Question 1:

This is a copy of a conversation between ChatGPT & Anonymous.

Report conversation

Consider a dense layer with input x ∈ Rd1 , weights W ∈ Rd2×d1 , 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 = W x? How does this scale
with d1 or d2? What constant should we multiply W by to ensure that the expected squared
RMS norm of W x is 1, regardless of d1 and d2?

`Thought for 6s`

Short answer:

$\mathbb{E}\big[\texttt{(squared RMS of } y\texttt{)}\big] = d_1$. It scales linearly with d_1 and does **not** depend on d_2. To make the expected squared RMS equal to 1 multiply W by $1/\sqrt{d_1}$.

Derivation (clean):

RMS of y is $\sqrt{\frac{1}{d_2}\sum_{i=1}^{d_2} y_i^2}$. The squared RMS is

$$\frac{1}{d_2}\|y\|^2 = \frac{1}{d_2}\sum_{i=1}^{d_2} y_i^2 .$$

With $y_i = \sum_{j=1}^{d_1} W_{ij}x_j$ and $W_{ij}, x_j \overset{\text{iid}}{\sim} N(0,1)$,

$$\mathbb{E}[y_i^2] = \mathbb{E}\left[\sum_{j,k} W_{ij}W_{ik}x_jx_k\right] = \sum_{j,k}\mathbb{E}[W_{ij}W_{ik}]\,\mathbb{E}[x_jx_k] = \sum_{j=1}^{d_1} 1\cdot 1 = d_1,$$

since cross terms vanish. Hence

$$\mathbb{E}\left[\frac{1}{d_2}\|y\|^2\right] = \frac{1}{d_2}\sum_{i=1}^{d_2}\mathbb{E}[y_i^2] = \frac{d_2\cdot d_1}{d_2} = d_1 .$$

If W instead has variance σ_W^2 per e [illegible] $= d_1\sigma_W^2$. So to force $\mathbb{E}$[squared RMS] $= 1$ choose σ_W [illegible] scale W by $1/\sqrt{d_1}$.

(As a side note: Xavier/Glorot or He initializations use other scalings that trade off fan-in and fan-out for signal/gradient flow; the choice above is the simple fan-in normalization.)

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 xi ∈ Rd1 are sampled from an i.i.d. unit Gaussian, and the raw gradient ∇W f (W) is an outer product of the input and a constant backpropagated vector gi ∈ Rd2 .

Wt+1 ← Wt + ηsign(gixiT
).
What is the expected RMS norm squared of the (unscaled) change in features y′ = sign(gixiT)xi?
How does this scale with d1 or d2? What constant should we multiply the update by to ensure that the expected RMS norm of y′ is 1, regardless of d1 and d2?

Thought for 13s

Answer (short)

$$\mathbb{E}\Big[(\text{squared RMS of } y')\Big] = d_1\left(1-\tfrac{2}{\pi}\right) + d_1^2\left(\tfrac{2}{\pi}\right).$$

It does **not** depend on d_2. For large d_1 the leading term grows like $\frac{2}{\pi}d_1^2$ (quadratic in d_1). To make the expected RMS of y' equal to 1 multiply the (unscaled) update by

Skip to content

$$c = \frac{1}{\sqrt{d_1\left(1-\frac{2}{\pi}\right)+d_1^2\left(\frac{2}{\pi}\right)}}.$$

Asymptotically for large d_1, $c \approx \sqrt{\frac{\pi}{2}}\,\frac{1}{d_1}$.

Derivation

The raw elementwise update matrix for one sample is

$$U = \operatorname{sign}(g\,x^\top),$$

and since $\operatorname{sign}(ab) = \operatorname{sign}(a)\,\operatorname{sign}(b)$ we can write

$$U = s_g\, s_x^\top$$

where $s_g = \operatorname{sign}(g) \in \{\pm 1\}^{d_2}$ (constant here) and $s_x = \operatorname{sign}(x) \in \{\pm 1\}^{d_1}$. The feature-change vector is

$$y' = Ux = s_g\,(s_x^\top x).$$

Since s_g has d_2 entries of magnitude 1,

$$\|y'\|^2 = \|s_g\|^2 (s_x^\top x)^2 = d_2\,(s_x^\top x)^2,$$

so the squared RMS across the d_2 outputs is

$$\frac{1}{d_2}\|y'\|^2 = (s_x^\top x)^2.$$

But $s_x^\top x = \sum_{j=1}^{d_1} \operatorname{sign}(x_j)\,x_j = \sum_{j=1}^{d_1} |x_j|$. Thus

$$\mathbb{E}\big[\text{(squared RMS)}\big] = \mathbb{E}\Big[\Big(\sum_{j=1}^{d_1} |x_j|\Big)^2\Big].$$

With independent standard normal components,

$$E|X| = \sqrt{\frac{2}{\pi}}, \qquad \operatorname{Var}(|X|) = 1 - \frac{2}{\pi}.$$

Therefore

$$\mathbb{E}\Big[\Big(\sum_{j=1}^{d_1} |x_j|\Big)^2\Big] = d_1\operatorname{Var}(|X|) + d_1^2 (E|X|)^2 = d_1\Big(1-\frac{2}{\pi}\Big) + d_1^2\Big(\frac{2}{\pi}\Big).$$

Set c so that c^2 times this expectatic Skip to content

$$c = \frac{1}{\sqrt{d_1(1 - \frac{2}{\pi}) + d_1^2(\frac{2}{\pi})}}.$$

While this answer is technically correct, deriving the exact expected RMS norm is unnecessary to understand that the norm scales (d_1)^2. The remarks clarify this, but it may have been more intuitive to have explained the single case as the official solutions did

Remarks

- The update magnitude grows like d_1^2 (dominant) for wide layers, so to keep feature changes width-independent you need a scaling $\Theta(1/d_1)$ on the sign-update (contrast this with the $\Theta(1/\sqrt{d_1})$ fan-in scaling for Gaussian-initialized forward weights).
- The result is independent of d_2 because the per-output RMS divides out the factor d_2 .
- If the backprop vector g has variable magnitudes (not just signs) add a factor $\|g\|^2/d_2$ into the calculation (it multiplies the RHS).

> You may notice that the above update rule only depends on d1. Why is this the case?

Good observation — yes, the expected RMS change in features y' depends **only on** d_1 (the input dimension, or *fan-in*), not on d_2 (the output dimension, or *fan-out*).

Let's unpack *why* that's true.

---

1. The update rule is row-separable in W

The weight update is

$$W_{t+1} = W_t + \eta\,\mathrm{sign}(gx^\top),$$

and since $\mathrm{sign}(gx^\top) = \mathrm{sign}(g)\,\mathrm{sign}(x)^\top$,
each row of W is updated as

$$W_{t+1}^{(i,:)} = W_t^{(i,:)} + \eta\,\mathrm{sign}(g_i)\,\mathrm{sign}(x)^\top.$$

Skip to content

So each output neuron i gets the **sa** **pdate in input space**, just possibly flipped by the sign of g_i.

No row interacts with the others — the update structure is *independent across outputs*.

---

2. The feature change $y^{'} = (\mathrm{sign}(gx^\top))x$

1 and 2 are slightly repetitive in saying that the magnitude of the gradient will be the same except flipped by the sign of g_i.

We can write:

$$y^{'} = \mathrm{sign}(g)\,(\mathrm{sign}(x)^\top x).$$

Here:

- $\mathrm{sign}(g) \in \{\pm 1\}^{d_2}$ just flips signs per output neuron.
- The scalar $(\mathrm{sign}(x)^\top x) = \sum_{j=1}^{d_1} |x_j|$ is shared across all outputs.

Hence, **all d_2 outputs get the same magnitude change**, differing only by sign.

---

3. RMS normalization cancels d_2

The squared RMS of the feature change is

$$\mathrm{RMS}^2(y^{'}) = \frac{1}{d_2}\|y^{'}\|^2 = \frac{1}{d_2}\,d_2\,(\mathrm{sign}(x)^\top x)^2 = (\mathrm{sign}(x)^\top x)^2.$$

The factor of d_2 from $\|y^{'}\|^2$ cancels the $1/d_2$ in the RMS definition.

Thus, the expected RMS depends **only** on the statistics of $(\mathrm{sign}(x)^\top x)$, i.e. on the *input dimension* d_1.

---

4. Intuition

- The update magnitude per output neuron is the same, so increasing d_2 just replicates the same change across more outputs — the RMS normalization removes this duplication.
- In contrast, increasing d_1 increases the number of input components contributing to $\mathrm{sign}(x)^\top x$, so the change scales with how many input features are combined — that's why it grows with d_1.

Summary

The dependence on d_1 but not d_2 arises because:

- d_1 controls how many input coordinates contribute to each neuron’s update (the *fan-in effect*),
- while d_2 only multiplies identical copies of the same output update, which cancels out in the RMS average.

In short:

chan

Message ChatGPT
Attach Search Study Voice

Question 3:

Skip to content

Log in

This is a copy of a conversation between ChatGPT & Anonymous.

Report conversation

Uploaded a file Uploaded a file

Here, I uploaded the two files necessary for the problem, "Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer" and "A Spectral Condition for Feature Learning"

3. 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 Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer. 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?

Figure 1 in *Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer* plots **training loss versus learning rate** for Transformers of various widths trained with Adam.

- **Left panel (Standard Practice):** Each curve corresponds to a different model width (128 → 8192). As width increases, the **optimal learning rate shifts**, meaning that the best learning rate depends strongly on width. Wider models can even perform *worse* than narrower ones, since they are not correctly scaled.
- **Right panel (Our Work = μP):** Again, training loss vs. learning rate is shown for the same widths, but now under the **maximal update parameterization (μP)**. The **optimal learning rate stays nearly constant** across widths, and training loss monotonically improves as width increases.

Observation about the width of a matrix:

Under standard parametrization, increasing width changes how much each layer's updates affect activations—wide matrices receive disproportionately large or small updates, causing instability. In μP, parameter and learning-rate scalings are chosen so that **each layer's updates remain o** Skip to content **regardless of width**, leading to stable behavior and transferable hyperparameters across widths 2203.03466.

Sources

(b) Look at Table 3 in the paper. In lecture, we gave an elementary argument to support the 1/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?

In **Table 3** of the *Tensor Programs V* paper 2203.03466, the **rightmost entry in the bottom row** corresponds to the **Adam learning rate for hidden weights** under **Maximal Update Parameterization (μP)**.

That entry is:

> **μP (Adam LR, hidden weights): 1 / fan_in**

The table also shows the **corresponding "Standard Parameterization (SP)"** entry directly beneath the "Adam LR" heading for hidden weights, which is:

> **SP (Adam LR, hidden weights): 1 (constant)**

So, summarizing:

Layer type	Adam LR (μP)	Adam LR (Standard Param.)
Hidden weights	1 / fan_in	1

Answer:
In standard parameterization, the corresponding Adam learning rate for hidden weights is **1 (constant)** — it does **not** scale with fan-in.

Sources

Skip to content

(c) The paper A Spectral Condition for Feature Learning presents the same scaling using elementary linear algebra. While understanding the entirely 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 ‖hℓ(x)‖2 ≤ Θ(√nℓ) and that ‖∆hℓ(x)‖2 ≤
√
Θ(nℓ)? What is the key assumption that allows us to also get the lower bounds on ‖hℓ(x)‖2
and ‖∆hℓ(x)‖2?

Let's unpack this step by step using *A Spectral Condition for Feature Learning*

A Spectral Condition for Featur… .

---

1. Desideratum 1 (from the paper)

It states:

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

Why this behavior is desired for feature learning

This ensures that:

- Each feature vector $h_\ell(x)$ has elements of **order one** (i.e., $O(1)$ magnitude per neuron), so activations don't blow up or vanish as width increases.
- Each feature **update** $\Delta h_\ell(x)$ is also $O(1)$ per element, meaning each neuron continues to *learn* as the network widens.

Skip to content

If instead features or updates scaled than $\sqrt{n_\ell}$, learning would either diverge (too large) or freeze (too small, as width grows). Thus, this scaling guarantees **nontrivial feature evolution** at all widths—what the paper calls “feature learning.”

---

2. Rewrite Desideratum 1 in terms of RMS norm

Define the **RMS norm** of a layer’s feature vector as

$$\lVert h_\ell(x) \rVert_{\mathrm{RMS}} \equiv \frac{\lVert h_\ell(x) \rVert_2}{\sqrt{n_\ell}}.$$

Then Desideratum 1 becomes:

$$\lVert h_\ell(x) \rVert_{\mathrm{RMS}} = \Theta(1), \quad \lVert \Delta h_\ell(x) \rVert_{\mathrm{RMS}} = \Theta(1).$$

In words:
Each neuron’s typical activation and update magnitude are of constant order, independent of network width.

Rather than defining the RMS-RMS norm of a matrix in terms of the spectral norm of a matrix, hallucinated the definition of the RMS matrix norm. It defined it as if the matrix was unrolled into a vector and then the RMS norm of that flattened vector was taken,

---

3. Rewrite Condition 1 in terms of RMS norm

Condition 1 (Spectral Scaling Condition) states:

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

To express this in RMS terms, recall that the RMS of a matrix is

$$\lVert W_\ell \rVert_{\mathrm{RMS}} = \frac{\lVert W_\ell \rVert_F}{\sqrt{n_\ell n_{\ell-1}}}.$$

For random matrices whose spectral and Frobenius norms have the usual relation $\lVert W_\ell \rVert_* \approx \lVert W_\ell \rVert_F / \sqrt{\min(n_\ell, n_{\ell-1})}$, this scaling means

$$\lVert W_\ell \rVert_{\mathrm{RMS}} = \Theta\left(\frac{1}{\sqrt{n_{\ell-1}}}\right), \quad \lVert \Delta W_\ell \rVert_{\mathrm{RMS}} = \Theta\left(\frac{1}{\sqrt{n_{\ell-1}}}\right),$$

which is the same as saying “weights (and their updates) scale like $1/\sqrt{\text{fan-in}}$”—exactly the μP scaling.

Skip to content

Concept	**Expression**	**rm form**	**Meaning**
Desideratum 1	$\lVert h_\ell \rVert_2, \lVert \Delta h_\ell \rVert_2 = \Theta(\sqrt{n_\ell})$	$\lVert h_\ell \rVert_{\text{RMS}}, \lVert \Delta h_\ell \rVert_{\text{RMS}} = \Theta(1)$	Feature and update magnitudes per neuron stay constant.
Condition 1	$\lVert W_\ell \rVert_*, \lVert \Delta W_\ell \rVert_* = \Theta(\sqrt{n_\ell / n_{\ell-1}})$	$\lVert W_\ell \rVert_{\text{RMS}}, \lVert \Delta W_\ell \rVert_{\text{RMS}} = \Theta(1/\sqrt{n_{\ell-1}})$	Weight spectral norms scale like $\sqrt{\text{fan-out/fan-i}}$.
Implication	Submultiplicativity gives ≤ bounds	Alignment gives ≥ bounds	Together yield full Θ-scaling → feature learning at all widths.

Sources

For RMS norm of a matrix, define it to be the induced RMS- RMS matrix norm (i.e. ‖A‖_{p,q}=max_{‖x‖_p≠0} ‖Ax‖_q/‖x‖_p, where the RMS norm of a vector is ‖x‖_2/sqrt(d)

Good — you're defining an **induced RMS–RMS operator norm**, meaning both the input and output vector norms are RMS-normalized (so that the "size per coordinate" is invariant with dimension). Let's formalize this and then rewrite **Condition 1** and **Desideratum 1** in that language.

---

1. RMS vector norm

For a vector $x \in \mathbb{R}^d$,

$$\lVert x \rVert_{\text{RMS}} = \frac{\lVert x \rVert_2}{\sqrt{d}}.$$

This measures the *average magnitude per coordinate* rather than total magnitude.

Skip to content

2. Induced RMS–RMS matrix no

For a matrix $A \in \mathbb{R}^{m\times n}$, define

$$\|A\|_{\mathrm{RMS}\to\mathrm{RMS}} = \max_{x\neq 0} \frac{\|Ax\|_{\mathrm{RMS}}}{\|x\|_{\mathrm{RMS}}} = \max_{x\neq 0} \frac{\|Ax\|_2/\sqrt{m}}{\|x\|_2/\sqrt{n}} = \sqrt{\frac{n}{m}}\,\|A\|_*.$$

So the RMS–RMS operator norm rescales the spectral norm by $\sqrt{n/m}$, where n is the input dimension and m is the output dimension.

---

3. Rewriting Condition 1 (spectral scaling)

Original form from the paper:

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

Convert to the induced RMS–RMS norm using the above relation:

$$\|W_\ell\|_{\mathrm{RMS}\to\mathrm{RMS}} = \sqrt{\frac{n_{\ell-1}}{n_\ell}}\,\|W_\ell\|_* = \Theta(1),$$

and likewise

$$\|\Delta W_\ell\|_{\mathrm{RMS}\to\mathrm{RMS}} = \Theta(1).$$

Interpretation:
Under the RMS–RMS normalization, both the weights and their gradient updates act as **order-one operators**: they neither amplify nor shrink RMS activations as width grows.

---

4. Rewriting Desideratum 1

Original:

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

In RMS form:

$$\|h_\ell(x)\|_{\mathrm{RMS}} = \Theta(1), \qquad \|\Delta h_\ell(x)\|_{\mathrm{RMS}} = \Theta(1).$$

That is, the RMS magnitude per feat Skip to content remain $O(1)$ — no explosion or vanishing with width.

---

5. How Condition 1 ⇒ upper bounds

From the forward relation $h_\ell = W_\ell h_{\ell-1}$:

$$\|h_\ell\|_{\mathrm{RMS}} = \frac{\|W_\ell h_{\ell-1}\|_2}{\sqrt{n_\ell}} \le \frac{\|W_\ell\|_* \|h_{\ell-1}\|_2}{\sqrt{n_\ell}} = \|W_\ell\|_{\mathrm{RMS}\to\mathrm{RMS}} \|h_{\ell-1}\|_{\mathrm{RMS}} = \Theta(1)\,\Theta(1) = \Theta(1).$$

Multiplying back by $\sqrt{n_\ell}$ gives

$$\|h_\ell\|_2 \le \Theta(\sqrt{n_\ell}).$$

The same argument, applied to
$\Delta h_\ell = \Delta W_\ell h_{\ell-1} + W_\ell \Delta h_{\ell-1} + \dots,$
gives
$\|\Delta h_\ell\|_2 \le \Theta(\sqrt{n_\ell}).$

---

6. Key assumption for lower bounds

To strengthen "≤" into full "Θ(·)" (i.e., to get matching lower bounds), we need the **alignment assumption** from the paper:

- **Claim 1:** Random weight matrices W_ℓ roughly scale their inputs by their spectral norm (so the upper bound is tight).
- **Claim 2:** Gradient updates ΔW_ℓ are rank-1 and aligned with $h_{\ell-1}$.

These imply that $W_\ell h_{\ell-1}$ and $\Delta W_\ell h_{\ell-1}$ actually *achieve* the upper-bound scaling, yielding

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

Summary

Concept	Spectral-norm form	RMS–RMS form	Meaning
Condition 1	$\|W_\ell\|_* = \Theta(\sqrt{n_\ell / n_{\ell-1}})$	$\|W_\ell\|_{\text{RMS}\to\text{RMS}} = \Theta(1)$	Each layer's operator and its update are $O(1)$ in RMS sense

Message ChatGPT

Attach Search Study Voice

With a course-correcting prompt to provide the correct definition of the RMS-RMS norm it was able to one shot the entire problem

Question 4:

This is a copy of a conversation between ChatGPT & Anonymous.

Report conversation

4. Policy Gradient and the Reparameterization Gradient
Estimator
In this question, you are going to derive two gradient
estimators for the following objective function:
F(θ) = Ex∼pθ (1)
where x is a random variable that follows the probability
distribution of pθ: X → ∆(X) that is parameterized by θ ∈
R^P , and f : X → R is a function
(a) Let x be a k-D multivariate Gaussian random variable
that is parameterized by the mean μ ∈ R
K
under the distribution,
pμ(x) = (2πσ)
−n/2
exp
−‖x − μ‖
2
2/(2σ
2
)
, (2)
where σ ∈ R is a scalar constant. Express ∇μF(μ) as an
expectation under pμ(x) (i.e., find g(x) in
∇F(μ) = E_{x∼pμ} [g(X)]. (Hint: Ex∼pθ[f(x)] = \int
pθ(x)f(x)dx)

We want $\nabla_\mu F(\mu) = \nabla_\mu \mathbb{E}_{x\sim p_\mu}[f(x)] = \nabla_\mu \int p_\mu(x)\, f(x)\, dx$.

Differentiate under the integral:

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

For the Gaussian (mean μ, covariance $\sigma^2 I$), Rather than directly taking the derivative of the pdf, it define the gradient of the pdf as the pdf * log (pdf) and took the gradient of that. Though this method was not as direct, I appreciated the different perspective (which was actually used later in the problem solution anyway)

$\log p_\mu(x) = -k$ Skip to content $- \frac{1}{2\sigma^2}\|x-\mu\|^2,$

so

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

Therefore one valid choice of $g(x)$ is

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

(b) Rewrite the expression in F(μ) with an expectation with the distribution over a standard normal
N (0, I)? (Hint: if x follows the distribution of pμ(x), then x−μ follows the distribution of N (0, σ2
I))

We start from

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

where

$$x \sim \mathcal{N}(\mu, \sigma^2 I).$$

Step 1. Change of variables

Let's define a new variable

$$\varepsilon = \frac{x-\mu}{\sigma} \quad \Longleftrightarrow \quad x = \mu + \sigma\varepsilon.$$

If $x \sim \mathcal{N}(\mu, \sigma^2 I)$, then $\varepsilon \sim \mathcal{N}(0, I)$.

Step 2. Rewrite the expectation

Using this substitution:

Skip to content

$$F(\mu) = \mathbb{E}_{x\sim\mathcal{N}(\mu,\sigma^2 I)}[f(x)] = \mathbb{E}_{\varepsilon\sim\mathcal{N}(0,I)}[f(\mu+\sigma\varepsilon)].$$

✅ Final Answer

$$\boxed{F(\mu) = \mathbb{E}_{\varepsilon\sim\mathcal{N}(0,I)}[f(\mu+\sigma\varepsilon)].}$$

This expresses the same objective as an expectation over a *standard normal* random variable ε.

(c) Using the expression you obtained from the previous part, can you express ∇μF(μ) in a similar way (i.e., with an expectation with the distribution over pμ=0(x))? (Hint: ∇μ can be safely moved inside the expectation because the expectation no longer follows a distribution that depends on μ and expectations are linear.)

Yes — using the reparameterized form from (b),

$$F(\mu) = \mathbb{E}_{\varepsilon\sim\mathcal{N}(0,I)}\left[f(\mu+\sigma\varepsilon)\right],$$

we can move ∇_μ inside the expectation (the integrand no longer depends on μ through the sampling distribution, and differentiation under the integral is allowed). Then

$$\begin{aligned}\nabla_\mu F(\mu) &= \nabla_\mu \mathbb{E}_\varepsilon[f(\mu+\sigma\varepsilon)] = \mathbb{E}_\varepsilon\left[\nabla_\mu f(\mu+\sigma\varepsilon)\right] \\ &= \mathbb{E}_\varepsilon\left[\nabla_x f(\mu+\sigma\varepsilon)\,\frac{\partial(\mu+\sigma\varepsilon)}{\partial\mu}\right] = \mathbb{E}_\varepsilon\left[\nabla_x f(\mu+\sigma\varepsilon)\right].\end{aligned}$$

So a convenient Monte-Carlo estimator is

$$\boxed{\nabla_\mu F(\mu) = \mathbb{E}_{\varepsilon\sim\mathcal{N}(0,I)}\left[\nabla_x f(\mu+\sigma\varepsilon)\right]},$$

i.e. sample $\varepsilon \sim \mathcal{N}(0, I)$, compute $x = \mu + \sigma\varepsilon$, evaluate $\nabla_x f(x)$, and average.

(You can check this equals the score Skip to content m part (a) under mild smoothness conditions.)

For the following two parts, we are going to generalize our findings from the specific Gaussian case to arbitrary pθ's (you may assume pθ has well-defined gradient ∇θpθ that expresses what change in parameters would make a particular realization more likely).
(d) In the general case, can you still write ∇F(θ) as an expectation under pθ(x)? (Hint: look at your answer in Part (a) and see how the term inside your expectation relates to log pθ(x)).

Yes. Exactly the same trick as in (a) works for any differentiable family p_θ. Start from

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

differentiate under the integral (assume the usual regularity conditions so this is allowed):

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

So you can write

$$\boxed{\nabla_\theta F(\theta) = \mathbb{E}_{x\sim p_\theta}\big[g(x)\big] \quad \text{with} \quad g(x) = f(x)\, \nabla_\theta \log p_\theta(x).}$$

This estimator is commonly called the score-function (or REINFORCE / likelihood-ratio) estimator.

A useful variance-reduction variant is to subtract any baseline b independent of x:

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

which has the same expectation but often lower variance.

ChatGPT 5 gave more information than necessary to solve the problem, seemed to provide helpful contextualization

Skip to content

(e) Assume there exists a function g(z, θ) : Z ×R
P 7→ X and a distribution over z, p(z), such that g(z, θ)has
the same distribution as x ~ pθ(x). Prove that ∇θF(θ) =
Ez~Z h
∇θg(z)
⊤∇xf|x=g(z,θ)
iIn the
previous part (b) and (c), we actually prove a special case
of this. Can you determine what g and
p(z) are for the special case?

Thought for 5s

Write

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

By assumption there exists a measurable map $g : Z \times \mathbb{R}^P \to X$ and a fixed distribution $p(z)$ such that the random variable $g(z, \theta)$ with $z \sim p(z)$ has the same law as $x \sim p_\theta$. Hence we can rewrite the objective as an expectation over z:

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

Now assume the usual regularity conditions that allow us to (i) push the gradient inside the expectation (dominated convergence / bounded derivative / integrability) and (ii) apply the chain rule to the integrand. For fixed z,

Highlighted key conditions to push gradient into expectation

$$\nabla_\theta\left[f(g(z,\theta))\right] = \big(\nabla_\theta g(z,\theta)\big)^\top \nabla_x f\big(g(z,\theta)\big),$$

where $\nabla_\theta g(z,\theta)$ is the Jacobian matrix of g with respect to θ (shape $|X| \times P$) and $\nabla_x f$ is the gradient of f w.r.t. x (shape $|X| \times 1$). Taking expectation over z and moving ∇_θ inside gives

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

This is exactly the reparameterization-gradient formula you were asked to prove.

Special case (Gaussian) identification

For the Gaussian example in parts (a)–(c) we had $x \sim \mathcal{N}(\mu, \sigma^2 I)$ and we reparametrized by $\varepsilon \sim \mathcal{N}(0, I)$ with

$$x = \mu + \sigma\varepsilon.$$

Thus in the notation above:

- $z = \varepsilon, p(z) = \mathcal{N}(0, I)$.
- $\theta = \mu$.
- $g(z, \theta) = \theta + \sigma z$.

Compute the Jacobian: $\nabla_\theta g(z, \theta) = I$. Plugging into the general formula yields

$$\nabla_\mu F(\mu) = \mathbb{E}_{\varepsilon \sim \mathcal{N}(0,I)}\left[I^\top \nabla_x f(\mu + \sigma\varepsilon)\right] = \mathbb{E}_\varepsilon\left[\nabla_x f(\mu + \sigma\varepsilon)\right],$$

which matches the result you obtained in part (c).

Message ChatGPT
Attach Search Study Voice

Question 5:

Skip to content

Log in

This is a copy of a conversation between ChatGPT & Anonymous.

Report conversation

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.

Skip to content

Figure rialization in action - Layer 5 and Layer stored in memory along with the inputs. Activations for other layers are recomputed on-demand from stored activations and inputs.
(a) 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.

`Thought for 18s`

You get **20 fwd operations**.

Reasoning: checkpoints are at inputs (layer 0), layer 5, and layer 10. For each layer i during backprop we load the *last stored* checkpoint before i and re-run forward up to i. Counting fwd ops per layer (layers counted 1..10):

- Layers 1–4: start from inputs → fwd ops = 4 + 3 + 2 + 1 = 10.
- Layer 5: activation is stored → 0 fwd ops.
- Layers 6–9: start from layer 5 → fwd ops = 1 + 2 + 3 + 4 = 10.
- Layer 10: activation is stored → 0 fwd ops.

Total $= 10 + 0 + 10 + 0 = 20$.

(b) 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?

Let’s carefully go step by step.

Stored activations:

- Inputs (layer 0)
- Layer 5
- Layer 10

During backprop, we start from the top (layer 10 → 1), loading the nearest stored activation below the layer we are processing.

Backprop through layers 10 → 6

Stored activation for **layer 10** is already in memory, so:

- **Layer 10:** no load needed (activation already stored).
- **Layer 9:** need activations up to layer 9 → load **layer 5** activation (nearest stored one) once.
- Then use it to recompute 6→9.
- **Layers 8, 7, 6**: still use same cached activation from layer 5 (no additional loads).

→ **1 loadmem** total for this block.

Backprop through layers 5 → 1

Stored activation for **layer 5** is already in memory.

- **Layer 5:** no load (stored).
- **Layer 4, 3, 2, 1:** to get activations, we must load **inputs** (the checkpoint before layer 1).
  Once loaded, we can reuse it for all 4 layers.

→ **1 loadmem** total for this block.

Total

- 1 load (for layer 5 checkpoint)
- 1 load (for input checkpoint)

$$\boxed{\texttt{2 loadmem operations}}$$

Skip to content

With just reiterating the key part of the problem statement, it was able to correctly derive that each layer required loading an activation

Remember, we are 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

Good — that clarification matters.
We're told **"for each layer,"** we **load the relevant last stored activation from memory**, then recompute forward until that layer.

So even if the same checkpoint (say, layer 5) could *in principle* be reused, the problem statement explicitly says that **each time we process a new layer during backprop**, we again **load from memory** the most recent stored activation.

Let's go layer by layer.

Stored activations:

- Inputs (layer 0)
- Layer 5
- Layer 10

We ignore the initial forward pass (loss computation) — we only count during backprop.

Backprop layers 10 → 6

For each layer $i \in \{10, 9, 8, 7, 6\}$:

- To recompute activations for layer i, we must **load** the most recent stored activation **below** i (which is layer 5).
- That means **one loadmem per layer**.

So layers 10–6 contribute **5 loadmem ops**.

Backprop layers 5 → 1

For each layer $i \in \{5, 4, 3, 2, 1\}$

Skip to content

- The nearest stored activation below these layers is the **inputs**.
- Again, we must load the relevant stored activation (inputs) for each layer separately.

So layers 5–1 contribute **5 loadmem ops**.

✅ Total loadmem operations

$$5 + 5 = \boxed{\texttt{10 loadmem operations}}$$

(c) 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 20ns and each loadmem operation (which loads from memory, not local disk) takes 10ns 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.

Let's restate what we have so far.

From parts (a) and (b):

Skip to content

Method	# fwd ops	s	Description
Tensor rematerialization	20	10	recompute activations by re-running forwards, each layer reloads last stored activation
Disk-storage (no rematerialization)	0	0	store all activations to disk and load each when needed

Now we'll compute the total cost of **tensor rematerialization**, then find what the **loaddisk time** must be to match it.

Step 1. Compute tensor rematerialization time

Each forward pass = **20 ns**, each loadmem = **10 ns**.

$$T_{\mathtt{remat}} = 20 \times 20 + 10 \times 10 = 400 + 100 = 500\ \mathtt{ns}$$

Step 2. Compute cost for full-storage (disk) approach

If we store all activations, we **don't need to recompute any**; we just load each activation when needed during backprop.

For a 10-layer network, there are **10 activations** to load (one per layer) during the backward pass.

$$T_{\mathtt{disk}} = 10 \times t_{\mathtt{loaddisk}}$$

Step 3. Equate total times

We want them to be equal in time:

$$T_{\mathtt{disk}} = T_{\mathtt{remat}}$$

$$10\, t_{\mathtt{loaddisk}} = 500\ \mathtt{ns}$$

$$\boxed{t_{\mathtt{loaddisk}} = 50\ \mathtt{ns}}$$

✅ Final Answer:

Each **loaddisk operation** must take **50 ns** to make the full-storage approach as fast as

Message ChatGPT

Attach Search Study Voice