Special Participation A - HW8

Hanna Roed
EECS 282A - Deep Neural Networks

November 17, 2025

This is my report on using Qwen3-Max on the written part of homework 8.
Concise summary of my findings:

Problem	**Outcome**
1(a)	one-shot
1(b)	one-shot
1(c)	one-shot
1(d)	minor correction
1(e)	extensive correction
1(f)	minor correction
2 (coding)	N/A
3(a)	one-shot
3(b)	one-shot
3(c)	one-shot
4(a)	one-shot
4(b)	one-shot
4(c)	one-shot
4(d)	one-shot
4(e)	one-shot
4(f)	one-shot
4(g)	one-shot

Overall, I'm very impressed by Qwen3-Max's performance on this homework. It seems like it really does well on the questions where it needs to fill in or do multiple choice, i. e., problems 3 and 4. The main issues I had were in the second half of problem 1, which was regarding computational efficiency; it had a hard time accurately responding in text along with giving a mathematical reasoning.

Therefore, my strategy after doing problem 1, was to start a new chat session with the same intro prompt that I gave it the first time in order to have a clean slate after the back and forth from the errors it made.

Introduction

Prompt:

You will be assisting a graduate student with a problem set on deep neural networks. Do not answer any problem at this time, only when I give you a problem. The following problems will be only theory based, so no coding should be done at any point. For the solutions you need to provide short, but sitll clear step-by-step derivation of your solution.

Response:

Understood. Please provide the first theory-based problem.

1 SSM Convolution Kernel

1.1 a)

Prompt:

1. SSM Convolution Kernel
Background and Setup: Consider a discrete-time State-Space Model (SSM) of the form

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k, \\ y_k &= Cx_k + Du_k, \end{aligned}$$

where $x_k \in \mathbb{R}^n$, $u_k \in \mathbb{R}^d$, $y_k \in \mathbb{R}^m$.
a) **Convolution Kernel and the Output Equation.** Given that the sequence length is L (input: $(u_0, \ldots, u_L)$, output: $(y_0, \ldots, y_L)$) and assume $x_0 = 0$, show that the output y_k can be expressed as a convolution of the input sequence $\{u_\ell\}_{\ell=0}^{L}$ with a kernel $K = \{K_\ell\}_{\ell=0}^{L}$:

$$y_k = \sum_{\ell=0}^{L} K_\ell \, u_{k-\ell},$$

where any u_j with a negative index $j < 0$ is set to 0 (zero-padding). Also, find K.

Response:

We are given a discrete-time linear time-invariant (LTI) state-space model (SSM):

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k, \\ y_k &= Cx_k + Du_k, \end{aligned}$$

with initial state $x_0 = 0$, and we are to show that the output y_k can be written as a convolution of the input sequence $\{u_\ell\}$ with a kernel $K = \{K_\ell\}$, and to find that kernel.
—

Step 1: Unroll the state equation
Since $x_0 = 0$, we can unroll the state recursion:

$$\begin{aligned}
x_1 &= Ax_0 + Bu_0 = Bu_0, \\
x_2 &= Ax_1 + Bu_1 = ABu_0 + Bu_1, \\
x_3 &= Ax_2 + Bu_2 = A^2Bu_0 + ABu_1 + Bu_2, \\
&\vdots \\
x_k &= \sum_{\ell=0}^{k-1} A^{k-1-\ell}Bu_\ell.
\end{aligned}$$

This can be written compactly as:

$$x_k = \sum_{\ell=0}^{k-1} A^{k-1-\ell}Bu_\ell, \quad \text{for } k \geq 1, \quad x_0 = 0.$$

—

Step 2: Plug into output equation
Now substitute into the output equation:

$$y_k = Cx_k + Du_k = C\left(\sum_{\ell=0}^{k-1} A^{k-1-\ell}Bu_\ell\right) + Du_k.$$

Change index of summation: let $j = k - \ell \Rightarrow \ell = k - j$. When $\ell = 0$, $j = k$; when $\ell = k - 1$, $j = 1$. So:

$$y_k = \sum_{j=1}^{k} CA^{j-1}Bu_{k-j} + Du_k.$$

Now include the Du_k term as the $j = 0$ term:

$$y_k = \sum_{j=0}^{k} K_j u_{k-j},$$

where we define:

$$K_j = \begin{cases} D, & j = 0, \\ CA^{j-1}B, & j \geq 1. \end{cases}$$

—

Step 3: Extend to full convolution over length L
For a sequence of length $L + 1$ (indices 0 to L), and using zero-padding for u_j when $j < 0$, we can write for all $k = 0, \ldots, L$:

$$y_k = \sum_{\ell=0}^{L} K_\ell u_{k-\ell},$$

with the convention $u_j = 0$ for $j < 0$ or $j > L$.
Thus, the system acts as a convolution with kernel $K = \{K_\ell\}_{\ell=0}^{L}$ given by:

$$\boxed{K_0 = D, \quad K_\ell = CA^{\ell-1}B \text{ for } \ell = 1, 2, \ldots, L.}$$

This completes the derivation.

Evaluation: Correct, one-shot.

1.2 b)

Prompt:

Concrete Examples.

1. **Scalar Case:** Let $n = 1$, and set $A = \alpha$, $B = \beta$, $C = \gamma$, $D = \delta$. Use $\alpha = 0.8$, $\beta = 1$, $\gamma = 1.5$ and compute the kernel up to $L = 4$.
2. **2D Case:** Let $A \in \mathbb{R}^{2\times 2}$ be, for instance,

$$A = \begin{bmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D = 0.$$

Compute kernels up to $L = 3$ and briefly discuss how the kernel captures the “impulse response”.

Response:

We now compute the convolution kernels for two concrete SSM instances using the formula derived in part (a):

$$K_0 = D, \quad K_\ell = CA^{\ell-1}B \text{ for } \ell \geq 1.$$

—

(i) Scalar Case
Given: - $A = \alpha = 0.8$,$B = \beta = 1$,$C = \gamma = 1.5$,$D = \delta$ (not explicitly given, so assume $D = 0$ unless stated; but since they ask to “compute the kernel” and give all other values, we assume $D = 0$ unless otherwise indicated — however, note: if D were nonzero, it would appear in K_0. Since D is not specified numerically, we assume $D = 0$.)
But actually—wait: the problem does list $D = \delta$, but does not give a numeric value for δ. However, in standard examples like this, if D is omitted numerically, it is often taken as 0. To be safe, we will assume $D = 0$ unless the problem implies otherwise. Let's proceed with $D = 0$.
Then for $\ell \geq 1$:

$$K_\ell = CA^{\ell-1}B = \gamma \cdot \alpha^{\ell-1} \cdot \beta = 1.5 \cdot (0.8)^{\ell-1} \cdot 1 = 1.5 \cdot (0.8)^{\ell-1}.$$

Compute up to $L = 4$:
- $K_0 = D = 0$

- $K_1 = 1.5 \cdot (0.8)^0 = 1.5$
- $K_2 = 1.5 \cdot (0.8)^1 = 1.5 \cdot 0.8 = 1.2$
- $K_3 = 1.5 \cdot (0.8)^2 = 1.5 \cdot 0.64 = 0.96$
- $K_4 = 1.5 \cdot (0.8)^3 = 1.5 \cdot 0.512 = 0.768$

Thus, the kernel is:

$$\boxed{K = \begin{bmatrix}0, \ 1.5, \ 1.2, \ 0.96, \ 0.768\end{bmatrix}}$$

(If $D \neq 0$, say $D = \delta$, then $K_0 = \delta$; but since no value is given, we take $D = 0$.)

—

(ii) 2D Case

Given:

$$A = \begin{bmatrix}0.7 & 0.1\\ 0.2 & 0.6\end{bmatrix}, \quad B = \begin{bmatrix}1\\ 0\end{bmatrix}, \quad C = \begin{bmatrix}1 & 0\end{bmatrix}, \quad D = 0.$$

We compute $K_\ell = CA^{\ell-1}B$ for $\ell = 1, 2, 3$. Since $D = 0$, $K_0 = 0$.
Step 1: $\ell = 1$ $A^0 = I$, so:

$$K_1 = CIB = CB = [1\ 0]\begin{bmatrix}1\\ 0\end{bmatrix} = 1.$$

Step 2: $\ell = 2$ Compute AB:

$$AB = \begin{bmatrix}0.7 & 0.1\\ 0.2 & 0.6\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \begin{bmatrix}0.7\\ 0.2\end{bmatrix}.$$

Then:

$$K_2 = C(AB) = [1\ 0]\begin{bmatrix}0.7\\ 0.2\end{bmatrix} = 0.7.$$

Step 3: $\ell = 3$ Compute $A^2B = A(AB) = A\begin{bmatrix}0.7\\ 0.2\end{bmatrix}$:

$$A^2B = \begin{bmatrix}0.7 & 0.1\\ 0.2 & 0.6\end{bmatrix}\begin{bmatrix}0.7\\ 0.2\end{bmatrix} = \begin{bmatrix}0.7 \cdot 0.7 + 0.1 \cdot 0.2\\ 0.2 \cdot 0.7 + 0.6 \cdot 0.2\end{bmatrix} = \begin{bmatrix}0.49 + 0.02\\ 0.14 + 0.12\end{bmatrix} = \begin{bmatrix}0.51\\ 0.26\end{bmatrix}.$$

Then:

$$K_3 = C(A^2B) = [1\ 0]\begin{bmatrix}0.51\\ 0.26\end{bmatrix} = 0.51.$$

So the kernel up to $L = 3$ is:

$$\boxed{K = \begin{bmatrix}0, \ 1, \ 0.7, \ 0.51\end{bmatrix}}$$

—

Impulse Response Interpretation

The kernel $K = \{K_\ell\}$ is precisely the impulse response of the system: it is the output sequence y_k when the input is a unit impulse $u_0 = 1$, $u_k = 0$ for $k > 0$, and zero initial state.
- In the scalar case, the impulse response decays exponentially, as expected for a stable first-order system ($|\alpha| < 1$). - In the 2D case, even though the state is 2-dimensional, the output

only observes the first state component ($C = [1\ 0]$), and the dynamics mix both states via A. The kernel values (1, 0.7, 0.51, ...) reflect how the initial impulse propagates through the internal state and leaks out over time.
Thus, the kernel captures how past inputs influence the current output via the hidden dynamics, and is fully determined by (A, B, C, D).

Evaluation: Correct, one-shot.

Note: For the scalar case given in this problem Qwen uses $D = 0$ as the first element of the kernel which i believe in correct according to what the solution is in part a) of this problem (contrary to the first element of the kernel being 1.5 in the handed out solution proposal).

1.3 c)

Prompt:

Efficient Computation with Convolutions. If we already know the kernel K, how much can we parallelize the computation of the output sequence $\{y_k\}$ for an input sequence $\{u_k\} \in \mathbb{R}^d$ of length L? What is the minimum critical path length of the computation? What about a naive, direct computation of y_k from the unrolled recursion?

Response:

We analyze the computational structure of generating the output sequence $\{y_k\}_{k=0}^{L}$ from an input $\{u_k\}_{k=0}^{L}$ in two ways: (1) using the precomputed convolution kernel K, and (2) via the naive unrolled state-space recursion.
—
1. Using the Precomputed Kernel K: Convolution View
From part (a), we have:

$$y_k = \sum_{\ell=0}^{k} K_\ell u_{k-\ell}, \quad \text{with } K_0 = D,\ K_\ell = CA^{\ell-1}B \text{ for } \ell \geq 1.$$

This is a causal convolution of input u with kernel K.
- The full output sequence $y = (y_0, ..., y_L)$ can be computed as the discrete convolution $y = K * u$, restricted to valid causal indices (with zero-padding for negative time indices).
- Convolution is a linear, shift-invariant operation and all outputs y_k can be computed independently once K and u are known. - Therefore, the computation is fully parallelizable across time steps: each y_k is a dot product between a slice of u and K.
Critical Path Length: - Each y_k requires a sum of up to $k+1 \leq L+1$ terms. - Assuming vector operations (e.g., dot products) can be done in $O(1)$ time with sufficient parallelism (e.g., using parallel reduction), the critical path for each y_k is $O(\log L)$ due to summation depth. - However, since all y_k are independent, the overall critical path of the entire sequence computation is just the time to compute the slowest single y_k, i.e., $O(\log L)$. - In practice,

with fixed-size dot products and hardware that computes them in constant time, the critical path is considered $O(1)$ in a parallel model.
Conclusion (kernel-based): - Highly parallelizable: all y_k computed simultaneously. - Critical path length: $O(\log L)$ (or $O(1)$ under unit-cost dot product assumption).
—

2. Naive Direct Computation via State Recursion
The state-space equations are:

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k, \\ y_k &= Cx_k + Du_k, \end{aligned} \quad x_0 = 0.$$

- To compute y_k, we must first compute x_k.
- But x_k depends on x_{k-1}, which depends on x_{k-2}, etc.
- Thus, the state must be computed sequentially: $x_0 \to x_1 \to \cdots \to x_L$.

Even if matrix-vector multiplications Ax_k are parallelized internally, the recurrence creates a data dependency chain across time.
Critical Path Length:
- Each step requires: (i) compute Ax_k, (ii) add Bu_k, (iii) compute $y_k = Cx_k + Du_k$.
- Assuming each step takes $O(1)$ time (with parallel matrix-vector ops), the total critical path is the number of time steps: $L + 1$.
- So critical path length is $O(L)$.

Conclusion (naive recursion):
- Not parallelizable across time: inherently sequential.
- Critical path length: $O(L)$.
—
Thus, knowing the kernel enables full parallelization, drastically reducing the critical path compared to the sequential state-space rollout. This is the key insight behind models like S4 and SSM-based sequence models, which leverage kernel convolution for efficient training/inference.

Evaluation: Correct, one-shot.

Note: It recognizes the paralellization which is the import part of uderstanding this subproblem.

1.4 d)

Prompt:

Efficient Kernel Computation. Given A, B, C, how can we compute the kernel K efficiently? What are some strategies to parallelize kernel computation? You may assume $L = 2^N$ for some N for simplicity.

Response:

Yes—divide-and-conquer provides a clean and powerful way to compute the SSM convolution kernel $K = \{K_\ell = CA^{\ell-1}B\}_{\ell=1}^{L}$ in parallel, especially when $L = 2^N$. Here's a step-by-step derivation of such a method.

—

Key Idea: Block Structure of the Kernel via State Propagation

The SSM can be viewed as a causal linear system: an impulse at time 0 produces output $K_1, K_2, \ldots$. Now consider splitting the time horizon $[0, L)$ into two halves: left $[0, M)$ and right $[M, L)$, where $M = L/2$.

- The kernel on the left half is simply $\{CA^{\ell-1}B\}_{\ell=1}^{M}$. - For the right half, note that the state at time M due to an impulse at time 0 is $x_M = A^M B$. - Then, for $\ell = M + r$ (with $r \geq 1$),

$$K_{M+r} = CA^{M+r-1}B = CA^{r-1}(A^M B).$$

That is, the right-half kernel is the impulse response of the same system, but with initial state $x'_0 = A^M B$ instead of input B.

However, since the system is linear and time-invariant, the response to initial state x'_0 is $CA^{r-1}x'_0$, which is not the same as convolving with B—unless we reinterpret.

But here's a better perspective:

—

Divide-and-Conquer via "Segmented" State Transitions

Define the transition matrix over a block of length τ as:

$$\Phi(\tau) = A^\tau.$$

Also define the block convolution response over length τ as:

$$\Psi(\tau) = \begin{bmatrix} CB \\ CAB \\ \vdots \\ CA^{\tau-1}B \end{bmatrix} \in \mathbb{R}^{\tau \times d}.$$

Now, suppose we want $\Psi(2\tau)$. We can write:

$$\Psi(2\tau) = \begin{bmatrix} \Psi(\tau) \\ CA^\tau B \\ CA^{\tau+1}B \\ \vdots \\ CA^{2\tau-1}B \end{bmatrix} = \begin{bmatrix} \Psi(\tau) \\ CA^\tau B \\ CA^\tau(AB) \\ \vdots \\ CA^\tau(A^{\tau-1}B) \end{bmatrix} = \begin{bmatrix} \Psi(\tau) \\ (CA^\tau)\begin{bmatrix} B & AB & \cdots & A^{\tau-1}B \end{bmatrix} \end{bmatrix}.$$

But note: $CA^{\tau+r}B = (CA^\tau)(A^r B)$. So the second half is:

$$\text{Second half} = \tilde{C} \cdot \begin{bmatrix} B & AB & \cdots & A^{\tau-1}B \end{bmatrix}, \quad \text{where } \tilde{C} = CA^\tau.$$

Thus, if we precompute: - $\Phi(\tau) = A^\tau$, - $\Psi(\tau) = \{CA^{\ell-1}B\}_{\ell=1}^{\tau}$,

then we can compute: - $\Phi(2\tau) = \Phi(\tau) \cdot \Phi(\tau) = A^{2\tau}$, - $\Psi(2\tau) = \begin{bmatrix} \Psi(\tau) \\ \tilde{C} \cdot [B, AB, \ldots, A^{\tau-1}B] \end{bmatrix}$, but the second term is just $\{\tilde{C}A^{\ell-1}B\}_{\ell=1}^{\tau}$, i.e., the kernel of the same system with output matrix $\tilde{C} = CA^{\tau}$.
So we recursively compute:

$$\mathrm{Kernel}(C, A, B, 2\tau) = \mathrm{concat}\Big(\mathrm{Kernel}(C, A, B, \tau),\ \mathrm{Kernel}(CA^{\tau}, A, B, \tau)\Big).$$

—

Recursive Algorithm (Assume $L = 2^N$)
Define a function ‘ComputeKernel(C, A, B, L)‘:
- If $L = 1$: return $[CB]$.
- Else:
1. Recursively compute $K_{\text{left}} = \mathrm{ComputeKernel}(C, A, B, L/2)$.
2. Compute $A^{L/2}$ (via repeated squaring or from prior recursion).
3. Compute $\tilde{C} = CA^{L/2}$.
4. Recursively compute $K_{\text{right}} = \mathrm{ComputeKernel}(\tilde{C}, A, B, L/2)$.
5. Return $\mathrm{concat}(K_{\text{left}}, K_{\text{right}})$.

To avoid redundant matrix power computations, we also return A^L from each recursive call. Thus, each level of recursion:
- Performs one matrix multiplication ($A^{L/2} \cdot A^{L/2}$) to get A^L,
- Performs one matrix-vector (or matrix-matrix) multiplication to get $\tilde{C} = CA^{L/2}$,
- Makes two recursive calls in parallel.

Since the two recursive calls are independent, they can be executed in parallel.
—

Complexity and Parallelism
- Recursion depth: $\log_2 L$.
- At each level, only constant sequential work (matrix multiplies), but the two halves are parallel.
- Critical path: $O(\log L)$ matrix multiplications.
- Total work: $O(Ln^2)$ (same as naive), but parallel depth is $O(\log L)$.

If A has structure (e.g., diagonal), matrix powers are cheap ($O(n)$), making this extremely efficient.
—

Conclusion
The divide-and-conquer strategy recursively splits the kernel into left and right halves, using the semigroup property $A^{i+j} = A^i A^j$ and the linearity of the output map. By propagating a modified output matrix CA^{mid} to the right half, the method enables logarithmic-depth parallel computation of the full kernel.
This is the foundation of fast SSM kernel algorithms (e.g., in S4, Structured State Space models), where such recursion (often combined with diagonalization) yields near-linear or

linear-time kernel generation with high parallelism.

Evaluation: Correct, after minor correction.

Note: I found that it was providing me with many methods and not really focusing on the details of the different approaches. I followed up with this prompt: What about applying a divide-and-conquer idea? And so it gave me a very in-depth divide and conquer idea similar to what was given in the solution proposal (perhaps a little too much detail).

1.5 e)

Prompt:

Adding structure to A. Suppose A is a diagonal matrix. How can we leverage this structure to compute the kernel K more efficiently? Answer clearly and short.

Response:

When A is diagonal, we do not need to precompute or store the matrices $I, A, A^2, \ldots, A^{L-1}$. Instead, we exploit the fact that:

$$K_\ell = \sum_{i=1}^{n} (C_i B_i)\, \lambda_i^{\ell-1}, \quad \ell = 1, \ldots, L.$$

So the entire kernel is a linear combination of n geometric sequences $\{\lambda_i^{\ell-1}\}_{\ell=1}^{L}$.
Computational Advantage:
- Precomputation: For each eigenvalue λ_i, precompute the vector

$$v^{(i)} = \left[1, \lambda_i, \lambda_i^2, \ldots, \lambda_i^{L-1}\right]^\top \in \mathbb{R}^L.$$

This can be done in $O(L)$ time per mode (or $O(\log L)$ depth with parallel prefix/pow).
- Combine: Compute

$$K[1:L] = \sum_{i=1}^{n} (C_i B_i)\, v^{(i)},$$

which is a weighted sum of n length-L vectors—fully parallelizable.
Savings:
- Memory: Store only n scalars $C_i B_i$ and n length-L vectors (or generate on the fly).
- Time: $O(nL)$ total work, vs. $O(n^2 L)$ for naive matrix powers.
- No matrix multiplications: Avoids $O(n^3 \log L)$ cost of repeated squaring.

Thus, diagonal A reduces kernel computation to scalar exponentiation and vector scaling, enabling highly efficient and parallelizable precomputation of the full kernel.

Evaluation: Correct, after extensive correction.

Note: I needed to explicitly ask for it to actually give me a relevant answer regarding the way we could use the diagonal aspect as an advantage, and I also needed to mention the idea of precomputing $(I, A, ..., A^{L-1})$ for it to provide any relevant response.

1.6 f)

Prompt:

Diagonal-plus-low-rank (DPLR) structure. Now if A has the following form:

$$A = I_n + pp^\top,$$

where $A \in \mathbb{R}^{n\times n}$, $p \in \mathbb{R}^n$. How can we leverage this structure to compute the kernel K more efficiently?

Response:

Start with the binomial expansion of A^L, where $A = I_n + pp^\top$:

$$A^L = (I_n + pp^\top)^L.$$

Since $pp^\top$ is rank-1, it satisfies $(pp^\top)^k = (p^\top p)^{k-1}pp^\top = \|p\|^{2(k-1)}pp^\top$ for all $k \geq 1$. Thus, all powers of $pp^\top$ are scalar multiples of $pp^\top$, and I_n commutes with $pp^\top$. Therefore, the binomial expansion simplifies to:

$$\begin{aligned} A^L &= I_n + \sum_{k=1}^{L} \binom{L}{k} (pp^\top)^k \\ &= I_n + \sum_{k=1}^{L} \binom{L}{k} \|p\|^{2(k-1)} pp^\top \\ &= I_n + \left(\sum_{k=1}^{L} \binom{L}{k} \|p\|^{2(k-1)} \right) pp^\top. \end{aligned}$$

Factor out $\|p\|^{-2}$ (assuming $p \neq 0$):

$$\sum_{k=1}^{L} \binom{L}{k} \|p\|^{2(k-1)} = \frac{1}{\|p\|^2} \sum_{k=1}^{L} \binom{L}{k} \|p\|^{2k} = \frac{(1+\|p\|^2)^L - 1}{\|p\|^2}.$$

Hence,

$$A^L = I_n + \big((1+\|p\|^2)^L - 1\big) \frac{pp^\top}{\|p\|^2} \cdot \|p\|^2 = I_n + \big((1+\|p\|^2)^L - 1\big) pp^\top.$$

So we obtain the closed-form:

$$\boxed{A^L = I_n + \beta_L \, pp^\top, \quad \text{where } \beta_L = (1+\|p\|^2)^L - 1.}$$

Now, for the kernel $K_\ell = CA^{\ell-1}B$ ($\ell \geq 1$):

$$\begin{aligned} K_\ell &= C\big(I_n + \beta_{\ell-1}pp^\top\big)B \\ &= CB + \beta_{\ell-1}(Cp)(p^\top B). \end{aligned}$$

Let $\gamma = 1 + \|p\|^2$, $a = CB$, $b = (Cp)(p^\top B)$. Then:

$$K_\ell = a + b\big(\gamma^{\ell-1} - 1\big) = (a - b) + b\,\gamma^{\ell-1}.$$

Thus, the entire kernel is a linear combination of a constant and a single geometric sequence, computable in $O(n)$ precomputation and $O(L)$ total time (or parallelizable in $O(\log L)$ depth). This leverages the DPLR structure for extreme efficiency.

Evaluation: Correct, after minor correction.

Note: I needed to make a minor correction by saying: Can you start with this: $A^L = (I_n + pp^T)^L$ so it does not go crazy with long mathematical derivations.

2 Coding SSM Forward

Coding question → skip

3 Self-Supervised Linear Purification

3.1 a)

Prompt:

3. Self-Supervised Linear Purification

Consider a linear encoder—square weight matrix $W \in \mathbb{R}^{m\times m}$—that we want to be a "purification" operation on m-dimensional feature vectors from a particular problem domain. We do this by using self-supervised learning to reconstruct n points of training data $X \in \mathbb{R}^{m\times n}$ by minimizing the loss:

$$L_1(W; X) = \|X - WX\|_F^2. \tag{1}$$

While the trivial solution $W = I$ can minimize the reconstruction loss (1), we will now see how weight-decay (or equivalently in this case, ridge-style regularization) can help us achieve non-trivial purification.

$$L_2(W; X, \lambda) = \underbrace{\|X - WX\|_F^2}_{\text{Reconstruction Loss}} + \lambda \underbrace{\|W\|_F^2}_{\text{Regularization Loss}} \tag{2}$$

Note above that λ controls the relative weighting of the two losses in the optimization.

a) Consider the simplified case for $m = 2$ with the following two candidate weight matrices:

$$W^{(\alpha)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad W^{(\beta)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

The training data matrix X is also given to you as follows:

$$X = \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix}.$$

1. Compute the reconstruction loss and the regularization loss for the two encoders, and fill in the missing entries in the table below.

Encoder	Reconstruction Loss	Regularization Loss
α	____	____
β	0.001	____

2. For what values of the regularization parameter λ is the identity matrix $W^{(\alpha)}$ higher loss L_2 in (2), as compared to $W^{(\beta)}$?

Response:

We are given a self-supervised linear purification problem with two candidate weight matrices and a data matrix $X \in \mathbb{R}^{2\times 4}$. The goal is to compute the reconstruction and regularization losses for two encoders and then determine for which values of λ the regularized loss L_2 favors $W^{(\beta)}$ over $W^{(\alpha)}$.

—

(a)(i) Compute losses

Definitions

- Reconstruction loss: $\|X - WX\|_F^2$ - Regularization loss: $\|W\|_F^2 = \sum_{i,j} W_{ij}^2$

Given:

$$X = \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix}$$

—

Encoder $W^{(\alpha)} = I$:

- Reconstruction loss:

$$X - W^{(\alpha)}X = X - X = 0 \Rightarrow \|0\|_F^2 = 0$$

- Regularization loss:

$$\|W^{(\alpha)}\|_F^2 = 1^2 + 0^2 + 0^2 + 1^2 = 2$$

So for α: - Reconstruction loss = 0 - Regularization loss = 2

—

Encoder $W^{(\beta)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$:

- Apply $W^{(\beta)}$ to X:

$$W^{(\beta)}X = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & x_1^{(4)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & x_2^{(4)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & x_1^{(4)} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

So the difference $X - W^{(\beta)}X$ is:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix}$$

- Reconstruction loss:

$$\|X - W^{(\beta)}X\|_F^2 = (0.02)^2 + (-0.01)^2 + (0.01)^2 + (-0.02)^2 = 0.0004 + 0.0001 + 0.0001 + 0.0004 = 0.001$$

(This matches the given value.)
- Regularization loss:

$$\|W^{(\beta)}\|_F^2 = 1^2 + 0^2 + 0^2 + 0^2 = 1$$

So for β: - Reconstruction loss = 0.001 (given) - Regularization loss = 1
—

Fill in the table:

Encoder	Reconstruction Loss	Regularization Loss
α	0	2
β	0.001	1

—

(a)(ii) For what λ is $L_2(W^{(\alpha)}) > L_2(W^{(\beta)})$?
Compute total losses:
- $L_2(W^{(\alpha)}) = 0 + \lambda \cdot 2 = 2\lambda$ - $L_2(W^{(\beta)}) = 0.001 + \lambda \cdot 1 = 0.001 + \lambda$
We want:

$$2\lambda > 0.001 + \lambda$$

Subtract λ from both sides:

$$\lambda > 0.001$$

Answer: For all $\lambda > 0.001$, the identity encoder $W^{(\alpha)}$ has higher total loss than $W^{(\beta)}$.
—

Final Answers:
(i) Table: - α: Reconstruction = 0, Regularization = 2 - β: Reconstruction = 0.001, Regularization = 1
(ii) $\lambda > 0.001$

Evaluation: Correct, one-shot.

3.2 b)

Prompt:

Now consider a generic square linear encoder $W \in \mathbb{R}^{m\times m}$ and the regularized objective L_2 reproduced below for your convenience:

$$L_2(W; X, \lambda) = \underbrace{\|X - WX\|_F^2}_{\text{Reconstruction Loss}} + \lambda \underbrace{\|W\|_F^2}_{\text{Regularization Loss}} .$$

Assume $\sigma_1 > \cdots > \sigma_m \geq 0$ are the m singular values in X, that the number of training points n is larger than the number of features m, and that X can be expressed in SVD coordinates as $X = U\Sigma V^\top$.

1. You are given that the optimizing weight matrix for the regularized objective L_2 above takes the following form. Fill in the empty matrices below.

$$\widehat{W} = [\_] \cdot \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2+\lambda} & & & \\ & \frac{\sigma_2^2}{\sigma_2^2+\lambda} & & \\ & & \ddots & \\ & & & \frac{\sigma_m^2}{\sigma_m^2+\lambda} \end{bmatrix} \cdot [\_\_] \tag{5}$$

2. Derive the above expression.
(Hint: Can you understand $L_2(W; X, \lambda)$ as a sum of m completely decoupled ridge-regression problems?)
(Hint: The Frobenius norm is equal to $\|A\|_F^2 := \text{tr}(AA^\top)$, and it is invariant under orthogonal transform. That is, $\|A\|_F^2 = \|UAV^\top\|_F^2$ for any orthogonal matrices U, V, and any rectangular matrix A, as long as U, A, V have compatible shapes.)

Response:

We are given a regularized self-supervised loss:

$$L_2(W; X, \lambda) = \|X - WX\|_F^2 + \lambda\|W\|_F^2,$$

with $X \in \mathbb{R}^{m\times n}$, $W \in \mathbb{R}^{m\times m}$, and SVD $X = U\Sigma V^\top$, where $U \in \mathbb{R}^{m\times m}$, $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_m) \in \mathbb{R}^{m\times m}$, and $V \in \mathbb{R}^{n\times m}$ has orthonormal columns ($V^\top V = I_m$).
We aim to find the minimizer $\widehat{W}$ of L_2.
—

(b)(i) Fill in the empty matrices

We claim the optimal $\widehat{W}$ has the form:

$$\widehat{W} = U \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2+\lambda} & & \\ & \ddots & \\ & & \frac{\sigma_m^2}{\sigma_m^2+\lambda} \end{bmatrix} U^\top.$$

So the blanks are:
- First bracket: U - Second bracket: $U^\top$
Answer:

$$\widehat{W} = U \cdot \operatorname{diag}\left(\frac{\sigma_i^2}{\sigma_i^2 + \lambda}\right)_{i=1}^m \cdot U^\top.$$

—

(b)(ii) Derivation
We minimize:

$$L_2(W) = \|X - WX\|_F^2 + \lambda\|W\|_F^2.$$

Step 1: Use SVD of X. Let $X = U\Sigma V^\top$. Since U is orthogonal ($U^\top U = I$), define a change of variable:

$$\widetilde{W} = U^\top W U.$$

Then $W = U\widetilde{W}U^\top$. Because the Frobenius norm is invariant under orthogonal transformations:
- $\|W\|_F^2 = \|\widetilde{W}\|_F^2$ - $\|X - WX\|_F^2 = \|U\Sigma V^\top - U\widetilde{W}U^\top U\Sigma V^\top\|_F^2 = \|U(\Sigma - \widetilde{W}\Sigma)V^\top\|_F^2 = \|\Sigma - \widetilde{W}\Sigma\|_F^2$
(because multiplying by orthogonal matrices U and $V^\top$ preserves Frobenius norm).
So the loss becomes:

$$L_2 = \|\Sigma - \widetilde{W}\Sigma\|_F^2 + \lambda\|\widetilde{W}\|_F^2.$$

Step 2: Analyze structure. Note: $\Sigma \in \mathbb{R}^{m\times m}$ is diagonal with entries σ_i. Let $\widetilde{W} = [\widetilde{w}_{ij}]$. Then:
- $(\widetilde{W}\Sigma)_{ij} = \widetilde{w}_{ij}\sigma_j$ - So $(\Sigma - \widetilde{W}\Sigma)_{ij} = \begin{cases} \sigma_i - \widetilde{w}_{ii}\sigma_i, & i = j \\ -\widetilde{w}_{ij}\sigma_j, & i \neq j \end{cases}$
Then the squared Frobenius norm is:

$$\|\Sigma - \widetilde{W}\Sigma\|_F^2 = \sum_{i=1}^m (\sigma_i - \widetilde{w}_{ii}\sigma_i)^2 + \sum_{i\neq j}(\widetilde{w}_{ij}\sigma_j)^2.$$

Add regularization:

$$L_2 = \sum_{i=1}^m (\sigma_i(1 - \widetilde{w}_{ii}))^2 + \sum_{i\neq j}(\widetilde{w}_{ij}\sigma_j)^2 + \lambda\left(\sum_{i=1}^m \widetilde{w}_{ii}^2 + \sum_{i\neq j}\widetilde{w}_{ij}^2\right).$$

Step 3: Minimize w.r.t. each entry.
- For off-diagonal entries $i \neq j$: the term is $(\widetilde{w}_{ij}\sigma_j)^2 + \lambda\widetilde{w}_{ij}^2 = \widetilde{w}_{ij}^2(\sigma_j^2 + \lambda)$. This is minimized at $\widetilde{w}_{ij} = 0$.
- For diagonal entries $i = j$: minimize $(\sigma_i(1 - \widetilde{w}_{ii}))^2 + \lambda\widetilde{w}_{ii}^2$.
Take derivative w.r.t. $\widetilde{w}_{ii}$:

$$\frac{d}{d\widetilde{w}_{ii}}\left[\sigma_i^2(1 - \widetilde{w}_{ii})^2 + \lambda\widetilde{w}_{ii}^2\right] = -2\sigma_i^2(1 - \widetilde{w}_{ii}) + 2\lambda\widetilde{w}_{ii}.$$

Set to zero:

$$-\sigma_i^2(1-\widetilde{w}_{ii})+\lambda\widetilde{w}_{ii}=0 \Rightarrow -\sigma_i^2+\sigma_i^2\widetilde{w}_{ii}+\lambda\widetilde{w}_{ii}=0 \Rightarrow \widetilde{w}_{ii}(\sigma_i^2+\lambda)=\sigma_i^2 \Rightarrow \widetilde{w}_{ii}=\frac{\sigma_i^2}{\sigma_i^2+\lambda}.$$

Step 4: Assemble solution.
Thus, $\widetilde{W}$ is diagonal with entries $\frac{\sigma_i^2}{\sigma_i^2+\lambda}$, and off-diagonals zero. Therefore,

$$\widehat{W}=U\widetilde{W}U^\top=U\cdot\operatorname{diag}\left(\frac{\sigma_i^2}{\sigma_i^2+\lambda}\right)\cdot U^\top.$$

This completes the derivation.
Interpretation via hint: Each row of W solves a ridge regression problem to predict the corresponding row of X from X itself. Because $X=U\Sigma V^\top$, the input covariance in the rotated basis is diagonal (Σ^2), decoupling the problems—exactly ridge regression per mode, yielding shrinkage factors $\frac{\sigma_i^2}{\sigma_i^2+\lambda}$.
—
Final Answers:
(i)

$$\widehat{W}=U\cdot\operatorname{diag}\left(\frac{\sigma_1^2}{\sigma_1^2+\lambda},\ldots,\frac{\sigma_m^2}{\sigma_m^2+\lambda}\right)\cdot U^\top.$$

(ii) Derived by changing variables to $\widetilde{W}=U^\top WU$, using orthogonal invariance of Frobenius norm, showing off-diagonal entries are zero at optimum, and solving 1D ridge regression for each diagonal entry, yielding the stated shrinkage factors.

Evaluation: Correct, one-shot.

3.3 c)

Prompt:

You are given that the data matrix $X\in\mathbb{R}^{8\times n}$ has the following singular values:

$$\{\sigma_i\}=\{10,8,4,1,0.5,0.36,0.16,0.01\}.$$

For what set of hyperparameter values λ can we guarantee that the learned purifier $\widehat{W}$ will preserve at least 80% of the feature directions corresponding to the first 3 singular vectors of X, while attenuating components in the remaining directions to at most 50% of their original strength?
(Hint: What are the two critical singular values to focus on?)

Response:

We are given:
- $X\in\mathbb{R}^{8\times n}$ with singular values:

$$\sigma_1=10,\ \sigma_2=8,\ \sigma_3=4,\ \sigma_4=1,\ \sigma_5=0.5,\ \sigma_6=0.36,\ \sigma_7=0.16,\ \sigma_8=0.01.$$

- From part (b), the optimal linear purifier is:

$$\widehat{W} = U\,\mathrm{diag}\,(\alpha_i)\,U^\top, \quad \text{where } \alpha_i = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}.$$

The action of $\widehat{W}$ on the i-th singular direction (i.e., u_i) is to scale it by α_i. So:
- Preservation: For $i = 1, 2, 3$, we require $\alpha_i \geq 0.8$. - Attenuation: For $i = 4, \ldots, 8$, we require $\alpha_i \leq 0.5$.
We seek all $\lambda > 0$ satisfying both conditions.
—

Step 1: Preservation condition (first 3 directions)
Require for $i = 1, 2, 3$:

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8 \quad \Longleftrightarrow \quad \sigma_i^2 \geq 0.8(\sigma_i^2 + \lambda) \quad \Longleftrightarrow \quad 0.2\sigma_i^2 \geq 0.8\lambda \quad \Longleftrightarrow \quad \lambda \leq \frac{0.2}{0.8}\sigma_i^2 = \frac{1}{4}\sigma_i^2.$$

So for each $i = 1, 2, 3$:
- $i = 1$: $\lambda \leq \frac{1}{4}(10^2) = 25$ - $i = 2$: $\lambda \leq \frac{1}{4}(8^2) = 16$ - $i = 3$: $\lambda \leq \frac{1}{4}(4^2) = 4$
The most restrictive is $\lambda \leq 4$.
So preservation requires:

$$\lambda \leq 4.$$

—

Step 2: Attenuation condition (remaining directions)
Require for $i = 4, \ldots, 8$:

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \leq 0.5 \quad \Longleftrightarrow \quad \sigma_i^2 \leq 0.5(\sigma_i^2 + \lambda) \quad \Longleftrightarrow \quad 0.5\sigma_i^2 \leq 0.5\lambda \quad \Longleftrightarrow \quad \lambda \geq \sigma_i^2.$$

So for each $i = 4, \ldots, 8$:
- $i = 4$: $\lambda \geq 1^2 = 1$ - $i = 5$: $\lambda \geq (0.5)^2 = 0.25$ - $i = 6$: $\lambda \geq (0.36)^2 = 0.1296$ - $i = 7$: $\lambda \geq (0.16)^2 = 0.0256$ - $i = 8$: $\lambda \geq (0.01)^2 = 0.0001$
The most restrictive (largest lower bound) is $\lambda \geq 1$.
So attenuation requires:

$$\lambda \geq 1.$$

—

Step 3: Combine conditions
We need:

$$1 \leq \lambda \leq 4.$$

Thus, the set of hyperparameter values λ that satisfy both requirements is:

$$\boxed{\lambda \in [1,\ 4]}.$$

Critical singular values: - $\sigma_3 = 4$ (smallest of the top-3 → sets upper bound) - $\sigma_4 = 1$ (largest of the bottom-5 → sets lower bound)
These are the two critical values, as hinted.

Evaluation: Correct, one-shot.

4 Ridge-Attention

4.1 a)

Prompt:

4. Ridge-Attention

In lecture, you saw how the standard softmax-attention mechanism can be viewed as a softened version of something akin to a nearest-neighbor model in which the value returned for a query reflects a weighted combination of the values that correspond to the keys closest to the query. In this view, the (key, value) pairs in the memory represent a kind of in-context "training data" and the query is a test input for which we want to predict the right output given that data.

To start, let us think about why it is possible to efficiently update simple averaging. Let

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i$$

be the average of n points. Use m, x_{n+1}, n and simple arithmetic operations to compute

$$m' = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i$$

—the average of all points including the new point x_{n+1}.

(Hint: Start by multiplying m by n.)

Response:

We are given the current average of n points:

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \Rightarrow \quad \sum_{i=1}^{n} x_i = nm$$

We want to compute the updated average after adding a new point x_{n+1}:

$$m' = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{1}{n+1}\left(\sum_{i=1}^{n} x_i + x_{n+1}\right) = \frac{1}{n+1}(nm + x_{n+1})$$

Final expression:

$$m' = \frac{nm + x_{n+1}}{n+1}$$

This allows efficient updating of the mean without storing all previous data points.

Evaluation: Correct, one-shot.

4.2 b)

Prompt:

Let us now shift to thinking about traditional ridge-regression with n training pairs (x_i, y_i) where x_i are d-dimensional vectors and y_i are scalars. Let

$$A = \begin{bmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_n^\top \end{bmatrix}, \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

so that we can find the familiar closed-form solution

$$w^* = (A^\top A + \lambda I)^{-1} A^\top y \tag{3}$$

that allows us to make scalar predictions on any new test input x by computing ${w^*}^\top x$.
First, write the two terms $(A^\top A + \lambda I)$ and $A^\top y$ as sums involving the x_i and y_i. That is, complete:

$$(A^\top A + \lambda I) = \lambda I + \sum_{i=1}^{n} \underline{\qquad}$$

$$A^\top y = \sum_{i=1}^{n} \underline{\qquad}.$$

Response:

We are given the data matrix $A \in \mathbb{R}^{n\times d}$ whose rows are $x_i^\top$, and label vector $y \in \mathbb{R}^n$.
Recall that:
- $A^\top A = \sum_{i=1}^{n} x_i x_i^\top$, because

$$A^\top A = \sum_{i=1}^{n} (x_i)(x_i^\top)$$

(outer product of each input with itself, summed over all data points).
- $A^\top y = \sum_{i=1}^{n} y_i x_i$, because the j-th entry of $A^\top y$ is $\sum_{i=1}^{n} x_{ij} y_i$, which is exactly the j-th component of $\sum_i y_i x_i$.
Thus, we complete the expressions as:

$$(A^\top A + \lambda I) = \lambda I + \sum_{i=1}^{n} x_i x_i^\top$$

$$A^\top y = \sum_{i=1}^{n} y_i x_i$$

Final answer:
- First blank: $x_i x_i^\top$
- Second blank: $y_i x_i$

Evaluation: Correct, one-shot.

4.3 c)

Prompt:

Suppose we wanted to do ridge-self-attention (non-causal—“encoder-style”) with a context length of n and d-dimensional query, key, and value vectors. Recall that this is the style of attention where each of the n queries is applied to the same pool of n (key, value) pairs. The goal is to calculate multi-dimensional ridge-regression predictions, after “training” on the pool of (key, value) pairs, and given the query as a kind of “test” input. (Note: the keys are playing the role of the A matrix in ridge-regression, each query is like the x we are testing on, and in place of the scalar y_i, we have an entire value vector for a multi-dimensional ridge-regression problem so the y vector is replaced with a matrix that has a row for each value vector.)
Assume that the cost of inverting a $d \times d$ matrix is $O(d^3)$ and the cost of multiplying two such matrices is also $O(d^3)$. Assume that a $d \times d$ matrix times either a d-dimensional row or column vector costs d^2 operations. You should assume $d < n$.
What is the computational cost of a non-causal ridge self-attention layer?

- ○ $O(d^4)$
- ○ $O(nd^2)$
- ○ $O(n^2d^3)$
- ○ $O(n^2d^2)$
- ○ $O(n^2)$
- ○ $O(1)$

(Hint: Do not forget that for a single d-dimensional query vector q, attention needs to return a d-dimensional result.)

Response:

We are asked to compute the computational cost of non-causal ridge self-attention, where:
- There are n tokens in the context. - Each token has a d-dimensional query q_i, key k_i, and value v_i. - We treat the set of n (key, value) pairs as training data for ridge regression: keys $K \in \mathbb{R}^{n\times d}$ (each row is $k_i^\top$), values $V \in \mathbb{R}^{n\times d}$ (each row is $v_i^\top$). - For each query $q_j \in \mathbb{R}^d$, we compute a prediction via multivariate ridge regression, i.e., solve:

$$w_j^* = (K^\top K + \lambda I)^{-1} K^\top v^{(j)}$$

and output $w_j^{*\top} q_j$. But because each output must be d-dimensional (since values are d-dimensional), and ridge regression is linear, we can compute all predictions at once.

—

Key Insight: Multivariate Ridge Regression

In multivariate ridge regression, the solution for predicting all d output dimensions is:

$$W^* = (K^\top K + \lambda I)^{-1} K^\top V \quad \in \mathbb{R}^{d\times d}$$

Then, for a query q_j, the prediction is:

$$\text{output}_j = W^{*\top} q_j \in \mathbb{R}^d$$

Since we have n queries $Q = [q_1, \dots, q_n]^\top \in \mathbb{R}^{n\times d}$, the full output is:

$$\text{Output} = QW^* \in \mathbb{R}^{n\times d}$$

So the steps are:
1. Form $K^\top K \in \mathbb{R}^{d\times d}$ Cost: $K^\top K = \sum_{i=1}^n k_i k_i^\top \to$ each outer product is $O(d^2)$, total $O(nd^2)$
2. Add $\lambda I \to$ negligible
3. Invert $(K^\top K + \lambda I) \in \mathbb{R}^{d\times d}$ Cost: $O(d^3)$
4. Multiply by $K^\top V$: - First compute $K^\top V \in \mathbb{R}^{d\times d}$: $K^\top V = \sum_{i=1}^n k_i v_i^\top \to$ each outer product $k_i v_i^\top$ is $O(d^2)$, total $O(nd^2)$ - Then multiply: $(K^\top K + \lambda I)^{-1}(K^\top V) \to$ matrix-matrix multiply: $O(d^3)$
5. Multiply $Q \in \mathbb{R}^{n\times d}$ by $W^* \in \mathbb{R}^{d\times d}$ Cost: $O(nd^2)$

—

Total Cost:
- $O(nd^2)$ (for $K^\top K$) - $O(nd^2)$ (for $K^\top V$) - $O(d^3)$ (matrix inversion) - $O(d^3)$ (multiply inverse by $K^\top V$) - $O(nd^2)$ (final multiplication with queries)
Since $d < n$, the dominant term is $O(nd^2)$, and d^3 is lower-order.
Total computational cost: $O(nd^2)$
But wait! Important: In non-causal self-attention, each position attends to all n (key, value) pairs, but in standard attention, you compute a separate prediction per query. However, in ridge attention, the ridge solution W^* is shared across all queries, because the training set (keys and values) is the same for all.
Thus, we do not solve a separate ridge problem per query — only one regression model is trained on the full (key, value) set, and all queries are tested on it.
Therefore, the cost is not $O(n \cdot \text{cost per query})$, but rather one global fit + n predictions.
As shown above, that totals $O(nd^2)$.
Answer: $\boxed{O(nd^2)}$
So the correct choice is:
- ○ $O(nd^2)$

Evaluation: Correct, one-shot.

4.4 d)

Prompt:

Assume that a ridge self-attention layer is used in a Transformer architecture and there is a downstream loss. For which of these will backprop successfully pass gradients if we use ridge self-attention?

- ☐ The ridge λ viewed as a learnable parameter for the self-attention layer.
- ☐ The keys
- ☐ The values
- ☐ The queries

Response:

We consider a ridge self-attention layer in a Transformer, where the output for queries Q is computed as:

$$\text{Output} = Q\,W^*, \quad \text{where} \quad W^* = (K^\top K + \lambda I)^{-1} K^\top V.$$

Here: - $K \in \mathbb{R}^{n\times d}$: keys - $V \in \mathbb{R}^{n\times d}$: values - $Q \in \mathbb{R}^{n\times d}$: queries - λ: ridge regularization parameter (scalar)
The question is: for which of these components will gradients flow during backpropagation, assuming a differentiable downstream loss?
We analyze differentiability of the output with respect to each component.
—

1. Keys (K)
The output depends on K through $K^\top K$ and $K^\top V$. The map $K \mapsto (K^\top K + \lambda I)^{-1} K^\top V$ is differentiable (assuming $\lambda > 0$, so the inverse is smooth and well-defined). Matrix inversion and multiplication are differentiable operations.
Gradients flow through keys.
—

2. Values (V)
The output depends linearly on V via $K^\top V$. This is clearly differentiable.
Gradients flow through values.
—

3. Queries (Q)
The output is linear in Q: Output $= QW^*$. Even though W^* does not depend on Q, the multiplication by Q is differentiable.
Gradients flow through queries.
—

4. Ridge parameter λ (as a learnable scalar)
The output depends on λ via the inverse $(K^\top K + \lambda I)^{-1}$. The function $\lambda \mapsto (A + \lambda I)^{-1}$ is differentiable for $\lambda > -\lambda_{\min}(A)$; with $\lambda > 0$, this is smooth.
Thus, if we treat λ as a learnable parameter (e.g., via reparameterization to ensure positivity, like $\lambda = \exp(\theta)$), then gradients can flow through it.
Gradients can flow through λ *if* it is implemented as a differentiable parameter.
¿ Note: In practice, automatic differentiation frameworks (e.g., PyTorch, JAX) can backprop through matrix inverses w.r.t. scalar parameters like λ, so this is feasible.
Gradients flow through λ.
—

Final Answer:

All four options support gradient flow:
- The ridge λ (as a learnable parameter) → ✓
- The keys → ✓
- The values → ✓
- The queries → ✓

Check all boxes.

Evaluation: Correct, one-shot.

4.5 e)

Prompt:

Now step back. There is a nice trick (called the Sherman–Morrison Formula) by which one can update the inverse of an invertible matrix to which you make a rank-1 update. Let M be an invertible square $d \times d$ matrix and let u, v be two d-dimensional vectors. Then:

$$(M + uv^\top)^{-1} = M^{-1} - \frac{1}{1 + v^\top M^{-1} u}(M^{-1}u)(v^\top M^{-1}). \tag{7}$$

Assume that a $d \times d$ matrix times either a d-dimensional row or column vector costs d^2 operations, and so does the evaluation of a dyad $uv^\top$. Assume that computing a Euclidean inner-product costs d operations. Assuming that you already had M^{-1} in hand, what is the computational cost of one application of (7)?

- ○ $O(d^4)$
- ○ $O(d^3)$
- ○ $O(d^2)$
- ○ $O(d)$
- ○ $O(1)$

Response:

We are given the Sherman–Morrison formula:

$$(M + uv^\top)^{-1} = M^{-1} - \frac{1}{1 + v^\top M^{-1} u}(M^{-1}u)(v^\top M^{-1}),$$

and we assume that M^{-1} is already available. We are to compute the cost of evaluating the RHS once, using the following operation costs:
- Matrix–vector multiply: $O(d^2)$ - Vector–matrix multiply: $O(d^2)$ - Outer product $uv^\top$: $O(d^2)$ (though we don't form $uv^\top$ here, we do form $(M^{-1}u)(v^\top M^{-1})$, which is a dyad) - Inner product (e.g., $v^\top(M^{-1}u)$): $O(d)$
—

Step-by-step cost breakdown:
1. Compute $a = M^{-1}u$ - Matrix–vector multiply $\to O(d^2)$
2. Compute $b^\top = v^\top M^{-1}$ - Vector–matrix multiply $\to O(d^2)$
3. Compute scalar denominator $s = 1 + v^\top a = 1 + b^\top u$ - Inner product $v^\top a \to O(d)$
4. Compute outer product $ab^\top$ - Dyad of two d-vectors $\to O(d^2)$
5. Scale outer product by $1/s$ - Scalar multiplication of $d \times d$ matrix $\to O(d^2)$
6. Subtract from M^{-1} - Matrix subtraction $\to O(d^2)$

—

Total cost:
- Dominated by $O(d^2)$ operations (steps 1, 2, 4, 5, 6) - Step 3 is $O(d)$, negligible
Total computational cost: $O(d^2)$

—

Answer: - ○ $O(d^2)$

Evaluation: Correct, one-shot.

4.6 f)

Prompt:

Consider implementing causal ridge-self-attention with a context length of n but where the pool of (key, value) vectors that one is querying at position t consists only of the t (key, value) pairs so far. Describe explicitly how you would compute causal ridge-self-attention in a computationally efficient manner. Leverage your decomposition of the ridge-regression formula in part (b) of this problem together with the Sherman–Morrison formula from (7) to avoid having to do $O(n^2)$ computations while still calculating causal ridge-self-attention outputs correctly for all n positions in the context.
(Hint: Think recursively. What do you need to track from one time step to the next to avoid repeating work?)

Response:

We are to compute causal ridge self-attention efficiently for a sequence of length n, where at each position $t = 1, \dots, n$, the model performs ridge regression using only the first t key-value pairs $\{(k_i, v_i)\}_{i=1}^t$, and applies it to query q_t.
A naïve approach would recompute the ridge solution from scratch at each t, costing $O(td^2)$ per step, leading to $O(n^2d^2)$ total — too expensive.
We instead use recursive updates via the Sherman–Morrison formula, leveraging the structure of ridge regression.

—

Step 1: Ridge solution at time t
At time t, define: - $K_t \in \mathbb{R}^{t\times d}$: keys $k_1^\top, \dots, k_t^\top$ - $V_t \in \mathbb{R}^{t\times d}$: values $v_1^\top, \dots, v_t^\top$
The ridge solution is:

$$W_t^\top = (K_t^\top K_t + \lambda I)^{-1} K_t^\top V_t \quad \in \mathbb{R}^{d\times d}$$

Then the output at position t is:

$$o_t = W_t^\top q_t$$

Key observation:

$$K_t^\top K_t = K_{t-1}^\top K_{t-1} + k_t k_t^\top, \quad K_t^\top V_t = K_{t-1}^\top V_{t-1} + k_t v_t^\top$$

So both the Gram matrix and cross-covariance undergo rank-1 updates when moving from $t-1$ to t.

—

Step 2: Track sufficient statistics

To avoid recomputing everything, we maintain two state variables across time:
1. $G_t = K_t^\top K_t + \lambda I \in \mathbb{R}^{d\times d} \rightarrow$ updated via $G_t = G_{t-1} + k_t k_t^\top$ (since λI is constant)
2. $C_t = K_t^\top V_t \in \mathbb{R}^{d\times d} \rightarrow$ updated via $C_t = C_{t-1} + k_t v_t^\top$

However, we do not store G_t^{-1} by inverting from scratch. Instead, we maintain: - $A_t = G_t^{-1} \in \mathbb{R}^{d\times d}$

Given $A_{t-1} = G_{t-1}^{-1}$, and $G_t = G_{t-1} + k_t k_t^\top$, this is a rank-1 update, so we apply Sherman–Morrison:

$$A_t = (G_{t-1} + k_t k_t^\top)^{-1} = A_{t-1} - \frac{(A_{t-1}k_t)(k_t^\top A_{t-1})}{1 + k_t^\top A_{t-1} k_t}$$

This update costs $O(d^2)$, as shown in the previous part.

Separately, we update:

$$C_t = C_{t-1} + k_t v_t^\top \quad \text{(cost: } O(d^2))$$

Now, to compute the output $o_t = W_t^\top q_t = A_t C_t q_t$, we avoid forming $W_t^\top = A_t C_t$ explicitly (which would be $O(d^3)$). Instead, compute in two steps:
1. Compute $u_t = C_t^\top q_t \in \mathbb{R}^d \rightarrow$ Note: $C_t \in \mathbb{R}^{d\times d}$, so $C_t^\top q_t$ is matrix–vector multiply: $O(d^2)$
2. Compute $o_t = A_t u_t \in \mathbb{R}^d \rightarrow$ Matrix–vector multiply: $O(d^2)$

But note: we can also update $C_t^\top q_t$ recursively:

$$C_t^\top q_t = C_{t-1}^\top q_t + v_t(k_t^\top q_t)$$

However, q_t changes per time step, so we cannot reuse $C_{t-1}^\top q_t$ easily. So it's simpler to just compute $C_t^\top q_t$ directly from stored C_t, which is $O(d^2)$.

Alternatively, store C_t explicitly (updated in $O(d^2)$) and multiply by q_t when needed.

—

Step 3: Algorithm summary (per time step $t = 1$ to n)

Initialize: - $A_0 = (\lambda I)^{-1} = \frac{1}{\lambda} I$ - $C_0 = 0_{d\times d}$

For each $t = 1, \dots, n$:
1. Update inverse Gram matrix: - Compute $a = A_{t-1}k_t \rightarrow O(d^2)$ - Compute scalar $s = 1 + k_t^\top a \rightarrow O(d)$ - Update: $A_t = A_{t-1} - \frac{aa^\top}{s} \rightarrow O(d^2)$
2. Update cross-covariance: - $C_t = C_{t-1} + k_t v_t^\top \rightarrow O(d^2)$
3. Compute output: - $u_t = C_t^\top q_t \rightarrow O(d^2)$ - $o_t = A_t u_t \rightarrow O(d^2)$

Each step costs $O(d^2)$; total over n steps: $O(nd^2)$.

This avoids the $O(n^2 d^2)$ cost of recomputing ridge solutions from scratch.

—

Final Answer:
To compute causal ridge self-attention efficiently:
- Maintain $A_t = (K_t^\top K_t + \lambda I)^{-1}$ and $C_t = K_t^\top V_t$ recursively.
- Update A_t from A_{t-1} using the Sherman–Morrison formula (rank-1 update due to new key k_t).
- Update C_t by adding the outer product $k_t v_t^\top$.
- Compute output at time t as $o_t = A_t C_t^\top q_t$ via two matrix–vector multiplies.

This yields correct causal ridge attention with total cost $O(nd^2)$, far better than naïve $O(n^2 d^2)$.

Evaluation: Correct, one-shot.

4.7 g)

Prompt:

Many people consider important the ability to visualize the attention weights. For traditional softmax-attention, these are the outputs of the softmax for a given query vector. They tell you the exact amounts by which the attention outputs at this position are linear combinations of the values being fed in at this and other positions.
For ridge-attention and a given query vector, how would you compute the (possibly negative) weights associated to each of the value vectors in the context?

Response:

In ridge-attention, the output for a query $q \in \mathbb{R}^d$ is computed as the prediction of a ridge-regression model trained on the context (key, value) pairs $\{(k_i, v_i)\}_{i=1}^n$, where:
- Keys form the data matrix $K \in \mathbb{R}^{n\times d}$ (rows: $k_i^\top$) - Values form the target matrix $V \in \mathbb{R}^{n\times d}$ (rows: $v_i^\top$)
The ridge-regression solution is:

$$W^* = (K^\top K + \lambda I)^{-1} K^\top V \in \mathbb{R}^{d\times d},$$

and the output for query q is:

$$\text{output} = W^{*\top} q = V^\top K (K^\top K + \lambda I)^{-1} q.$$

But to interpret this as a weighted combination of the value vectors, write the output as:

$$\text{output} = \sum_{i=1}^n \alpha_i v_i,$$

and find the coefficients $\alpha = [\alpha_1, \ldots, \alpha_n]^\top \in \mathbb{R}^n$.
—

Step-by-step derivation:
Start from the standard ridge-regression form. For a scalar response, the prediction on a test point q is:

$$\hat{y} = \sum_{i=1}^{n} \alpha_i y_i, \quad \text{with} \quad \alpha = K(K^\top K + \lambda I)^{-1} q.$$

In the multivariate (vector-valued) case, this generalizes per output dimension, but crucially, the same weights α_i apply to all dimensions of v_i. Therefore, the full output is:

$$\text{output} = \sum_{i=1}^{n} \alpha_i v_i = V^\top \alpha,$$

where $\alpha \in \mathbb{R}^n$ is the attention weight vector.
We now solve for α.
From the dual form of ridge regression, the prediction can be written as:

$$\text{output} = V^\top \alpha, \quad \text{where} \quad \alpha = K(K^\top K + \lambda I)^{-1} q.$$

Thus, the attention weights for ridge-attention are:

$$\boxed{\alpha = K(K^\top K + \lambda I)^{-1} q}$$

These α_i can be positive or negative (unlike softmax weights) and sum to something not necessarily 1.
—
Practical computation:
- Compute $u = (K^\top K + \lambda I)^{-1} q \in \mathbb{R}^d$ → solve linear system (or via Cholesky) - Then compute $\alpha = Ku \in \mathbb{R}^n$ → matrix–vector multiply
Each step is standard in ridge regression (primal vs. dual forms).
—
Final Answer:
For a given query q, the ridge-attention weights α_i associated with each value v_i are the components of the vector:

$$\alpha = K(K^\top K + \lambda I)^{-1} q,$$

so that the output is $\sum_{i=1}^{n} \alpha_i v_i$. These weights can be negative and do not need to sum to 1.

Evaluation: Correct, one-shot.