

EECS 182 Participation A: Claude Sonnet 4.5 on HW 3

Note: A link to the raw conversation can be found [here](#)

1 Maximal Update Parameterization

During lecture, we learned how SGD and Adam can be seen as constrained optimization problems, defined by some norm over parameter space. In this problem (which builds directly on the last discussion section), we will explore the payoff for defining these norms over the output features of a dense layer and then bringing it back to the matrices involved.

- (a) Consider a dense layer with input $\mathbf{x} \in \mathbb{R}^{d_1}$, weights $W \in \mathbb{R}^{d_2 \times d_1}$, where W is initialized with i.i.d. standard Gaussian entries, and entries of \mathbf{x} are sampled from an i.i.d. unit Gaussian.

What is the expected squared RMS norm of the output features $\mathbf{y} = W\mathbf{x}$? 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 $W\mathbf{x}$ 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 $\mathbf{y} = W\mathbf{x}$? What is its variance?

Response: I'll work through this step-by-step.

Step 1: Single output feature case

For $W \in \mathbb{R}^{1 \times d_1}$ and $\mathbf{x} \in \mathbb{R}^{d_1}$: - $y = W\mathbf{x} = \sum_{i=1}^{d_1} W_i x_i$ - Since $W_i \sim \mathcal{N}(0, 1)$ and $x_i \sim \mathcal{N}(0, 1)$ are independent, each product $W_i x_i \sim \mathcal{N}(0, 1)$ - The sum of d_1 independent $\mathcal{N}(0, 1)$ variables gives $y \sim \mathcal{N}(0, d_1)$ - Therefore, $\mathbb{E}[y^2] = d_1$

Step 2: Full layer with d_2 output features

For $W \in \mathbb{R}^{d_2 \times d_1}$: - Each output feature $y_j = W_j \mathbf{x}$ has $\mathbb{E}[y_j^2] = d_1$ - The squared RMS norm is: $\frac{1}{d_2} \|\mathbf{y}\|^2 = \frac{1}{d_2} \sum_{j=1}^{d_2} y_j^2$ - Taking expectation: $\mathbb{E} \left[\frac{1}{d_2} \|\mathbf{y}\|^2 \right] = \frac{1}{d_2} \sum_{j=1}^{d_2} \mathbb{E}[y_j^2] = \frac{1}{d_2} \cdot d_2 \cdot d_1 = d_1$

Answer: - Expected squared RMS norm: d_1

- Scaling: Scales linearly with d_1 , independent of d_2

- Normalization constant: Multiply W by $\frac{1}{\sqrt{d_1}}$ to ensure expected squared RMS norm equals 1

This is the $\frac{1}{\sqrt{\text{fan}_{\text{in}}}}$ (also known as Xavier/He initialization for this case).

Comments: Claude was able to solve this part without any help. It followed the hint provided, which likely made its solution more concise.

- (b) 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 $g_i \in \mathbb{R}^{d_2}$.

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

What is the expected RMS norm squared of the (unscaled) change in features $y' = \operatorname{sign}(x_i g_i^T) x_i$? 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 ?

Response: I'll work through the expected RMS norm squared of the feature update.

Setup:

- Update: $W_{t+1} \leftarrow W_t + \eta \operatorname{sign}(x_i g_i^T)$
- Change in features: $y' = \operatorname{sign}(x_i g_i^T) x_i \in \mathbb{R}^{d_2}$

Step 1: Analyze a single output feature

For the j -th component of y' :

$$y'_j = \sum_{k=1}^{d_1} \operatorname{sign}(x_k g_j) \cdot x_k$$

Since $x_k \sim \mathcal{N}(0, 1)$ and g_j is constant: - $\operatorname{sign}(x_k g_j) = \operatorname{sign}(x_k) \cdot \operatorname{sign}(g_j)$ - So $\operatorname{sign}(x_k g_j) \cdot x_k = \operatorname{sign}(g_j) \cdot |x_k|$

Therefore:

$$y'_j = \operatorname{sign}(g_j) \sum_{k=1}^{d_1} |x_k|$$

Step 2: Compute variance

Since $x_k \sim \mathcal{N}(0, 1)$, we have $|x_k|$ follows a half-normal distribution with:

- $\mathbb{E}[|x_k|] = \sqrt{\frac{2}{\pi}}$
- $\mathbb{E}[|x_k|^2] = 1$, so $\mathbb{E}[|x_k|^2] = 1$
- $\operatorname{Var}(|x_k|) = 1 - \frac{2}{\pi}$

For $y'_j = \operatorname{sign}(g_j) \sum_{k=1}^{d_1} |x_k|$:

$$\begin{aligned} \mathbb{E}[(y'_j)^2] &= \operatorname{Var}\left(\sum_{k=1}^{d_1} |x_k|\right) + \left(\mathbb{E}\left[\sum_{k=1}^{d_1} |x_k|\right]\right)^2 \\ &= d_1 \left(1 - \frac{2}{\pi}\right) + d_1 \cdot \frac{2}{\pi} = d_1 \end{aligned}$$

Step 3: Compute RMS norm squared

$$\mathbb{E} \left[\frac{1}{d_2} \|y'\|^2 \right] = \frac{1}{d_2} \sum_{j=1}^{d_2} \mathbb{E}[(y'_j)^2] = \frac{d_2 \cdot d_1}{d_2} = d_1$$

Answer:

- Expected RMS norm squared: d_1
- Scaling: Scales linearly with d_1 , independent of d_2
- Normalization constant: Multiply the update by $\frac{1}{\sqrt{d_1}}$ to ensure expected RMS norm equals 1

Not quite. $\left(\mathbb{E} \left[\sum_{k=1}^{d_1} |x_k| \right] \right)^2$ should be $d_1^2 \times \left(\frac{2}{\pi}\right)$

Response: You're absolutely right! Let me correct that.

Step 2 (Corrected): Compute variance

For $y'_j = \text{sign}(g_j) \sum_{k=1}^{d_1} |x_k|$:

$$\mathbb{E} \left[\sum_{k=1}^{d_1} |x_k| \right] = d_1 \cdot \sqrt{\frac{2}{\pi}}$$

So:

$$\left(\mathbb{E} \left[\sum_{k=1}^{d_1} |x_k| \right] \right)^2 = d_1^2 \cdot \frac{2}{\pi}$$

And:

$$\begin{aligned} \mathbb{E}[(y'_j)^2] &= \text{Var} \left(\sum_{k=1}^{d_1} |x_k| \right) + \left(\mathbb{E} \left[\sum_{k=1}^{d_1} |x_k| \right] \right)^2 \\ &= d_1 \left(1 - \frac{2}{\pi} \right) + d_1^2 \cdot \frac{2}{\pi} \end{aligned}$$

Step 3: Compute RMS norm squared

$$\begin{aligned} \mathbb{E} \left[\frac{1}{d_2} \|y'\|^2 \right] &= \frac{1}{d_2} \sum_{j=1}^{d_2} \mathbb{E}[(y'_j)^2] = \frac{d_2}{d_2} \left[d_1 \left(1 - \frac{2}{\pi} \right) + d_1^2 \cdot \frac{2}{\pi} \right] \\ &= d_1 - \frac{2d_1}{\pi} + \frac{2d_1^2}{\pi} = d_1 \left(1 - \frac{2}{\pi} + \frac{2d_1}{\pi} \right) \end{aligned}$$

Answer:

- Expected RMS norm squared: $d_1 \left(1 - \frac{2}{\pi} + \frac{2d_1}{\pi} \right) \approx \frac{2d_1^2}{\pi}$ for large d_1
- Scaling: Scales as $O(d_1^2)$ for large d_1 , independent of d_2
- Normalization constant: Multiply by $\frac{1}{d_1} \sqrt{\frac{\pi}{2}}$ to ensure expected RMS norm equals 1

Comments: Claude was mostly able to get this answer correct, however, it made an error in its first pass through the problem and got an initial result of $1/\sqrt{d_1}$. Looking through its initial response, the only thing Claude got incorrect was not properly squaring the expectation $\mathbb{E} \left[\sum_{k=1}^{d_1} |x_k| \right]$. I find this very interesting that Claude forgot to do this, because (a) it seems like just a typo, and (b) in its incorrect solution, a lot of the nasty terms cancel out leading to a supposedly clean solution. Maybe Claude suspected the answer should be very clean and wanted to get a result that matched that assumption? If so, I would very much relate.

Another factor to mention is that Claude takes a similar approach to part (a) by considering a single output feature to solve the problem. I wonder if by not including the hint in part (a), if Claude does not take this approach. Still, Claude does a bit more work in finding the exact constants needed to scale our update rather than just considering the asymptotic case. I feel like this is more of a problem with the wording of the question rather than Claude's reasoning, as the question does not specify that the solution can be given in the asymptotic case.

- (c) You may notice that the above update rule only depends on d_1 . **Why is this the case?**

Response: The update rule only depends on d_1 because of how the dimensions interact in the computation:

Key observation:

When we compute the change in features $y' = \text{sign}(x_i g_i^T) x_i$:

- Each output feature y'_j is computed as a sum over the input dimension d_1 :

$$y'_j = \sum_{k=1}^{d_1} \text{sign}(x_k g_j) \cdot x_k$$

- The RMS norm averages over the output dimension d_2 , but since all d_2 output features have the same expected squared magnitude (by symmetry), this averaging doesn't change the scaling behavior.

Why d_1 matters but d_2 doesn't:

- d_1 (fan-in): Determines how many terms we're summing over when computing each output feature. More input dimensions means more terms accumulate, increasing the magnitude.

- d_2 (fan-out): Only determines how many independent output features we compute. Since we normalize by d_2 when computing the RMS norm (dividing by d_2 in $\frac{1}{d_2} \|y'\|^2$), the effect of d_2 cancels out.

This is analogous to part (a) where the initialization also only depended on d_1 (fan-in), not d_2 .

Comments: I would argue that Claude's response here is correct, even though it is not exactly the same as the key. Claude mostly focuses on the dimensions, but briefly adds in the (fan-out) section that the output features are independent, which mostly explains why d_2 , which only determines the number of output features we have, is not needed for scaling.

2 Maximal Update Parameterization Research

Note: For this question, I provided Claude a PDF file of the research papers that it could reference to answer each part.

This homework problem talks about the research papers behind maximal update parameterization, also called μ P 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?**

Response: Looking at Figure 1 of the paper:

What the figures are plotting:

Both side-by-side figures plot training loss against learning rate (on a log scale) for Transformers of varying widths (d_{model}), trained with Adam. Different colored curves represent different model widths (128, 256, 512, 1024, 2048, 4096, 8192).

Key observations about width:

1. Left (Standard Practice): The optimal learning rate shifts significantly as width increases. Wider networks require different (generally smaller) optimal learning rates. Additionally, wider networks don't consistently outperform narrower ones - in fact, even with tuned learning rates, the widest models can underperform (see the dashed line comparison).
2. Right (Their Work/ μ P): The optimal learning rate remains stable across all widths. All curves have their minimum at approximately the same learning rate value. Furthermore, wider networks consistently achieve better (lower) training loss than narrower networks at the same learning rate.

Main takeaway: Under standard parametrization, hyperparameters like learning rate don't transfer well across model scales, but under μ P (maximal update parametrization), the optimal hyperparameters are stable across widths, enabling "zero-shot" hyperparameter transfer from small to large models.

Comments: Claude was able to answer this question correctly and concisely. I quite like that they added a "main takeaway" to the figure, even though it was not necessary to answer the question.

- (b) Look at Table 3 in the paper. In lecture, we gave an elementary argument to support the $1/\text{fan}_{in}$ 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?**

Response: Looking at Table 3 in the paper:

For hidden weights with Adam optimizer the table shows:

- μP (Maximal Update Parametrization): Adam LR = $1/\text{fan}_{\text{in}}$
- Standard Parametrization (SP): Adam LR = 1 (shown in gray text in parentheses)

So the corresponding standard parametrization uses a learning rate of 1 (i.e., constant with respect to width), whereas μP scales it as $1/\text{fan}_{\text{in}}$ to ensure proper feature learning as the network width increases.

This difference is crucial - in SP, using a constant learning rate leads to the instability and lack of hyperparameter transferability shown in the left panel of Figure 1, while μP 's $1/\text{fan}_{\text{in}}$ scaling enables stable hyperparameter transfer across widths.

Comments: Claud answered this question correctly. Again, there's not too much to this question, but I do appreciate the explanation for this difference that Claude provides, and it helped me understand a bit better what the table was describing.

- (c) The paper *A Spectral Condition for Feature Learning* 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 $\|\Delta h_\ell(x)\|_2$?

Response: Great question! Let me work through each part:

Why is Desideratum 1 desired for feature learning?

Desideratum 1 asks that $\|h_\ell\|_2 = \Theta(\sqrt{n_\ell})$ and $\|\Delta h_\ell\|_2 = \Theta(\sqrt{n_\ell})$.

Why this is desired: 1. Typical element size is (1): If $\|h_\ell\|_2 = \Theta(\sqrt{n_\ell})$, then the "typical" entry of h_ℓ has size $\Theta(1)$ (since there are n_ℓ entries). This keeps activations in a reasonable numerical range for activation functions.

2. Nontrivial feature updates: Similarly, $\|\Delta h_\ell\|_2 = \Theta(\sqrt{n_\ell})$ means features undergo $\Theta(1)$ changes per entry during training. If updates were smaller (e.g., $o(\sqrt{n_\ell})$), features would vanish at large width. If larger (e.g., $\omega(\sqrt{n_\ell})$), they would explode.

3. Feature learning at all widths: This ensures that internal representations evolve meaningfully during training, regardless of network width.

Rewriting in terms of RMS norm

The RMS (root-mean-square) norm of a vector $v \in \mathbb{R}^n$ is:

$$\text{RMS}(v) = \sqrt{\frac{1}{n} \|v\|_2^2} = \frac{1}{\sqrt{n}} \|v\|_2$$

Desideratum 1 in RMS form:

$$\text{RMS}(h_\ell) = \Theta(1) \quad \text{and} \quad \text{RMS}(\Delta h_\ell) = \Theta(1)$$

Condition 1 in RMS form: For $W_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$:

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

This can be rewritten as:

$$\text{RMS}_{\text{spectral}}(W_\ell) = \Theta(1) \quad \text{and} \quad \text{RMS}_{\text{spectral}}(\Delta W_\ell) = \Theta(1)$$

where the "spectral RMS" accounts for the matrix dimensions.

How Condition 1 implies the upper bounds

Upper bound on $\|h_\ell(x)\|_2$:

By submultiplicativity of the spectral norm:

$$\|h_\ell(x)\|_2 = \|W_\ell h_{\ell-1}(x)\|_2 \leq \|W_\ell\|_* \cdot \|h_{\ell-1}(x)\|_2$$

By induction, assuming $\|h_{\ell-1}(x)\|_2 = \Theta(\sqrt{n_{\ell-1}})$ and using Condition 1:

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

Upper bound on $\|\Delta h_\ell(x)\|_2$:

From the update equation: $\Delta h_\ell(x) = \Delta W_\ell h_{\ell-1}(x) + W_\ell \Delta h_{\ell-1}(x) + \Delta W_\ell \Delta h_{\ell-1}(x)$

Using the triangle inequality and submultiplicativity:

$$\|\Delta h_\ell(x)\|_2 \leq \|\Delta W_\ell\|_* \cdot \|h_{\ell-1}(x)\|_2 + \|W_\ell\|_* \cdot \|\Delta h_{\ell-1}(x)\|_2 + \|\Delta W_\ell\|_* \cdot \|\Delta h_{\ell-1}(x)\|_2$$

All three terms are $\Theta(\sqrt{n_\ell})$, giving the upper bound.

Key assumption for lower bounds: Alignment

The key assumption is that vectors align with the top singular subspaces of weight matrices and updates:

Claim 1 (Alignment of weights): For random initialization, $\|W_\ell h_{\ell-1}(x)\|_2 = \Theta(\|W_\ell\|_* \cdot \|h_{\ell-1}(x)\|_2)$

Claim 2 (Alignment of updates): For gradient updates, $\|\Delta W_\ell h_{\ell-1}(x)\|_2 = \|W_\ell\|_* \cdot \|h_{\ell-1}(x)\|_2$ (exact equality!)

Why alignment matters: Without alignment, the upper bound inequality could be very loose. For example, if $h_{\ell-1}(x)$ aligned only with small singular values of W_ℓ , then $\|W_\ell h_{\ell-1}(x)\|_2 \ll \|W_\ell\|_* \cdot \|h_{\ell-1}(x)\|_2$.

Why alignment occurs: - At initialization: Random matrices with sufficient rank naturally align incoming vectors with their top singular subspace - During training: Gradient

updates are rank-1 (outer products $\nabla_{h_\ell} L \cdot h_{\ell-1}^T$), so $h_{\ell-1}$ is exactly the right singular vector of ΔW_ℓ , giving perfect alignment!

This alignment turns the inequalities into (approximate) equalities, establishing the matching lower bounds.

Comments: Claude wrote a lot for this question, but it is all generally correct. It is able to identify that Desideratum 1 allows for entries of h_ℓ to have size $\Theta(1)$ and updates of $\Theta(1)$, and even further describes why updates of different sizes would be bad. It is also able to rewrite Condition 1 in terms of the RMS norm, although I found it a little odd how Claude formatted the norm, making it look more like a function rather than a norm. It argues how Condition 1 implies the upper bounds almost exactly how the answer key does it, and lastly identifies Claim 1 and Claim 2 as giving the lower bounds.

I would have appreciated a shorter answer, but overall not much else to be said. I am quite surprised how well Claude was able to synthesize the information from the PDFs to answer these questions, especially because these papers are not short (48 pages and 24 pages respectively).

3 Policy Gradient and the Reparameterization Gradient Estimator

In this question, you are going to derive two gradient estimators for the following objective function:

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

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

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

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

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 $\nabla F(\mu) = \mathbb{E}_{\mathbf{x} \sim p_\mu}[g(\mathbf{x})]$). (Hint: $\mathbb{E}_{x \sim p_\theta} f(x) = \int p_\theta(\mathbf{x}) f(\mathbf{x}) d\mathbf{x}$.)

Response: I'll derive the gradient $\nabla_\mu F(\mu)$ as an expectation under $p_\mu(x)$.

Setup

We have:

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

Derivation

Taking the gradient with respect to μ :

$$\nabla_\mu F(\mu) = \nabla_\mu \int p_\mu(\mathbf{x}) f(\mathbf{x}) d\mathbf{x}$$

Assuming we can interchange the gradient and integral (by dominated convergence theorem):

$$\nabla_\mu F(\mu) = \int \nabla_\mu p_\mu(\mathbf{x}) \cdot f(\mathbf{x}) d\mathbf{x}$$

Key Trick: Log-derivative

Use the identity: $\nabla_\mu p_\mu(\mathbf{x}) = p_\mu(\mathbf{x}) \nabla_\mu \log p_\mu(\mathbf{x})$

This gives:

$$\begin{aligned} \nabla_\mu F(\mu) &= \int p_\mu(\mathbf{x}) \nabla_\mu \log p_\mu(\mathbf{x}) \cdot f(\mathbf{x}) d\mathbf{x} \\ &= \mathbb{E}_{\mathbf{x} \sim p_\mu}[f(\mathbf{x}) \nabla_\mu \log p_\mu(\mathbf{x})] \end{aligned}$$

Computing $\nabla_\mu \log p_\mu(\mathbf{x})$

For the Gaussian distribution:

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

Taking the gradient with respect to μ :

$$\begin{aligned}\nabla_\mu \log p_\mu(\mathbf{x}) &= \nabla_\mu \left(-\frac{\|\mathbf{x} - \mu\|_2^2}{2\sigma^2} \right) \\ &= \nabla_\mu \left(-\frac{1}{2\sigma^2} \sum_{i=1}^k (x_i - \mu_i)^2 \right) \\ &= -\frac{1}{2\sigma^2} \cdot 2(\mu - \mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma^2}\end{aligned}$$

Final Result

Therefore:

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

Or equivalently:

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

This is known as the REINFORCE or score function gradient estimator (also called the policy gradient in reinforcement learning contexts).

Comments: Claude was able to get this question correct, but they did it in a kind of funny way. Rather than plugging in the gaussian equation into the integral immediately, they essentially solve for the general case and then plug in the gaussian. By doing this, they basically already solved part (d) of this question. Claude was also able to identify this as the policy gradient used in the REINFORCE algorithm, which is a nice touch.

- (b) Rewrite the expression in $F(\mu)$ with an expectation with the distribution over a standard normal $\mathcal{N}(0, \mathbf{I})$? (Hint: if \mathbf{x} follows the distribution $p_\mu(\mathbf{x})$, then $\mathbf{x} - \mu$ follows the distribution $\mathcal{N}(0, \sigma^2 \mathbf{I})$.)

Response: I'll rewrite $F(\mu)$ as an expectation over a standard normal distribution $\mathcal{N}(0, \mathbf{I})$.

Using the Reparameterization

Key insight: If $\mathbf{x} \sim p_\mu(\mathbf{x}) = \mathcal{N}(\mu, \sigma^2 \mathbf{I})$, we can write:

$$\mathbf{x} = \mu + \sigma \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ is a standard normal random variable.

Verification

Let's verify this satisfies the hint:

- If $\mathbf{x} \sim \mathcal{N}(\mu, \sigma^2 \mathbf{I})$
- Then $\mathbf{x} - \mu \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$
- So $\frac{\mathbf{x} - \mu}{\sigma} = \epsilon \sim \mathcal{N}(0, \mathbf{I})$

Rewriting $F(\mu)$

Starting with:

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

Using the reparameterization $\mathbf{x} = \mu + \sigma\epsilon$:

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

This is the reparameterization trick, which expresses the expectation over the parameter-dependent distribution p_μ as an expectation over a fixed, parameter-independent distribution $\mathcal{N}(0, \mathbf{I})$. The dependence on μ is now explicit in the argument of f .

Comments: Claude also got this question correct. I think it's nice that Claude made sure to verify their solution with the provided hint, although I don't think they would have needed it to get this question correct.

- (c) 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 $p_{\mu=0}(x)$)? (*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.*)

Response: I'll derive $\nabla_\mu F(\mu)$ using the reparameterized form from the previous part.

Starting Point

From the previous part, we have:

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

Taking the Gradient

Since the distribution $\mathcal{N}(0, \mathbf{I})$ does not depend on μ , we can move the gradient inside the expectation:

$$\nabla_\mu F(\mu) = \nabla_\mu \mathbb{E}_{\epsilon \sim \mathcal{N}(0, \mathbf{I})}[f(\mu + \sigma\epsilon)]$$

$$= \mathbb{E}_{\epsilon \sim \mathcal{N}(0, \mathbf{I})}[\nabla_\mu f(\mu + \sigma\epsilon)]$$

Computing the Inner Gradient

Now, μ appears only in the argument of f . Using the chain rule:

$$\nabla_\mu f(\mu + \sigma\epsilon) = \nabla_{\mathbf{x}} f(\mathbf{x}) \Big|_{\mathbf{x}=\mu+\sigma\epsilon} \cdot \nabla_\mu(\mu + \sigma\epsilon)$$

Since $\nabla_\mu(\mu + \sigma\epsilon) = \mathbf{I}$ (the identity matrix), we have:

$$\nabla_\mu f(\mu + \sigma\epsilon) = \nabla_{\mathbf{x}} f(\mathbf{x}) \Big|_{\mathbf{x}=\mu+\sigma\epsilon}$$

Final Result

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

Or equivalently, since $\mathbf{x} = \mu + \sigma\epsilon$ when $\epsilon \sim \mathcal{N}(0, \mathbf{I})$:

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

This is the reparameterization gradient estimator (also called the pathwise derivative estimator). It's often preferred over the policy gradient because it typically has lower variance, as it directly uses the gradient of f rather than multiplying f by a score function.

Comments: Claude was able to get this question correct, although it added a few lines to explain that the gradient of $\mu + \sigma\epsilon$ is just \mathbf{I} , and does the chain rule. I like this addition, since I think it's a detail that can be easily overlooked.

For the following two parts, we are going to generalize our findings from the specific Gaussian case to arbitrary p_θ 's (you may assume p_θ has a well-defined gradient $\nabla_\theta p_\theta$ that expresses what change in parameters would make a particular realization more likely).

- (a) In the general case, **can you still write $\nabla F(\theta)$ as an expectation under $p_\theta(x)$?**
 Hint: look at your answer in Part (a) and see how the term inside your expectation relates to $\log p_\theta(x)$.

Response: Yes! I can write $\nabla_\theta F(\theta)$ as an expectation under $p_\theta(x)$ for the general case.

Derivation

Following the same approach as in part (a):

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

Taking the gradient with respect to θ :

$$\nabla_{\theta} F(\theta) = \nabla_{\theta} \int p_{\theta}(\mathbf{x}) f(\mathbf{x}) d\mathbf{x}$$

$$= \int \nabla_{\theta} p_{\theta}(\mathbf{x}) \cdot f(\mathbf{x}) d\mathbf{x}$$

Log-Derivative Trick

Using the identity $\nabla_{\theta} p_{\theta}(\mathbf{x}) = p_{\theta}(\mathbf{x}) \nabla_{\theta} \log p_{\theta}(\mathbf{x})$:

$$\nabla_{\theta} F(\theta) = \int p_{\theta}(\mathbf{x}) \nabla_{\theta} \log p_{\theta}(\mathbf{x}) \cdot f(\mathbf{x}) d\mathbf{x}$$

Final Result

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

This is the policy gradient theorem or REINFORCE gradient estimator.

Key observation from part (a): The term inside the expectation involves $\nabla_{\theta} \log p_{\theta}(\mathbf{x})$, which is called the score function. This is exactly what we computed in part (a) for the Gaussian case, where we found $\nabla_{\mu} \log p_{\mu}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma^2}$.

This gradient estimator works for any parameterized distribution p_{θ} , not just Gaussians, making it very general and widely applicable (e.g., in reinforcement learning with discrete action spaces where reparameterization doesn't apply).

Comments: Claude basically already answered this question in the first part of the question, but correctly reiterated itself for this part, and even explained how the Gaussian case applied here.

- (b) 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** $\nabla_{\theta} F(\theta) = \mathbb{E}_{z \sim p(z)} [\nabla_{\theta} g(z, \theta)^T \nabla_x f|_{x=g(z, \theta)}]$. In the previous parts (b) and (c), we actually proved a special case of this. **Can you determine what g and $p(z)$ are for the special case?**

Response: I'll prove the general reparameterization gradient formula and identify the special case.

Proof of the General Formula

Given:

- $g(z, \theta) : \mathcal{Z} \times \mathbb{R}^P \rightarrow \mathcal{X}$
- $p(z)$ is a distribution over z (independent of θ)
- $g(z, \theta) \sim p_{\theta}(x)$ when $z \sim p(z)$

Starting point:

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

Since $\mathbf{x} = g(z, \theta)$ has the same distribution as $\mathbf{x} \sim p_\theta$ when $z \sim p(z)$:

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

Taking the gradient:

Since $p(z)$ does not depend on θ , we can move the gradient inside:

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

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

Applying the chain rule:

Let $\mathbf{x} = g(z, \theta)$. By the chain rule:

$$\nabla_\theta f(g(z, \theta)) = \nabla_{\mathbf{x}} f(\mathbf{x}) \Big|_{\mathbf{x}=g(z,\theta)} \cdot \nabla_\theta g(z, \theta)$$

Since $\nabla_{\mathbf{x}} f$ is a row vector (or covector) and $\nabla_\theta g(z, \theta)$ is a Jacobian matrix, we can write:

$$\nabla_\theta f(g(z, \theta)) = \nabla_\theta g(z, \theta)^\top \nabla_{\mathbf{x}} f \Big|_{\mathbf{x}=g(z,\theta)}$$

Final result:

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

Identifying the Special Case from Parts (b) and (c)

In parts (b) and (c), we had:

- $\mathbf{x} = \mu + \sigma \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$

Therefore, for the special case:

$$\boxed{g(z, \theta) = \mu + \sigma z}$$

where:

- $z = \boldsymbol{\epsilon}$

- $\theta = \mu$

- $p(z) = \mathcal{N}(0, \mathbf{I})$ (standard normal distribution)

Verification:

$$\nabla_\mu g(z, \mu) = \nabla_\mu (\mu + \sigma z) = \mathbf{I}$$

So:

$$\begin{aligned}\nabla_{\mu} F(\mu) &= \mathbb{E}_{z \sim \mathcal{N}(0, \mathbf{I})} \left[\mathbf{I}^{\top} \nabla_{\mathbf{x}} f \Big|_{\mathbf{x}=\mu+\sigma z} \right] \\ &= \mathbb{E}_{z \sim \mathcal{N}(0, \mathbf{I})} [\nabla_{\mathbf{x}} f(\mu + \sigma z)]\end{aligned}$$

which matches our result from part (c)!

Comments: Claude was also able to get this question correct, and had good explanations for each step. Like in part (c) where Claude explained how the chain rule had to be applied, here, the chain rule is necessary in order to get the question correct. This is why I appreciated the reasoning in that earlier step. Claude also verifies his results from parts (b) and (c) using his solution, which is a nice touch.

4 Tensor Rematerialization

Note: For this question, I provided Claude a PNG image of the figure that they could use to reference.

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 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 until 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 obtain 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.

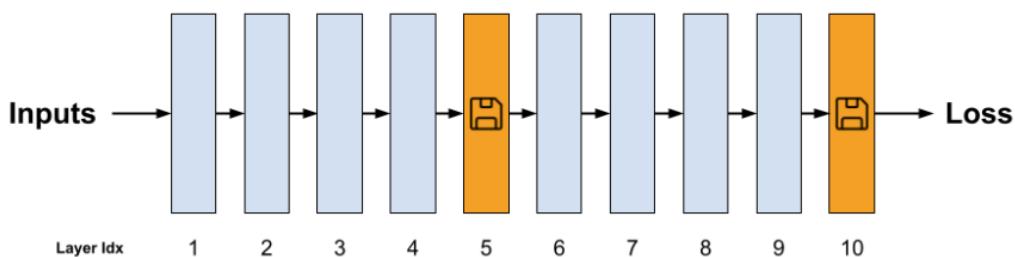


Figure 1: Tensor rematerialization in action - Layer 5 and Layer 10 activations are stored in memory along with the inputs. Activations for other layers are recomputed on-demand from stored activations and inputs..

- (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 do not worry about any extra computation beyond activations to actually backpropagate gradients.

Response: I'll work through this tensor rematerialization problem step by step.

Understanding the Setup

- 10-layer network, each layer has input/output size s - Memory constraint: can only store **2s activations** (plus inputs and optimizer states) - Strategy: Store activations only at layers 5 and 10 during initial forward pass - During backprop: recompute activations on-the-fly by loading last stored activation and rerunning forward

Counting `fwd` Operations During Backward Pass

Let me trace through what happens during backpropagation for each layer:

Layer 10 (output layer):

- Activations already stored from initial forward pass
- 0 `fwd` operations needed

Layer 9:

- Need Layer 9 activations
- Load Layer 5 activations from memory
- Run: Layer 6 Layer 7 Layer 8 Layer 9
- 4 `fwd` operations

Layer 8:

- Need Layer 8 activations
- Load Layer 5 activations from memory
- Run: Layer 6 Layer 7 Layer 8
- 3 `fwd` operations

Layer 7:

- Need Layer 7 activations
- Load Layer 5 activations from memory
- Run: Layer 6 Layer 7
- 2 `fwd` operations

Layer 6:

- Need Layer 6 activations
- Load Layer 5 activations from memory
- Run: Layer 6
- 1 `fwd` operation

Layer 5:

- Activations already stored from initial forward pass
- 0 `fwd` operations needed

Layer 4:

- Need Layer 4 activations
- Load inputs from memory

- Run: Layer 1 Layer 2 Layer 3 Layer 4
- 4 **fwd** operations

Layer 3:

- Need Layer 3 activations
- Load inputs from memory
- Run: Layer 1 Layer 2 Layer 3
- 3 **fwd** operations

Layer 2:

- Need Layer 2 activations
- Load inputs from memory
- Run: Layer 1 Layer 2
- 2 **fwd** operations

Layer 1:

- Need Layer 1 activations
- Load inputs from memory
- Run: Layer 1
- 1 **fwd** operation

Total Count

$$\text{Total} = 0 + 4 + 3 + 2 + 1 + 0 + 4 + 3 + 2 + 1 = \boxed{20 \text{ fwd operations}}$$

This is exactly twice the number of layers in the network (excluding the checkpoint layers themselves), which makes sense: we're essentially doing 2 forward passes through the network during backpropagation.

Comments: Claude's solution is correct, and uses the same logic as the answer key to come to their conclusion. I like how they went layer by layer and explained how many forward passes were required for each one.

- (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?**

Response: I'll count the **loadmem** operations needed during the backward pass.

Counting **loadmem** Operations

Recall that we need to load stored activations (or inputs) from memory each time we want to recompute activations for a layer during backpropagation.

Layer 10:

- Activations already stored, no recomputation needed
- 0 **loadmem** operations

Layer 9:

- Need to recompute from Layer 5's stored activations

- Load Layer 5 activations: 1 `loadmem`

Layer 8:

- Need to recompute from Layer 5's stored activations
- Load Layer 5 activations: 1 `loadmem`

Layer 7:

- Need to recompute from Layer 5's stored activations
- Load Layer 5 activations: 1 `loadmem`

Layer 6:

- Need to recompute from Layer 5's stored activations
- Load Layer 5 activations: 1 `loadmem`

Layer 5:

- Activations already stored, no recomputation needed
- 0 `loadmem` operations

Layer 4:

- Need to recompute from inputs
- Load inputs: 1 `loadmem`

Layer 3:

- Need to recompute from inputs
- Load inputs: 1 `loadmem`

Layer 2:

- Need to recompute from inputs
- Load inputs: 1 `loadmem`

Layer 1:

- Need to recompute from inputs
- Load inputs: 1 `loadmem`

Total Count

$$\text{Total} = 0 + 1 + 1 + 1 + 1 + 0 + 1 + 1 + 1 + 1 = \boxed{8 \text{ loadmems operations}}$$

This equals the number of non-checkpoint layers (8 layers total, excluding layers 5 and 10 which are checkpointed). Each non-checkpoint layer requires exactly one memory load operation to fetch the nearest stored checkpoint.

Not quite. Even though we save the activations of layers 5 and 10 in memory, we still need to load them from memory during backprop. What should the answer be instead?

Response: You're absolutely right! Even though the activations are saved in memory, we still need to load them from memory to use them during backpropagation.

Let me recount:

Corrected Count of `loadmem` Operations

Layer 10:

- Activations are stored, but we need to load them to use during backprop
- 1 `loadmem` (load Layer 10 activations)

Layer 9:

- Load Layer 5 activations to recompute
- 1 `loadmem`

Layer 8:

- Load Layer 5 activations to recompute
- 1 `loadmem`

Layer 7:

- Load Layer 5 activations to recompute
- 1 `loadmem`

Layer 6:

- Load Layer 5 activations to recompute
- 1 `loadmem`

Layer 5:

- Activations are stored, but we need to load them to use during backprop
- 1 `loadmem` (load Layer 5 activations)

Layer 4:

- Load inputs to recompute
- 1 `loadmem`

Layer 3:

- Load inputs to recompute
- 1 `loadmem`

Layer 2:

- Load inputs to recompute
- 1 `loadmem`

Layer 1:

- Load inputs to recompute
- 1 `loadmem`

Total Count

$$\text{Total} = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = \boxed{10 \text{ } \text{loadmem} \text{ operations}}$$

We perform exactly one `loadmem` operation per layer during the backward pass each layer needs to load either stored checkpoint activations or inputs from memory.

Comments: Claude initially got this question incorrect, and I had to tell it that layers 5 and 10 still need to load their activations from memory even though we have them

stored in memory directly. I'm a bit confused about why Claude forgot to count the load memory operations for these layers, since it understands that for layers 6-9 we need to load from layer 5 to recompute the activations. However, after correcting it, Claude seems very capable of answering the question correctly.

- (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.

Response: I'll calculate the required speed of `loaddisk` operations to match the time of tensor rematerialization.

Time for Tensor Rematerialization (Backward Pass)

From the previous parts:

- 20 `fwd` operations at 20ns each = $20 \times 20 = 400$ ns
- 10 `loadmem` operations at 10ns each = $10 \times 10 = 100$ ns
- Total time = $400 + 100 = 500$ ns

Time for Disk-Based Approach (Backward Pass)

Without tensor rematerialization, we store all layer activations on disk and load them when needed during backprop.

What needs to be loaded? During backpropagation through the network, we need the activations from each layer to compute gradients. For each of the 10 layers, we need to load its activations once.

- Number of `loaddisk` operations = 10 (one for each layer)
- Number of `fwd` operations = 0 (no recomputation needed)

Let t = time per `loaddisk` operation in nanoseconds.

Total time = $10t$

Finding Required Speed

To match the tensor rematerialization time:

$$10t = 500$$

$$t = 50\text{ns}$$

Therefore, each `loaddisk` operation should take [50 ns] to match the performance of tensor rematerialization.

Interpretation: The disk would need to be 5 slower than memory (50ns vs 10ns) but still be competitive because we avoid all the recomputation overhead. However, if disk is slower than 50ns per load, tensor rematerialization becomes the faster approach despite requiring more computation.

Comments: Claude answered this question correctly. It uses the answers from the previous parts to help it answer the question which is good, and further identifies how long using a disk should take. The math itself is pretty straightforward after that.