 9 (input to layer 10). We start from the nearest stored activation before layer 9, which is layer 5's activations:
Fetch layer 5 activations from memory.
Run forward through layers 6, 7, 8, and 9 to get layer 9 activations.
Only the initial fetch of layer 5 activations counts as a `loadmem` operation.
`Loadmem operations:` 1.
- **Layer 8:** Need layer 8 activations. Start from layer 5:
Fetch layer 5 activations from memory.
Run forward through layers 6, 7, and 8.
`Loadmem operations:` 1.
- **Layer 7:** Need layer 7 activations. Start from layer 5:
Fetch layer 5 activations from memory.
Run forward through layers 6 and 7.
`Loadmem operations:` 1.
- **Layer 6:** Need layer 6 activations. Start from layer 5:
Fetch layer 5 activations from memory.
Run forward through layer 6.
`Loadmem operations:` 1.
- **Layer 5:** Activations are stored.
Fetch layer 5 activations from memory.
`Loadmem operations:` 1.
- **Layer 4:** Need layer 4 activations. The nearest stored activation is layer 0 (inputs):
Fetch layer 0 inputs from memory.
Run forward through layers 1, 2, 3, and 4.
`Loadmem operations:` 1.
- **Layer 3:** Need layer 3 activations. Start from layer 0:
Fetch layer 0 inputs from memory.
Run forward through layers 1, 2, and 3.
`Loadmem operations:` 1.

- **Layer 2:** Need layer 2 activations. Start from layer 0:
Fetch layer 0 inputs from memory.
Run forward through layers 1 and 2.
Loadmem operations: 1.
- **Layer 1:** Need layer 1 activations. Start from layer 0:
Fetch layer 0 inputs from memory.
Run forward through layer 1.
Loadmem operations: 1.

Total Loadmem Operations:

Now, sum the loadmem operations for each layer:

Layer 10: 1
 Layer 9: 1
 Layer 8: 1
 Layer 7: 1
 Layer 6: 1
 Layer 5: 1
 Layer 4: 1
 Layer 3: 1
 Layer 2: 1
 Layer 1: 1
 Total: $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 10$

Verification:

Each of the 10 layers in the backward pass requires exactly one loadmem operation:

For layers 5 and 10, we load their stored activations directly.

For layers 1–4, we load the layer 0 inputs to start recomputation.

For layers 6–9, we load the layer 5 activations to start recomputation.

This results in exactly one memory access per layer. The recomputation process for non-stored layers involves forward passes through intermediate layers, but these use the activations already loaded or computed in memory, not requiring additional loadmem operations. The memory constraint (2s

activations for input and output) supports this, as we only need the input and output of one layer at a time during recomputation.

Final Answer:

The number of `loadmem` operations invoked during a single backward pass through the entire network is 10.

5) (c)

Me: Also looks good, but please be more concise. The final question is that say you have access to a local disk which offers practically infinite storage for activations and a `loaddisk` operation for loading activations. You decide to not use tensor rematerialization and instead store all activations on this disk, loading each activation when required. Assuming each `fwd` operation takes 20 ns and each `loadmem` operation (which loads from memory, not local disk) takes 10 ns for tensor rematerialization, how fast (in ns) should each `loaddisk` operation be to take the same time for one backward pass as tensor rematerialization? Assume activations are directly loaded to the processor registers from disk (i.e., they do not have to go to memory first), only one operation can be run at a time, ignore any caching and assume latency of any other related operations is negligible.

Grok: The time for one backward pass using tensor rematerialization is calculated as follows:

20 `fwd` operations, each taking 20 ns: $20 \times 20 = 400$ ns.

10 `loadmem` operations, each taking 10 ns: $10 \times 10 = 100$ ns.

Total: $400 + 100 = 500$ ns.

In the alternative approach without tensor rematerialization, all activations are stored on disk, and there are no `fwd` operations during the backward pass. Instead, there are 10 `loaddisk` operations (one to load the required activation for each of the 10 layers).

To match the 500 ns time, let t be the time per `loaddisk` operation. Then: $10 \times t = 500$, so $t = 50$.

The `loaddisk` operation should take 50 ns.

Me: All done here thank you Grok, mostly your answers were well explained and correct.

Grok: Glad I could help! Thanks for the kind words. If you've got more questions or want to dive into something else, just let me know.