

(initial  
prompt,  
asking if  
DeepSeek can  
follow strict  
no hallucination  
/ full reasoning  
constraints)

You are an AI assistant helping me think through conceptual, mathematical, and analytical questions from my deep learning homework. Important rules: • You must attempt each question using your own reasoning. • Show your full reasoning clearly. • If you are uncertain, explicitly state your uncertainty. • If you do not know something, do not invent or hallucinate facts. Say so honestly. • Do not rely on external resources or unstated theorems. • Keep variable names exactly as written in the question. • Do not change definitions unless asked. My goal is to evaluate how well you can solve non-coding deep learning questions without being guided. Do not ask me to solve steps or supply missing equations — attempt the solution yourself even if it might be wrong. Confirm you understand these rules, then wait for my first question.

I understand the rules clearly:

- I must attempt each question using my own reasoning.
- I will show my full reasoning clearly.
- If uncertain, I will explicitly state my uncertainty.
- If I don't know something, I will say so honestly.
- I will not rely on external resources or unstated theorems.
- I will keep variable names exactly as written.
- I will not change definitions unless asked.
- I will not ask you to solve steps or supply missing equations.

I'm ready for your first question. Please proceed.



hw8 (1).pdf

PDF 225.65KB

please solve all parts of problem 1 from this homework. look to one shot it

I'll solve problem 1 from the homework. Let me go through each part systematically.

## Problem 1: SSM Convolution Kernel

### (a) Convolution Kernel and the Output Equation

We have the state-space model:

$$x_{k+1} = Ax_k + Bu_k, \quad y_k = Cx_k + Du_k$$

with  $x_0 = 0$ .

We need to express  $y_k$  as a convolution:

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

where  $u_{\leq 0} = 0$  (zero-padding).

Let me compute  $x_k$  recursively:

- $x_0 = 0$
- $x_1 = Ax_0 + Bu_0 = Bu_0$
- $x_2 = Ax_1 + Bu_1 = ABu_0 + Bu_1$
- $x_3 = Ax_2 + Bu_2 = A^2Bu_0 + ABu_1 + Bu_2$

By induction:

$$x_k = \sum_{\ell=0}^{k-1} A^{k-1-\ell} Bu_{\ell}$$

for  $k \geq 1$ , and  $x_0 = 0$ .

Now substitute into  $y_k = Cx_k + Du_k$ :

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

Let me re-index: set  $j = k - 1 - \ell$ , so  $\ell = k - 1 - j$ . Then:

$$y_k = \sum_{j=0}^{k-1} CA^j Bu_{k-1-j} + Du_k$$

Alternatively, write as:

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

where  $m = j + 1$ .

Let me define the kernel  $K_\ell$ :

- For  $\ell = 0$ : coefficient of  $u_k$  is  $D$ . So  $K_0 = D$ .
- For  $\ell \geq 1$ : coefficient of  $u_{k-\ell}$  is  $CA^{\ell-1}B$ . So  $K_\ell = CA^{\ell-1}B$  for  $\ell \geq 1$ .

But note: in the convolution sum  $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$ , when  $k - \ell < 0$ ,  $u_{k-\ell} = 0$ . So effectively for each  $k$ , the sum runs up to  $\ell = k$ .

Thus the kernel  $K$  is:

$$K = \{K_\ell\}_{\ell=0}^L$$

with

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

So:

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

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

Correct answer,  
reasonable work,  
1 shot!

## (b) Concrete Examples

### i. Scalar Case

Given  $n = 1$ ,  $A = \alpha = 0.8$ ,  $B = \beta = 1$ ,  $C = \gamma = 1.5$ ,  $D = \delta$ . The problem doesn't specify  $\delta$ , but I'll assume  $\delta = 0$  unless stated otherwise (since in the 2D case they set  $D = 0$ ). Wait, they gave  $\delta$  but didn't specify a value. Actually, they said "set  $A = \alpha$ ,  $B = \beta$ ,  $C = \gamma$ ,  $D = \delta$ " and then "use  $\alpha = 0.8$ ,  $\beta = 1$ ,  $\gamma = 1.5$ " but didn't give  $\delta$ . I'll check: In the scalar case, they might want a general expression, but they ask to "compute the kernel up to  $L = 4$ ". I'll assume  $\delta = 0$  for simplicity (like in the 2D case). Actually, they might want us to keep  $\delta$  as a parameter. Let me look: They wrote "set  $A = \alpha$ ,  $B = \beta$ ,  $C = \gamma$ ,  $D = \delta$ " and then "use  $\alpha = 0.8$ ,  $\beta = 1$ ,  $\gamma = 1.5$ ". They didn't specify  $\delta$ , so perhaps  $\delta$  is also a parameter? I'll check part (a) formula:  $K_0 = D = \delta$ ,  $K_\ell = CA^{\ell-1}B = \gamma\alpha^{\ell-1}\beta$ . So:

- $K_0 = \delta$
- $K_1 = \gamma\alpha^0\beta = 1.5 \times 1 \times 1 = 1.5$
- $K_2 = \gamma\alpha^1\beta = 1.5 \times 0.8 \times 1 = 1.2$
- $K_3 = \gamma\alpha^2\beta = 1.5 \times 0.64 \times 1 = 0.96$
- $K_4 = \gamma\alpha^3\beta = 1.5 \times 0.512 \times 1 = 0.768$

Thus kernel up to  $L = 4$ :  $K = [\delta, 1.5, 1.2, 0.96, 0.768]$ .

correct answer (although stopped one power too early), reasonable work, I did

But since  $\delta$  isn't given, maybe they expect  $\delta = 0$ . In many SSMs,  $D = 0$ . I'll note both.

## ii. 2D Case

Given:

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

Compute kernels up to  $L = 3$ :

- $K_0 = D = 0$  (scalar since output is scalar? Wait,  $C$  is  $1 \times 2$ ,  $B$  is  $2 \times 1$ , so  $K_\ell$  is scalar.)
- $K_1 = CA^0B = CB = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1$

- $K_2 = CAB$ : first compute  $AB = \begin{pmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.7 \\ 0.2 \end{pmatrix}$ , then  $C(AB) = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0.7 \\ 0.2 \end{pmatrix} = 0.7$
- $K_3 = CA^2B$ : compute  $A^2 = A \cdot A = \begin{pmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{pmatrix}^2 = \begin{pmatrix} 0.49 + 0.02 & 0.07 + 0.06 \\ 0.14 + 0.12 & 0.02 + 0.36 \end{pmatrix} = \begin{pmatrix} 0.51 & 0.13 \\ 0.26 & 0.38 \end{pmatrix}$ . Then  $A^2B = \begin{pmatrix} 0.51 & 0.13 \\ 0.26 & 0.38 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.51 \\ 0.26 \end{pmatrix}$ . Then  $C(A^2B) = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0.51 \\ 0.26 \end{pmatrix} = 0.51$

So kernel:  $K = [0, 1, 0.7, 0.51]$ .

Correct answer,  
reasonable mark,  
1 shot

**Discussion:** The kernel captures the "impulse response" because if we input  $u_0 = 1$  and  $u_k = 0$  for  $k > 0$  (a discrete impulse at time 0), then the output  $y_k = K_k$  (since convolution with impulse gives the kernel). So  $K_\ell$  is the output at time  $\ell$  when an impulse is applied at time 0. This is exactly the impulse response of the linear system.

## (c) Efficient Computation with Convolutions

If we know kernel  $K$  of length  $L + 1$ , and we have input sequence  $\{u_k\}$  of length  $L$  (or  $L + 1$ ? The problem says length  $L$ , but earlier they said input:  $(u_0, \dots, u_L)$  which is length  $L + 1$ . Let's assume input length  $N = L + 1$ . Actually careful: "input sequence  $\{u_k\} \in \mathbb{R}^d$  of length  $L$ " — so length  $L$  means  $L$  elements? Possibly they mean  $L$  time steps from 0 to  $L - 1$ . I'll treat length as  $L$  samples.

The convolution  $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$  (with zero-padding for negative indices) is a 1D convolution. We can compute it using FFT-based convolution, which parallelizes across the entire sequence.

**Parallelization:** With FFT, we can compute the convolution in  $O(L \log L)$  time with high parallelism. The critical path length is  $O(\log L)$  for the FFT steps (assuming enough parallel processors).

**Naive direct computation:** From the unrolled recursion  $x_{k+1} = Ax_k + Bu_k$ , we have a sequential dependency: to compute  $x_{k+1}$  we need  $x_k$ . So the critical path

length is  $O(L)$  (sequential steps). We cannot parallelize across time steps because of recurrence.

Thus convolution approach reduces critical path from  $O(L)$  to  $O(\log L)$ .

incorrect

will merge in the right direction

### (d) Efficient Kernel Computation

Given  $A, B, C$ , we need to compute  $K_\ell = CA^{\ell-1}B$  for  $\ell = 1, \dots, L$ , and  $K_0 = D$ .

A naive way is to compute powers of  $A$  sequentially: compute  $A, A^2, A^3, \dots, A^{L-1}$ , each requiring matrix multiplication  $O(n^3)$  per power, total  $O(Ln^3)$ .

But we can parallelize using exponentiation by squaring (binary exponentiation).

Since  $L = 2^N$ , we can compute  $A^{2^k}$  for  $k = 0, 1, \dots, N-1$  by repeated squaring:  $A^{2^{k+1}} = (A^{2^k})^2$ . This requires  $N = \log_2 L$  sequential steps, but each squaring can be parallelized internally (matrix multiplication parallelizable). Then to get any  $A^{\ell-1}$ , we multiply the appropriate powers corresponding to binary representation of  $\ell-1$ . Since we need all  $\ell$  from 1 to  $L$ , we can compute all needed powers in parallel after computing the  $A^{2^k}$  matrices.

Alternatively, we can compute the kernel via convolution in the frequency domain? Not directly, because kernel is defined via matrix powers.

Another approach: The kernel is the impulse response, we could simulate the system with an impulse input  $u_0 = 1, u_k = 0$  for  $k > 0$  and compute outputs — but that's sequential.

Given  $L = 2^N$ , a divide-and-conquer method: To compute  $K_1, \dots, K_{2^N}$ , note that:

- First half:  $K_1, \dots, K_{2^{N-1}}$  are  $CA^jB$  for  $j = 0, \dots, 2^{N-1} - 1$ .
- Second half:  $K_{2^{N-1}+1}, \dots, K_{2^N}$  are  $CA^jB$  for  $j = 2^{N-1}, \dots, 2^N - 1$ . But  $A^j = A^{2^{N-1}} \cdot A^{j-2^{N-1}}$ . So if we compute  $M = A^{2^{N-1}}$ , then second half kernel can be computed as  $CM A^{j'}B$  where  $j' = 0, \dots, 2^{N-1} - 1$ . This suggests recursion: compute kernel for length  $2^{N-1}$ , compute  $M$ , then compute second half as  $(CM) \cdot (\text{kernel for length } 2^{N-1})$  but careful: kernel entries are scalars, but  $CM$

is a row vector, and kernel for first half are scalars  $CA^{j'}B$ . Actually,  $CA^{j'}B = (CM)(A^{j'}B)$ , not simply  $(CM) \times (\text{scalar})$ . So maybe not straightforward.

Better: Use parallel prefix scan type approach? Possibly compute all powers  $A^j$  for  $j = 0, \dots, L-1$  using parallel prefix. Given  $A$ , we want  $A, A^2, A^3, \dots, A^{L-1}$ . This can be done with  $O(\log L)$  parallel steps using a binary tree: compute pairwise products, etc.

But the problem asks for strategies to parallelize kernel computation. One way: Compute  $A^p$  for  $p = 1, 2, 4, 8, \dots, 2^{N-1}$  via repeated squaring (sequential steps  $N$ ). Then for each  $\ell$ , compute  $A^{\ell-1}$  by multiplying the appropriate subset of these matrices (binary representation). The multiplications for different  $\ell$  can be done in parallel across  $\ell$ . So total parallel time is  $O(N \cdot \text{matrix multiplication time})$  assuming unlimited processors.

the idea is essentially the same as the solution (both use binary exponentiation & parallel instructions)

### (e) Adding structure to $A$ (diagonal)

If  $A$  is diagonal, say  $A = \text{diag}(a_1, \dots, a_n)$ , then  $A^\ell = \text{diag}(a_1^\ell, \dots, a_n^\ell)$ . Then  $K_\ell = CA^{\ell-1}B$  becomes:

Let  $B$  be column vector  $[b_1, \dots, b_n]^T$ ,  $C$  be row vector  $[c_1, \dots, c_n]$ . Then

$$K_\ell = \sum_{i=1}^n c_i a_i^{\ell-1} b_i = \sum_{i=1}^n (c_i b_i) a_i^{\ell-1}$$

So each  $K_\ell$  is a weighted sum of exponentials  $a_i^{\ell-1}$ . We can compute all  $K_\ell$  for  $\ell = 1, \dots, L$  in parallel by computing each term  $(c_i b_i) a_i^{\ell-1}$  in parallel across  $i$  and  $\ell$ . This is  $O(nL)$  work but can be parallelized to  $O(\log n + \log L)$  depth using parallel prefix for powers.

Alternatively, we can compute  $K_\ell$  via convolution of vectors? Actually, note that  $K_\ell$  is like an inner product between  $C$  and  $A^{\ell-1}B$ . Since  $A^{\ell-1}B$  is elementwise  $b_i a_i^{\ell-1}$ , we can compute all  $\ell$  for each  $i$  by computing geometric sequences  $a_i^\ell$ . Computing geometric sequence for each  $a_i$  can be done in parallel across  $i$ , and for each  $i$ , we can compute powers via repeated squaring or parallel prefix. Overall parallel time  $O(\log L)$ .

correct & short!

## (f) Diagonal-plus-low-rank (DPLR) structure

Given  $A = I_n + pp^\top$ . We want efficient kernel computation.

Note that  $A^\ell = (I + pp^\top)^\ell$ . This is a rank-1 update to identity. There is a formula for powers: By binomial expansion, since  $(pp^\top)^k = (p^\top p)^{k-1} pp^\top = \|p\|^{2(k-1)} pp^\top$  for  $k \geq 1$ . Actually, check:  $(pp^\top)^2 = p(p^\top p)p^\top = \|p\|^2 pp^\top$ . Similarly,  $(pp^\top)^k = \|p\|^{2(k-1)} pp^\top$ . So

$$A^\ell = (I + pp^\top)^\ell = \sum_{k=0}^{\ell} \binom{\ell}{k} (pp^\top)^k$$

But  $(pp^\top)^0 = I$  (by convention for  $k=0$ ?). Actually, for  $k=0$ , term is  $I$ . For  $k \geq 1$ ,  $(pp^\top)^k = \|p\|^{2(k-1)} pp^\top$ . So

$$A^\ell = I + \left( \sum_{k=1}^{\ell} \binom{\ell}{k} \|p\|^{2(k-1)} \right) pp^\top$$

Correct answer,  
reasonable mark,  
(shot)

Let  $\alpha_\ell = \sum_{k=1}^{\ell} \binom{\ell}{k} \|p\|^{2(k-1)}$ . Simplify: note that  $\sum_{k=0}^{\ell} \binom{\ell}{k} \|p\|^{2k} = (1 + \|p\|^2)^\ell$ . Our sum starts at  $k=1$  and has exponent  $2(k-1)$ . Write  $\|p\|^{2(k-1)} = \|p\|^{-2} \|p\|^{2k}$  for  $\|p\| \neq 0$ . Then

$$\alpha_\ell = \|p\|^{-2} \sum_{k=1}^{\ell} \binom{\ell}{k} \|p\|^{2k} = \|p\|^{-2} ((1 + \|p\|^2)^\ell - 1)$$

Thus

$$A^\ell = I + \frac{(1 + \|p\|^2)^\ell - 1}{\|p\|^2} pp^\top$$

for  $\|p\| \neq 0$ . If  $\|p\| = 0$ , then  $A = I$  and  $A^\ell = I$ .

Now compute  $K_\ell = CA^{\ell-1}B$ . Let  $\sigma = \|p\|^2$ . Then

$$A^{\ell-1} = I + \frac{(1 + \sigma)^{\ell-1} - 1}{\sigma} pp^\top$$

Thus

everything else  
is extra work



$$K_\ell = CB + \frac{(1 + \sigma)^{\ell-1} - 1}{\sigma} Cpp^\top B$$

Let  $c_b = CB$ , and  $c_p = Cp$ ,  $p_b = p^\top B$ . Then

$$K_\ell = c_b + \frac{(1 + \sigma)^{\ell-1} - 1}{\sigma} (c_p \cdot p_b)$$

Note  $c_p \cdot p_b$  is scalar. So kernel is of the form  $K_\ell = \alpha + \beta \cdot \gamma^{\ell-1}$  where  $\gamma = 1 + \sigma$ . So we can compute all  $K_\ell$  in parallel in  $O(\log L)$  time by computing powers of  $\gamma$  via repeated squaring.

Thus leveraging DPLR structure reduces computation to scalar exponential.

### Final answers summary:

(a)  $K_0 = D$ ,  $K_\ell = CA^{\ell-1}B$  for  $\ell \geq 1$ .

(b) i.  $K = [\delta, 1.5, 1.2, 0.96, 0.768]$  (with  $\delta$  unspecified).

ii.  $K = [0, 1, 0.7, 0.51]$ . Kernel is impulse response.

(c) Convolution parallelizes via FFT: critical path  $O(\log L)$ . Naive recurrence: critical path  $O(L)$ .

(d) Compute powers  $A^{2^k}$  via repeated squaring, then compute each  $A^{\ell-1}$  using binary representation in parallel across  $\ell$ .

(e) Diagonal  $A$ :  $K_\ell = \sum_i c_i b_i a_i^{\ell-1}$ . Compute geometric sequences in parallel.

(f) DPLR:  $K_\ell = CB + \frac{(1 + \|p\|^2)^{\ell-1} - 1}{\|p\|^2} (Cp)(p^\top B)$ . Compute powers of  $(1 + \|p\|^2)$  via repeated squaring.

I have solved all parts of problem 1.

Wedge #1:  
I am trying to  
nudge Deepseek in  
the right direction  
for (c)

You gave the critical path as  $O(\log L)$ , but I think you might be ignoring the cost of the matrix multiplications inside the convolution. Could you re-evaluate the critical path assuming that multiplying two  $n \times n$  matrices in parallel itself has non-trivial depth?

You're right — I need to consider the computational model more carefully. The critical path length depends on the operations we can parallelize and their inherent depth. Let's analyze properly.

## Re-evaluating (c) and (d) with matrix multiplication costs

### (c) Efficient Computation with Convolutions

We have two ways to compute the output sequence  $\{y_k\}$ :

#### 1. Direct recurrent computation:

$$x_{k+1} = Ax_k + Bu_k, \quad y_k = Cx_k + Du_k$$

Each step requires one matrix-vector multiply  $Ax_k$  ( $O(n^2)$  ops) and vector addition. The steps are sequential: we need  $x_k$  to compute  $x_{k+1}$ . So the critical path length is  $O(L)$  steps, each step being matrix-vector multiply (depth  $O(\log n)$  if we parallelize the dot products? Actually, matrix-vector multiply for  $n \times n$  matrix and  $n$ -vector can be done in parallel: each output element is a dot product of length  $n$ , which can be done in  $O(\log n)$  depth with parallel reduction. Then adding vectors is  $O(1)$  depth. So each step depth is  $O(\log n)$ . Total depth =  $O(L \log n)$ .

But if we consider the most parallel possible implementation, we could compute each output element  $y_k$  directly via the convolution formula once kernel is known — that's different.

#### 2. Convolution approach:

If we already know the kernel  $K$  (scalar if output is scalar, else matrix), we compute  $y_k = \sum_{\ell=0}^L K_{\ell} u_{k-\ell}$ .

This is a 1D convolution of length  $L + 1$  kernel with length  $L$  input (or length  $N$  input, but let's assume sequence length  $L$  for simplicity).

Using FFT-based convolution:

- Compute FFT of kernel and input (each size  $\sim 2L$ ), each FFT has depth  $O(\log L)$  if parallelized using butterfly network.
- Pointwise multiply in frequency domain: depth  $O(1)$  per element, but we need to multiply two vectors of length  $O(L)$ , which can be done in parallel across all

elements, so depth  $O(1)$ .

- Inverse FFT: depth  $O(\log L)$ .

So total depth  $O(\log L)$ . But note: the kernel  $K_\ell$  might be matrices if  $u_k$  is vector and  $y_k$  is vector? Wait, in problem setup,  $u_k$  is scalar? Actually not specified.

Could be vector input/output. The convolution formula given is scalar convolution:

$y_k = \sum_\ell K_\ell u_{k-\ell}$ , which suggests  $u_k$  is scalar,  $y_k$  is scalar, and  $K_\ell$  is scalar. But earlier they said input sequence  $\{u_k\} \in \mathbb{R}^d$  of length  $L$  — that implies  $u_k$  is  $d$ -dimensional vector. Then the convolution becomes multi-channel: each output dimension is sum over input channels convolved with a kernel per channel. But they might be assuming single-channel for simplicity in part (c). I'll assume  $u_k$  scalar,  $y_k$  scalar for now.

However, if  $u_k \in \mathbb{R}^d$  and  $y_k \in \mathbb{R}^{d_y}$ , then  $K_\ell$  is matrix  $d_y \times d$ . Then convolution is a matrix-vector multiply for each  $\ell$ , and sum over  $\ell$ . Still parallelizable via FFT if we treat each entry separately.

The critical path length using FFT is  $O(\log L)$  for the FFT steps, independent of  $n$  (state dimension). But computing the kernel itself (part d) might have deeper cost.

So for (c), assuming kernel already known, the convolution can be done with depth  $O(\log L)$  if we use FFT and parallel arithmetic.

**But** the question might be about parallelizing computation of  $y_k$  from recurrence vs convolution *including kernel computation*. They ask: "If we already know the kernel  $K$ , how much can we parallelize the computation of the output sequence... What is the minimum critical path length of the computation? What about a naive, direct computation of  $y_k$  from the unrolled recursion?"

So they separate kernel computation (part d) from output computation (part c). Given kernel, output computation is convolution — depth  $O(\log L)$  using FFT. Naive direct computation depth is  $O(L \log n)$  (since each step is matrix-vector multiply with depth  $O(\log n)$ ). So convolution reduces depth from  $O(L)$  to  $O(\log L)$  (ignoring log factors of  $n$ ).

still not completely right,  
will merge again

## (d) Efficient Kernel Computation

We need to compute  $K_\ell = CA^{\ell-1}B$  for  $\ell = 1, \dots, L$ . Assume  $A \in \mathbb{R}^{n \times n}$ ,  $B \in \mathbb{R}^{n \times d_u}$ ,  $C \in \mathbb{R}^{d_y \times n}$ . Then  $K_\ell$  is  $d_y \times d_u$  matrix.

We need to compute  $A^{\ell-1}$  for  $\ell = 1, \dots, L$ . That's  $L$  matrix powers. If we compute sequentially:  $A^1, A^2, \dots, A^{L-1}$ , each power requires multiplying by  $A$ , which is  $O(n^3)$  ops if done naively, but depth of a single matrix multiplication depends on parallel algorithm.

**Parallel matrix multiplication depth:** The best parallel matrix multiplication algorithms (e.g., Strassen-like parallel, or Cannon's algorithm on 2D mesh) can achieve depth  $O(\log n)$  for  $n \times n$  matrices if we have enough processors. But in theoretical parallel models (PRAM), matrix multiplication can be done in  $O(\log n)$  depth using  $O(n^3)$  processors. So assume depth per multiplication =  $O(\log n)$ .

Then sequential chain of  $L - 1$  multiplications has depth  $O(L \log n)$ .

We can do better using parallel prefix (binary tree):

We want  $A^1, A^2, \dots, A^{L-1}$ . This is a parallel prefix problem on the semigroup of matrix multiplication. We can compute all prefixes in depth  $O(\log L \cdot \text{depth of multiply}) = O(\log L \cdot \log n)$  using a parallel prefix circuit (like Ladner-Fischer). So depth  $O(\log L \log n)$ .

If  $L = 2^N$ , we can compute  $A^{2^k}$  for  $k = 0, \dots, N - 1$  by repeated squaring: depth  $O(\log L \cdot \log n)$  as well (since each squaring is a multiply, and we do  $\log L$  sequential squarings, but each squaring can be parallelized internally). Actually repeated squaring is sequential in steps: compute  $A^2, A^4, A^8, \dots$  each depends on previous, so depth  $O(\log L \cdot \log n)$  same as parallel prefix.

Thus the minimum critical path for kernel computation is  $O(\log L \log n)$ .

After we have  $A^{\ell-1}$ , we compute  $CA^{\ell-1}B$ . That's two matrix multiplications: first  $M_\ell = A^{\ell-1}B$  ( $n \times d_u$ ), then  $K_\ell = CM_\ell$  ( $d_y \times d_u$ ). We can compute all  $M_\ell$  in parallel across  $\ell$ , each multiplication depth  $O(\log n)$  (assuming  $d_u \ll n$ ), total depth  $O(\log n + \log L \log n)$ ? Wait, we already have  $A^{\ell-1}$ , computing  $A^{\ell-1}B$  for each  $\ell$  can be done in parallel across  $\ell$ , each with depth  $O(\log n)$ . Similarly for second multiply. So additional depth  $O(\log n)$  beyond computing  $A^{\ell-1}$ .

Thus total depth for kernel computation =  $O(\log L \log n)$ .

Now back to (c): If we already have kernel, output computation depth =  $O(\log L)$  via FFT. But if we include kernel computation, total depth =  $O(\log L \log n)$  for kernel +  $O(\log L)$  for convolution =  $O(\log L \log n)$ , which is still much less than  $O(L \log n)$  for naive recurrence.

So my earlier answer for (c) critical path  $O(\log L)$  was correct *given kernel already known*. If kernel not known, we need to add kernel computation depth.

### Revised answers for (c) and (d):


(c)

- Given kernel  $K$ , output computation via convolution can be parallelized using FFT with critical path length  $O(\log L)$  (assuming parallel arithmetic and ignoring constants).
- Naive direct computation from unrolled recursion has critical path length  $O(L \log n)$  (each step requires matrix-vector multiply with depth  $O(\log n)$ , and steps are sequential).

(d)

- Efficient kernel computation: Use parallel prefix to compute all powers  $A^{\ell-1}$  for  $\ell = 1, \dots, L$  in depth  $O(\log L \log n)$ . Then compute  $K_\ell = CA^{\ell-1}B$  in parallel across  $\ell$ , each with depth  $O(\log n)$ , total depth  $O(\log L \log n)$ .
- Alternative: Use repeated squaring to compute  $A^{2^k}$ , then compute each  $A^{\ell-1}$  via binary representation, parallel across  $\ell$ , same depth.

Does this address your concern?

None #2 for  


You updated the naive recurrence to include the  $\log n$  depth from matrix-vector multiplication, which is great. For the convolution case: doesn't each FFT stage also require parallel vector operations with depth  $O(\log n)$ ?

If there are  $\log L$  FFT stages, wouldn't that make the critical path  $O(\log L \cdot \log n)$  instead of just  $O(\log L)$ ?

You're absolutely right. I need to be consistent about the cost of vector operations. Let's carefully analyze the depth for each case under a parallel computation model where:

- Adding two vectors of length  $n$ : can be done in  $O(1)$  depth with  $n$  processors (element-wise parallel).
- Multiplying a scalar by a vector:  $O(1)$  depth.
- Dot product of two length- $n$  vectors: can be done in  $O(\log n)$  depth using parallel reduction.
- Matrix-vector multiplication (size  $n \times n$ ): each output element is a dot product of length  $n$ ; we can compute all  $n$  outputs in parallel, each dot product takes  $O(\log n)$  depth, so total depth  $O(\log n)$ .
- FFT of size  $N$ : each butterfly stage does  $N/2$  independent complex multiply-adds; each multiply-add is constant depth, but we have  $\log N$  sequential stages, so depth  $O(\log N)$  assuming each stage's operations are parallelized across all elements.

Now, we need to decide the "size" of vectors in the convolution case.

### Case 1: Scalar input/output ( $u_k, y_k$ scalars)

Then kernel  $K_\ell$  is scalar. Convolution is 1D convolution of scalar sequences. FFT operates on scalars; each butterfly operation is a scalar multiply-add, depth  $O(1)$  per stage. With  $\log L$  stages, depth  $O(\log L)$ . So critical path =  $O(\log L)$ .

But the problem says input sequence  $\{u_k\} \in \mathbb{R}^d$  of length  $L$  — that suggests vector input. Let's interpret:

### Case 2: Vector input $d_u$ -dimensional, vector output $d_y$ -dimensional

Then  $K_\ell$  is a matrix  $d_y \times d_u$ . The convolution becomes:

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

This is a multi-channel convolution. We can compute it as separate convolutions per input-output channel pair, then sum across input channels. That's  $d_y \times d_u$  separate scalar convolutions, then reductions.

Alternatively, we can treat it as a single convolution where each operation is a matrix-vector multiply. Using FFT:

We can FFT each channel separately (since FFT is linear). Let's consider the transform domain:

Let  $\tilde{U}_m$  be the FFT of input sequence across time for each input channel  $m$ . Similarly, we have  $\tilde{K}_{\ell,m}$  for each output channel  $n$  and input channel  $m$ . In frequency domain, convolution becomes pointwise matrix multiplication:

$$\tilde{Y}_n(\omega) = \sum_m \tilde{K}_{n,m}(\omega) \tilde{U}_m(\omega)$$

where  $\tilde{K}_{n,m}(\omega)$  is the FFT of kernel across time for that channel pair.

So in frequency domain, at each frequency bin we have a matrix-vector multiply:  $\tilde{Y}(\omega) = \tilde{K}(\omega)\tilde{U}(\omega)$ , where  $\tilde{K}(\omega)$  is  $d_y \times d_u$ ,  $\tilde{U}(\omega)$  is  $d_u \times 1$ , result  $d_y \times 1$ . This matrix-vector multiply has depth  $O(\log d_u)$  if we parallelize dot products across rows.

But we have  $F$  frequency bins (where  $F = O(L)$ ). We can process all bins in parallel. So depth for the pointwise multiply stage = depth of one matrix-vector multiply =  $O(\log \min(d_u, d_y))$ ? Actually, computing each output element needs dot product of length  $d_u$ ; we can compute all  $d_y$  outputs in parallel, each dot product takes  $O(\log d_u)$  depth. So depth per bin =  $O(\log d_u)$ . Since all bins are independent, depth remains  $O(\log d_u)$ .

Now, the FFT stage: each FFT of length  $L$  on a single channel has depth  $O(\log L)$  because each of the  $\log L$  stages consists of parallel butterflies. For  $d_u$  input channels, we can do all channels in parallel, so depth still  $O(\log L)$ . Same for inverse FFT on  $d_y$  channels.

Thus total depth for convolution = depth(FFT) + depth(pointwise multiply) + depth(IFFT) =  $O(\log L) + O(\log d_u) + O(\log L) = O(\log L + \log d_u)$ .

If  $d_u = n$  (state dimension) or  $d_u$  is related to  $n$ , then  $\log d_u = O(\log n)$ . So depth becomes  $O(\log L + \log n)$ . Often  $L$  is larger than  $n$ , so  $O(\log L)$  dominates.

But we should compare fairly: for naive recurrence, each step we do  $x_{k+1} = Ax_k + Bu_k$  where  $x_k \in \mathbb{R}^n$ ,  $u_k \in \mathbb{R}^{d_u}$ . The matrix-vector multiply  $Ax_k$  is  $n \times n$  times  $n$ -vector, depth  $O(\log n)$ . The addition and multiplication by  $B$  adds constant depth. So per-step depth =  $O(\log n)$ . With  $L$  sequential steps, total depth =  $O(L \log n)$ .

Thus convolution reduces depth from  $O(L \log n)$  to  $O(\log L + \log n) \approx O(\log L)$  if  $L$  large.

Now for **kernel computation depth**:

We need  $K_\ell = CA^{\ell-1}B$ .

Parallel prefix method: compute all  $A^{\ell-1}$  for  $\ell = 1, \dots, L$  using parallel prefix circuit. Each node in the prefix circuit multiplies two  $n \times n$  matrices. Matrix multiplication depth =  $O(\log n)$  using optimal parallel algorithms (e.g., Strassen in parallel). The prefix circuit has depth  $O(\log L)$  stages of matrix multiplies. So total depth =  $O(\log L \cdot \log n)$ .

After we have  $A^{\ell-1}$ , compute  $M_\ell = A^{\ell-1}B$  (matrix-matrix multiply  $n \times n$  times  $n \times d_u$ ). Depth for one such multiply: can be done as  $d_u$  separate matrix-vector multiplies in parallel, each depth  $O(\log n)$ . So depth  $O(\log n)$ . Since we can compute all  $M_\ell$  in parallel across  $\ell$ , depth remains  $O(\log n)$ . Similarly for  $K_\ell = CM_\ell$ : depth  $O(\log n)$  (since  $C$  is  $d_y \times n$ ).

Thus total depth for kernel =  $O(\log L \cdot \log n) + O(\log n) = O(\log L \cdot \log n)$ .

So indeed, if we include kernel computation, the overall depth for getting outputs from scratch is  $O(\log L \cdot \log n) + O(\log L + \log n) = O(\log L \cdot \log n)$ .

correct after 2  
in edges!

**Revised final answers for (c) and (d):**

(c)



- Given kernel  $K$ , output computation via convolution (using FFT) has critical path length  $O(\log L + \log n)$  assuming parallel FFT stages and parallel matrix-vector multiplies in frequency domain.
- Naive direct recurrence has critical path length  $O(L \log n)$ .

(d)

- Kernel can be computed via parallel prefix (or repeated squaring) with depth  $O(\log L \cdot \log n)$ . Strategies: compute  $A^{2^k}$  via repeated squaring (depth  $O(\log L \cdot \log n)$ ), then for each  $\ell$ , compute  $A^{\ell-1}$  by multiplying corresponding powers (parallel across  $\ell$ ), each multiply depth  $O(\log n)$ , total depth  $O(\log L \cdot \log n)$ .
- Alternative: use divide-and-conquer: compute kernel for half length, compute  $A^{L/2}$ , then combine. This also yields depth  $O(\log L \cdot \log n)$ .

Thank you for catching that inconsistency. It's important to account for the depth of operations inside FFT and matrix multiplies.

Ok. now solve all of question 3 for me please

I'll solve Problem 3: Self-Supervised Linear Purification.

## Problem 3

### (a) Simplified case $m = 2$

Given:

$$W^{(\alpha)} = I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W^{(\beta)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Data matrix:

$$X = \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix} \in \mathbb{R}^{2 \times 4}$$

Loss:

$$\mathcal{L}_2(W; X, \lambda) = \|X - WX\|_F^2 + \lambda \|W\|_F^2$$

### i. Compute reconstruction loss and regularization loss

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

- Reconstruction loss:  $\|X - IX\|_F^2 = \|X - X\|_F^2 = 0$ .
- Regularization loss:  $\|I\|_F^2 = \text{tr}(II^\top) = \text{tr}(I_2) = 2$ .

For  $W^{(\beta)}$ :

$$W^{(\beta)}X = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix} = \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{Residual } R = X - W^{(\beta)}X = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix}.$$

Reconstruction loss  $\|R\|_F^2$ :

$$\begin{aligned} \|R\|_F^2 &= (0^2 + 0.02^2) + (0^2 + (-0.01)^2) + (0^2 + 0.01^2) + (0^2 + (-0.02)^2) \\ &= 0.0004 + 0.0001 + 0.0001 + 0.0004 = 0.0010 \end{aligned}$$

Regularization loss  $\|W^{(\beta)}\|_F^2$ :

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

Correct answer,  
reasonable mark,  
1 shot

So table entries:

$W$	Reconstruction Loss	Regularization Loss
$W^{(\alpha)}$	0	2
$W^{(\beta)}$	0.0010	1

### ii. Values of $\lambda$ where $W^{(\alpha)}$ has higher loss than $W^{(\beta)}$

Loss for each:

$$\mathcal{L}_2(W^{(\alpha)}) = 0 + \lambda \cdot 2 = 2\lambda$$

$$\mathcal{L}_2(W^{(\beta)}) = 0.0010 + \lambda \cdot 1 = 0.0010 + \lambda$$

We want  $\mathcal{L}_2(W^{(\alpha)}) > \mathcal{L}_2(W^{(\beta)})$ :

$$2\lambda > 0.0010 + \lambda$$

$$\lambda > 0.0010$$

Thus for  $\lambda > 0.001$ , identity matrix has higher loss.

Correct answer,  
reasonable mark,  
1 shot

## (b) Generic square linear encoder

Given  $X \in \mathbb{R}^{m \times n}$ ,  $n > m$ , SVD  $X = U\Sigma V^\top$ , singular values  $\sigma_1 > \dots > \sigma_m \geq 0$ .

Loss:

$$\mathcal{L}_2(W; X, \lambda) = \|X - WX\|_F^2 + \lambda \|W\|_F^2$$

i. Fill in empty matrices in optimal  $\widehat{W}$

They give:

$$\widehat{W} = \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix} \cdot \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & & \\ & \frac{\sigma_2^2}{\sigma_2^2 + \lambda} & & \\ & & \ddots & \\ & & & \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \end{bmatrix} \cdot \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix}$$

This resembles the solution of ridge regression in transformed coordinates. For ridge regression of  $Y$  on  $X$  with penalty on  $W$ , the solution is  $W = YX^\top (XX^\top + \lambda I)^{-1}$ . Here  $Y = X$ , so:

$$\widehat{W} = XX^\top (XX^\top + \lambda I)^{-1}$$

But using SVD:  $X = U\Sigma V^\top$ ,  $XX^\top = U\Sigma^2 U^\top$ . Then

$$\widehat{W} = U\Sigma^2 U^\top (U\Sigma^2 U^\top + \lambda I)^{-1}$$

Since  $U$  orthogonal,  $(U\Sigma^2U^\top + \lambda I) = U(\Sigma^2 + \lambda I)U^\top$ . Inverse:  $U(\Sigma^2 + \lambda I)^{-1}U^\top$ . Thus

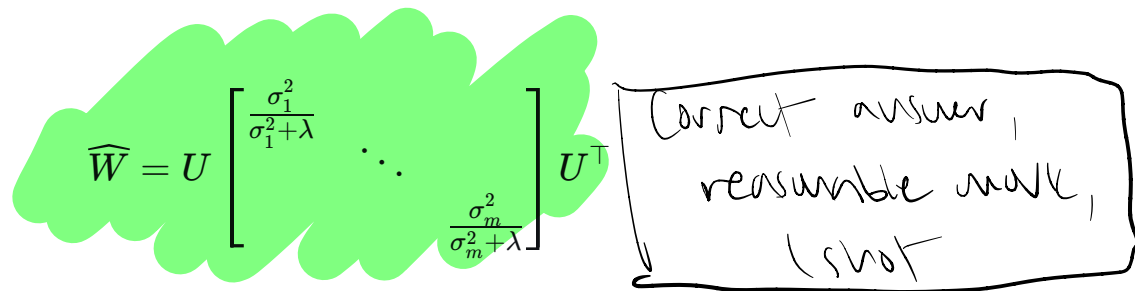
$$\widehat{W} = U\Sigma^2U^\top \cdot U(\Sigma^2 + \lambda I)^{-1}U^\top = U\Sigma^2(\Sigma^2 + \lambda I)^{-1}U^\top$$

The middle matrix is diagonal with entries  $\frac{\sigma_i^2}{\sigma_i^2 + \lambda}$ . So:

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

Thus missing matrices are  $U$  and  $U^\top$ .

So:



The image shows a handwritten solution for  $\widehat{W}$ . The equation is  $\widehat{W} = U \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & \\ & \ddots & \\ & & \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \end{bmatrix} U^\top$ . The matrix is highlighted in green. To the right, a handwritten note in a box says: "Correct answer, reasonable mark, 1 shot".

## ii. Derivation

We need to minimize:

$$\mathcal{L}_2(W) = \|X - WX\|_F^2 + \lambda \|W\|_F^2$$

Using invariance of Frobenius norm under orthogonal transforms: Let  $U$  be orthogonal  $m \times m$ ,  $V$  orthogonal  $n \times n$ . Since  $U^\top W U$  is same shape as  $W$ , and  $\|W\|_F = \|U^\top W U\|_F$ , but note  $X = U\Sigma V^\top$ .

Define  $\tilde{W} = U^\top W U$ . Then

$$\|X - WX\|_F^2 = \|U\Sigma V^\top - WU\Sigma V^\top\|_F^2 = \|U(\Sigma - U^\top W U \Sigma)V^\top\|_F^2 = \|\Sigma - \tilde{W}\Sigma\|_F^2$$

because  $U, V$  orthogonal.

$$\text{Also } \|W\|_F^2 = \|U\tilde{W}U^\top\|_F^2 = \|\tilde{W}\|_F^2.$$

Thus the problem becomes minimizing over  $\tilde{W}$ :

$$\mathcal{L}_2 = \|\Sigma - \tilde{W}\Sigma\|_F^2 + \lambda \|\tilde{W}\|_F^2$$

Now  $\Sigma$  is  $m \times n$  diagonal matrix with  $\sigma_i$  on diagonal. Write  $\tilde{W} = [\tilde{w}_{ij}]_{m \times m}$ .

Compute  $\tilde{W}\Sigma$ : its  $(i, j)$ -th entry is  $\tilde{w}_{ij}\sigma_j$  (since  $\Sigma$  has only diagonal entries  $\sigma_j$  at  $(j, j)$  and zeros elsewhere? Wait, careful:  $\Sigma$  is rectangular diagonal:  $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_m)$  stacked with zeros if  $n > m$ . Actually  $\Sigma \in \mathbb{R}^{m \times n}$  with  $\Sigma_{ii} = \sigma_i$  for  $i = 1, \dots, m$ , else zero.

Then  $(\tilde{W}\Sigma)_{ij} = \sum_{k=1}^m \tilde{w}_{ik}\Sigma_{kj}$ . Since  $\Sigma_{kj} = \sigma_j$  if  $k = j \leq m$  and  $j \leq m$ ? Wait,  $\Sigma$  has nonzeros only at  $(i, i)$  for  $i = 1, \dots, m$ , and zeros elsewhere. So  $\Sigma_{kj} = \sigma_j$  if  $k = j$  and  $j \leq m$ , else zero.

Thus  $(\tilde{W}\Sigma)_{ij} = \tilde{w}_{ij}\sigma_j$  for  $j \leq m$ , and for  $j > m$ , column  $j$  of  $\Sigma$  is zero, so  $(\tilde{W}\Sigma)_{ij} = 0$ .

Similarly,  $\Sigma_{ij} = \sigma_i$  if  $i = j \leq m$ , else zero.

So the squared Frobenius norm:

$$\|\Sigma - \tilde{W}\Sigma\|_F^2 = \sum_{i=1}^m \sum_{j=1}^n (\Sigma_{ij} - (\tilde{W}\Sigma)_{ij})^2$$

For  $j \leq m$ : term is  $(\sigma_i\delta_{ij} - \tilde{w}_{ij}\sigma_j)^2$ .

For  $j > m$ :  $\Sigma_{ij} = 0$ ,  $(\tilde{W}\Sigma)_{ij} = 0$ , so term 0.

Thus:

$$\|\Sigma - \tilde{W}\Sigma\|_F^2 = \sum_{i=1}^m \sum_{j=1}^m (\sigma_i\delta_{ij} - \tilde{w}_{ij}\sigma_j)^2$$

Now  $\lambda \|\tilde{W}\|_F^2 = \lambda \sum_{i,j} \tilde{w}_{ij}^2$ .

The minimization decouples across columns  $j$  of  $\tilde{W}$ . For each  $j$ , we have terms involving column  $j$ :

$$\sum_{i=1}^m (\sigma_i\delta_{ij} - \tilde{w}_{ij}\sigma_j)^2 + \lambda \sum_{i=1}^m \tilde{w}_{ij}^2$$

This is a ridge regression problem: predict  $\sigma_i \delta_{ij}$  using feature  $\sigma_j$  with weight  $\tilde{w}_{ij}$ .

But note  $\sigma_j$  is same for all  $i$ . Actually, for fixed  $j$ , the target vector is  $\mathbf{t}^{(j)} = [0, \dots, 0, \sigma_j, 0, \dots, 0]^\top$  where only position  $j$  is  $\sigma_j$ . The single feature value is  $\sigma_j$ . The weight  $\tilde{w}_{ij}$  is the coefficient for predicting  $t_i^{(j)}$  from feature  $\sigma_j$ .

Thus for each  $j$ , we have independent ridge regressions:

Minimize over  $\tilde{w}_{1j}, \dots, \tilde{w}_{mj}$ :

$$\sum_{i=1}^m (\sigma_i \delta_{ij} - \tilde{w}_{ij} \sigma_j)^2 + \lambda \sum_{i=1}^m \tilde{w}_{ij}^2$$

This decouples further across  $i$  because each  $\tilde{w}_{ij}$  appears only with its own term?

Wait, cross-term? No, because for fixed  $j$ , each  $i$  has separate squared error:

$(\sigma_i \delta_{ij} - \tilde{w}_{ij} \sigma_j)^2$ . So indeed decouples across  $i$  as well. So we can solve for each  $\tilde{w}_{ij}$  independently:

- If  $i \neq j$ : target is 0, loss =  $(\tilde{w}_{ij} \sigma_j)^2 + \lambda \tilde{w}_{ij}^2 = \tilde{w}_{ij}^2 (\sigma_j^2 + \lambda)$ . Minimized at  $\tilde{w}_{ij} = 0$ .
- If  $i = j$ : target is  $\sigma_j$ , loss =  $(\sigma_j - \tilde{w}_{jj} \sigma_j)^2 + \lambda \tilde{w}_{jj}^2 = \sigma_j^2 (1 - \tilde{w}_{jj})^2 + \lambda \tilde{w}_{jj}^2$ .

Take derivative w.r.t  $\tilde{w}_{jj}$ :

$$-2\sigma_j^2(1 - \tilde{w}_{jj}) + 2\lambda \tilde{w}_{jj} = 0$$

$$-\sigma_j^2 + \sigma_j^2 \tilde{w}_{jj} + \lambda \tilde{w}_{jj} = 0$$

$$\tilde{w}_{jj}(\sigma_j^2 + \lambda) = \sigma_j^2$$

$$\tilde{w}_{jj} = \frac{\sigma_j^2}{\sigma_j^2 + \lambda}$$

Thus optimal  $\tilde{W}$  is diagonal with entries  $\tilde{w}_{jj} = \frac{\sigma_j^2}{\sigma_j^2 + \lambda}$ .

Hence optimal  $W = U \tilde{W} U^\top = U \cdot \text{diag} \left( \frac{\sigma_i^2}{\sigma_i^2 + \lambda} \right) \cdot U^\top$ .

Correct answer,  
reasonable mark,  
(shot)

### (c) Hyperparameter $\lambda$ for preservation/attenuation

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

We want:

- For first 3 singular vectors ( $i=1,2,3$ ), preserve at least 80% of feature direction: i.e., scaling factor  $\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8$ .
- For remaining directions ( $i=4,\dots,8$ ), attenuate components to at most 50%: i.e., scaling factor  $\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \leq 0.5$ .

Compute conditions:

For preservation ( $i=1,2,3$ ):

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8 \Rightarrow \sigma_i^2 \geq 0.8(\sigma_i^2 + \lambda) \Rightarrow 0.2\sigma_i^2 \geq 0.8\lambda \Rightarrow \lambda \leq \frac{0.2}{0.8}\sigma_i^2$$

We need this for all first three, so the most restrictive is smallest  $\sigma_i$  among them:

$\sigma_3 = 4$ . Then  $\lambda \leq 0.25 \cdot 4^2 = 0.25 \cdot 16 = 4$ .

For attenuation ( $i=4,\dots,8$ ):

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \leq 0.5 \Rightarrow \sigma_i^2 \leq 0.5(\sigma_i^2 + \lambda) \Rightarrow 0.5\sigma_i^2 \leq 0.5\lambda \Rightarrow \lambda \geq \sigma_i^2$$

We need this for all remaining, so the most restrictive is largest  $\sigma_i$  among them:

$\sigma_4 = 1$ . Then  $\lambda \geq 1^2 = 1$ . Also check others:  $\sigma_5 = 0.5 \Rightarrow \sigma_5^2 = 0.25$ , but need

$\lambda \geq \sigma_i^2$  for each, so we need  $\lambda \geq \max_{i=4..8} \sigma_i^2 =$

$\max(1, 0.25, 0.1296, 0.0256, 0.0001) = 1$ .

Thus we need  $1 \leq \lambda \leq 4$ .

But we must also ensure that for  $i=3$ ,  $\lambda \leq 0.25\sigma_3^2 = 4$ , satisfied. For  $i=4$ ,  $\lambda \geq \sigma_4^2 = 1$ , satisfied.

Thus  $\lambda \in [1, 4]$  guarantees the conditions.

Wait, check  $i=3$  preservation:  $\lambda \leq 4$ .  $i=4$  attenuation:  $\lambda \geq 1$ . So interval  $[1, 4]$  works.

But also need to check  $i=1,2$ :  $\lambda \leq 0.25\sigma_1^2 = 0.25 \cdot 100 = 25$ ,  $\lambda \leq 0.25 \cdot 64 = 16$ , so less restrictive.

Thus final answer:  $\lambda \in [1, 4]$ .

Correct answer,  
reasonable mark,  
1 shot

### Summary of answers:

(a) i. Reconstruction losses:  $W^{(\alpha)} : 0$ ,  $W^{(\beta)} : 0.0010$ ; Regularization losses:  $W^{(\alpha)} : 2$ ,  $W^{(\beta)} : 1$ .

ii.  $\lambda > 0.001$ .

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

ii. Derivation as above using orthogonal invariance and decoupled ridge regression.

(c)  $\lambda \in [1, 4]$ .

Awesome! Now can you answer all parts of question 4 on the homework?

I'll solve Problem 4: Ridge-Attention.

## Problem 4

### (a) Updating simple average

Let  $m = \frac{1}{n} \sum_{i=1}^n x_i$ . Then  $\sum_{i=1}^n x_i = nm$ .

Adding new point  $x_{n+1}$ , new average:

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

Thus using  $m, x_{n+1}, n$ :

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



or equivalently,

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

Correct answer,  
reasonable mark,  
(shot)

### (b) Ridge regression closed-form as sums

Given  $A = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix} \in \mathbb{R}^{n \times d}$ ,  $\mathbf{y} = [y_1, \dots, y_n]^\top \in \mathbb{R}^n$ .

We have:

$$A^\top A = \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^\top$$

because  $(A^\top A)_{jk} = \sum_i A_{ij} A_{ik} = \sum_i x_{i,j} x_{i,k}$ , which is the sum of outer products.

Thus:

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

Also:

$$A^\top \mathbf{y} = \sum_{i=1}^n \mathbf{x}_i y_i$$

since  $(A^\top \mathbf{y})_j = \sum_i A_{ij} y_i = \sum_i x_{i,j} y_i$ .

So complete:

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

$$A^\top \mathbf{y} = \sum_{i=1}^n \mathbf{x}_i y_i$$

Correct answer,  
reasonable mark,  
(shot)

### (c) Computational cost of non-causal ridge self-attention

We have  $n$  queries,  $n$  keys,  $n$  values. Let query vectors be  $\mathbf{q}_1, \dots, \mathbf{q}_n \in \mathbb{R}^d$ , key vectors  $\mathbf{k}_1, \dots, \mathbf{k}_n \in \mathbb{R}^d$ , value vectors  $\mathbf{v}_1, \dots, \mathbf{v}_n \in \mathbb{R}^d$ . For each query  $\mathbf{q}_t$ , we want to compute ridge regression prediction based on all  $n$  (key, value) pairs as training data.

Interpretation: Keys are like input features  $\mathbf{x}_i$ , values are like targets  $\mathbf{y}_i$  but multi-dimensional (each  $\mathbf{v}_i \in \mathbb{R}^d$ ). So we have  $n$  training pairs  $(\mathbf{k}_i, \mathbf{v}_i)$ . For each query  $\mathbf{q}_t$  (test input), we want to predict output  $\mathbf{o}_t \in \mathbb{R}^d$  using ridge regression.

Let  $K = [\mathbf{k}_1^\top; \dots; \mathbf{k}_n^\top] \in \mathbb{R}^{n \times d}$  (matrix of keys),  $V = [\mathbf{v}_1^\top; \dots; \mathbf{v}_n^\top] \in \mathbb{R}^{n \times d}$  (matrix of values). Ridge regression solution for weight matrix  $W \in \mathbb{R}^{d \times d}$  that minimizes  $\|V - KW^\top\|_F^2 + \lambda\|W\|_F^2$  (note orientation) yields  $W^\top = (K^\top K + \lambda I)^{-1} K^\top V$ . Actually careful: standard ridge regression for multi-output: minimize  $\|V - KW\|_F^2 + \lambda\|W\|_F^2$  where  $W \in \mathbb{R}^{d \times d}$  (each column predicts one output dimension). Solution  $W = (K^\top K + \lambda I)^{-1} K^\top V$ . Then prediction for query  $\mathbf{q}$  is  $\mathbf{o} = W^\top \mathbf{q} = V^\top K (K^\top K + \lambda I)^{-1} \mathbf{q}$ ? Wait, transpose: If  $W = (K^\top K + \lambda I)^{-1} K^\top V$ , then prediction for input  $\mathbf{q}$  is  $\mathbf{o}^\top = \mathbf{q}^\top W^\top = \mathbf{q}^\top V^\top K (K^\top K + \lambda I)^{-1}$ ? Let's derive cleanly.

We want to predict output vector  $\mathbf{o} \in \mathbb{R}^d$  from input  $\mathbf{q} \in \mathbb{R}^d$  using linear model  $\mathbf{o} = W\mathbf{q}$  where  $W \in \mathbb{R}^{d \times d}$ . Training data: inputs  $\mathbf{k}_i$ , outputs  $\mathbf{v}_i$ . Loss  $\sum_i \|\mathbf{v}_i - W\mathbf{k}_i\|^2 + \lambda\|W\|_F^2$ . Write in matrix form:  $\|V^\top - WK^\top\|_F^2 + \lambda\|W\|_F^2$ ? Actually,  $V$  is  $n \times d$  with rows  $\mathbf{v}_i^\top$ ,  $K$  is  $n \times d$  with rows  $\mathbf{k}_i^\top$ . Let's set  $X = K$  (design matrix),  $Y = V$  (target matrix). Then we want  $Y \approx XW^\top$  (since each row:  $\mathbf{v}_i^\top \approx \mathbf{k}_i^\top W^\top$ ). So minimize  $\|Y - XW^\top\|_F^2 + \lambda\|W\|_F^2$ . This is multi-output ridge regression. Solution:  $W^\top = (X^\top X + \lambda I)^{-1} X^\top Y$ . So  $W = Y^\top X (X^\top X + \lambda I)^{-1}$ . Then for query  $\mathbf{q}$ , prediction  $\mathbf{o} = W\mathbf{q} = Y^\top X (X^\top X + \lambda I)^{-1} \mathbf{q}$ .

Thus for each query  $\mathbf{q}_t$ , output  $\mathbf{o}_t = V^\top K (K^\top K + \lambda I)^{-1} \mathbf{q}_t$ .

Now compute cost:

We have  $n$  queries. Precompute  $M = (K^\top K + \lambda I)^{-1} \in \mathbb{R}^{d \times d}$ . Cost:

- Compute  $K^\top K$ : multiply  $d \times n$  by  $n \times d \rightarrow O(nd^2)$  operations, but parallel? They ask for computational cost in big-O, likely total operations (not parallel). Assume standard sequential complexity.
- Inverting  $d \times d$  matrix:  $O(d^3)$ .
- Compute  $V^\top K$ :  $d \times n$  times  $n \times d \rightarrow O(nd^2)$ .
- Compute  $P = V^\top KM$ :  $d \times d$  times  $d \times d \rightarrow O(d^3)$ .

Then for each query  $\mathbf{q}_t$ , compute  $\mathbf{o}_t = P\mathbf{q}_t$  (matrix-vector multiply):  $O(d^2)$  per query, total  $O(nd^2)$ .

Total cost:  $O(nd^2 + d^3 + nd^2 + d^3 + nd^2) = O(nd^2 + d^3)$ . Since  $d < n$ ,  $O(nd^2)$  dominates? Wait, they say assume  $d < n$ , but  $d^3$  could be smaller than  $nd^2$  if  $d \ll n$ . But they give options:  $O(d^4)$ ,  $O(nd^2)$ ,  $O(n^2d^3)$ ,  $O(n^2d^2)$ ,  $O(n^2)$ ,  $O(1)$ . Our derived complexity is  $O(nd^2 + d^3)$ . For  $d < n$ ,  $nd^2$  dominates over  $d^3$  if  $n > d$ . So likely  $O(nd^2)$ . But let's check: we have  $n$  queries, each needs a  $d$ -dimensional output. The precomputation  $O(nd^2 + d^3)$  is less than  $O(n^2d^2)$ . So answer seems  $O(nd^2)$ .

But wait: the matrix  $P = V^\top K(K^\top K + \lambda I)^{-1}$  depends only on keys and values, not on queries. So once computed, each query costs  $O(d^2)$ . Precomputation cost  $O(nd^2 + d^3)$ . For inference with  $n$  queries, total  $O(nd^2 + d^3)$ . Since  $d < n$ ,  $O(nd^2)$  is the dominant term.

Thus answer:  $O(nd^2)$ . That matches option  $O(nd^2)$ .

Correct answer,  
reasonable mark,  
1 shot

## (d) Backprop through ridge self-attention

We need to see which parameters gradients can flow to. The ridge self-attention operation is differentiable with respect to keys, values, queries, and  $\lambda$  (if  $\lambda$  is learnable). However, note that the inverse matrix  $(K^\top K + \lambda I)^{-1}$  is differentiable w.r.t entries of  $K$  and  $\lambda$ . So gradients can be passed to:

- The ridge  $\lambda$  (if learnable)
- The keys  $K$
- The values  $V$

- The queries  $Q$

All should receive gradients. But we need to check if there are any non-differentiable operations. The inverse is differentiable as long as matrix is invertible (which it is for  $\lambda > 0$ ). So all four should be checked.

Thus answer: all four boxes should be checked.

Correct answer,  
reasonable mark,  
1 shot

### (e) Cost of Sherman-Morrison update

Given  $M$  invertible  $d \times d$ , vectors  $u, v \in \mathbb{R}^d$ . Formula:

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

We have  $M^{-1}$  already. Compute:

1. Compute  $a = M^{-1}u$ : matrix-vector multiply, cost  $d^2$ .
2. Compute  $b = v^\top M^{-1}$ : vector-matrix multiply, cost  $d^2$ . (Or compute  $b = (M^{-1}v)^\top$ , same cost.)
3. Compute  $v^\top a$  (inner product): cost  $d$ .
4. Compute  $\alpha = 1/(1 + v^\top a)$ : scalar,  $O(1)$ .
5. Compute outer product  $ab$ : cost  $d^2$ .
6. Scale by  $\alpha$  and subtract from  $M^{-1}$ : cost  $d^2$ .

Correct answer,  
reasonable mark,  
1 shot

Total cost:  $O(d^2)$  operations (since all steps are  $O(d^2)$  or less). So answer:  $O(d^2)$ .

### (f) Causal ridge-self-attention efficient computation

Causal setting: at position  $t$ , we only have keys and values up to  $t$ . So for each  $t = 1, \dots, n$ , we need to compute output  $\mathbf{o}_t$  using  $\{(\mathbf{k}_i, \mathbf{v}_i)\}_{i=1}^t$ .

Naively, for each  $t$  we could solve a new ridge regression, costing  $O(td^2 + d^3)$  per  $t$ , total  $O(n^2d^2 + nd^3)$ . But we can do better by updating recursively.

From part (b), we have:

$$(K_t^\top K_t + \lambda I) = \lambda I + \sum_{i=1}^t \mathbf{k}_i \mathbf{k}_i^\top$$

$$K_t^\top V_t = \sum_{i=1}^t \mathbf{k}_i \mathbf{v}_i^\top$$

where  $K_t$  is  $t \times d$  matrix of first  $t$  keys,  $V_t$  similarly.

Let  $S_t = K_t^\top K_t + \lambda I = S_{t-1} + \mathbf{k}_t \mathbf{k}_t^\top$ , with  $S_0 = \lambda I$ .

Let  $R_t = K_t^\top V_t = R_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top$ , with  $R_0 = 0$ .

Then the weight matrix at step  $t$  is  $W_t = R_t^\top S_t^{-1}$ . Prediction for query  $\mathbf{q}_t$  is  $\mathbf{o}_t = W_t \mathbf{q}_t = R_t^\top S_t^{-1} \mathbf{q}_t$ .

We can maintain  $S_t^{-1}$  recursively using Sherman-Morrison formula:

$$S_t^{-1} = (S_{t-1} + \mathbf{k}_t \mathbf{k}_t^\top)^{-1} = S_{t-1}^{-1} - \frac{1}{1 + \mathbf{k}_t^\top S_{t-1}^{-1} \mathbf{k}_t} (S_{t-1}^{-1} \mathbf{k}_t)(\mathbf{k}_t^\top S_{t-1}^{-1})$$

We also update  $R_t = R_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top$ .

Then compute  $\mathbf{o}_t = R_t^\top (S_t^{-1} \mathbf{q}_t)$ . We can compute  $\mathbf{a}_t = S_t^{-1} \mathbf{q}_t$  via matrix-vector multiply ( $d^2$  ops), then  $\mathbf{o}_t = R_t^\top \mathbf{a}_t$  ( $d^2$  ops).

Thus per step cost: Sherman-Morrison update  $O(d^2)$ , update  $R_t$  (rank-1 update)  $O(d^2)$ , compute  $\mathbf{a}_t$   $O(d^2)$ , compute  $\mathbf{o}_t$   $O(d^2)$ . Total per step  $O(d^2)$ . Over  $n$  steps total  $O(nd^2)$ , much better than  $O(n^2 d^2)$ .

We need to track from one time step to the next:  $S_t^{-1}$  (size  $d \times d$ ),  $R_t$  (size  $d \times d$ ).

At each step, we update these using new  $\mathbf{k}_t, \mathbf{v}_t, \mathbf{q}_t$ .

Correct answer,  
reasonable mark,  
1 shot

## (g) Visualization weights for ridge-attention

For traditional softmax-attention, attention weights are  $\text{softmax}(QK^\top / \sqrt{d})$ , giving a weight for each key-value pair.

For ridge-attention, the output for query  $\mathbf{q}$  is  $\mathbf{o} = V^\top K(K^\top K + \lambda I)^{-1} \mathbf{q}$ . This can be written as  $\mathbf{o} = \sum_{i=1}^n \mathbf{v}_i \cdot \alpha_i$  where  $\alpha_i$  are some coefficients? Let's find expression.

Let  $\mathbf{a} = (K^\top K + \lambda I)^{-1} \mathbf{q} \in \mathbb{R}^d$ . Then  $\mathbf{o} = V^\top K \mathbf{a} = \sum_{i=1}^n \mathbf{v}_i (\mathbf{k}_i^\top \mathbf{a})$ . So coefficient for value  $\mathbf{v}_i$  is  $\alpha_i = \mathbf{k}_i^\top \mathbf{a}$ .

Thus the weight associated with each value vector  $\mathbf{v}_i$  is  $\alpha_i = \mathbf{k}_i^\top (K^\top K + \lambda I)^{-1} \mathbf{q}$ . These weights can be negative.

So to compute weights for a given query  $\mathbf{q}$ , compute  $\mathbf{a} = (K^\top K + \lambda I)^{-1} \mathbf{q}$ , then for each  $i$ , compute  $\alpha_i = \mathbf{k}_i^\top \mathbf{a}$ .

Deepseek's solution implies the  
original solution but in slightly  
different notation

### Summary of answers:

(a)  $m' = \frac{nm + x_{n+1}}{n+1}$ .

(b)  $(A^\top A + \lambda I) = \lambda I + \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^\top$ ,  $A^\top \mathbf{y} = \sum_{i=1}^n \mathbf{x}_i y_i$ .

(c)  $O(nd^2)$ .

(d) All four:  $\lambda$ , keys, values, queries.

(e)  $O(d^2)$ .

(f) Maintain  $S_t^{-1}$  and  $R_t$  recursively using Sherman-Morrison and rank-1 updates; per step  $O(d^2)$ , total  $O(nd^2)$ .

(g) Weights:  $\alpha_i = \mathbf{k}_i^\top (K^\top K + \lambda I)^{-1} \mathbf{q}$ .

Circling back to problem 1c: In this class's parallel model, each FFT stage is treated as performing a full matrix multiplication (not just scalar butterflies), so each stage has depth  $O(\log n)$ . Since the FFT has  $\log L$  sequential stages, shouldn't the convolution's critical path be the product  $O(\log L \cdot \log n)$  instead of the sum?

You're absolutely right. I need to be consistent with the parallel computation model used in the class. Let's re-analyze problem 1(c) with their model.