

# Special Participation A: Claude Sonnet 4.5 on HW 8

Celine Tan

December 2025

## 1 Problem 1

User: [\[hw8.pdf\]](#) Solve problem 1.

LLM: I'll solve Problem 1 on SSM Convolution Kernel step by step.

### Problem 1: SSM Convolution Kernel

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

Given the SSM:  $-x_{k+1} = Ax_k + Bu_k - y_k = Cx_k + Du_k - x_0 = 0$

##### Finding the output expression:

Let me unroll the recursion:

- $x_1 = Ax_0 + Bu_0 = Bu_0$
- $x_2 = Ax_1 + Bu_1 = ABu_0 + Bu_1$
- $x_3 = Ax_2 + Bu_2 = A^2Bu_0 + ABu_1 + Bu_2$
- $x_k = A^{k-1}Bu_0 + A^{k-2}Bu_1 + \dots + ABu_{k-2} + Bu_{k-1}$

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

Now for the output:

$$\begin{aligned} y_k &= Cx_k + Du_k = C \sum_{\ell=0}^{k-1} A^{k-1-\ell}Bu_\ell + Du_k \\ &= Du_k + \sum_{\ell=0}^{k-1} CA^{k-1-\ell}Bu_\ell \end{aligned}$$

Reindexing with  $j = k - \ell$ :

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

where we extend this to  $\sum_{\ell=0}^L K_\ell u_{k-\ell}$  with zero-padding for negative indices.

\*\*The kernel is:\*\*

$$K_\ell = \begin{cases} D & \text{if } \ell = 0 \\ CA^{\ell-1}B & \text{if } \ell \geq 1 \end{cases}$$

Or more compactly:  $K = [D, CB, CAB, CA^2B, CA^3B, \dots, CA^{L-1}B]$

This is entirely correct. The LLM begins with a few small examples, which makes things easier for a human to understand (and is the same approach as the key).

---

## 1.2 (b) Concrete Examples

### 1.2.1 i. Scalar Case

Given:  $A = 0.8, B = 1, C = 1.5, D = \delta$  (assuming  $\delta = 0$  since not specified)

Computing kernel up to  $L = 4$ :

- $K_0 = D = 0$
- $K_1 = CB = 1.5 \times 1 = 1.5$
- $K_2 = CAB = 1.5 \times 0.8 \times 1 = 1.2$
- $K_3 = CA^2B = 1.5 \times 0.64 \times 1 = 0.96$
- $K_4 = CA^3B = 1.5 \times 0.512 \times 1 = 0.768$

**Kernel:**  $K = [0, 1.5, 1.2, 0.96, 0.768]$

### 1.2.2 ii. 2D Case

Given:  $A = \begin{bmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, C = [1, 0], D = 0$

Computing:

- $K_0 = D = 0$
- $K_1 = CB = [1, 0] \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1$
- $K_2 = CAB = [1, 0] \begin{bmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = [1, 0] \begin{bmatrix} 0.7 \\ 0.2 \end{bmatrix} = 0.7 - A^2 = \begin{bmatrix} 0.51 & 0.13 \\ 0.26 & 0.38 \end{bmatrix}$
- $K_3 = CA^2B = [1, 0] \begin{bmatrix} 0.51 \\ 0.26 \end{bmatrix} = 0.51$

**Kernel:**  $K = [0, 1, 0.7, 0.51]$

This is entirely correct. Claude makes the (correct) assumption that  $\delta = 0$  from the context of the problem.

**Discussion:** The kernel captures the impulse response - if we input a unit impulse at time 0 (i.e.,  $u_0 = 1$  and  $u_t = 0$  for  $t > 0$ ), the output sequence is exactly the kernel values. The decay pattern shows how the system's memory of past inputs fades over time through the eigenvalues of  $A$ .

I think this would technically be correct, but I (personally) would like to have had a tad more explanation specifically tying the "system's memory" to the kernel values.

---

### 1.3 (c) Efficient Computation with Convolutions

If we already have kernel  $K$ :

- **Parallelization:** We can compute all outputs  $\{y_0, y_1, \dots, y_L\}$  in **fully parallel** fashion using FFT-based convolution

- **Critical path length:**  $O(\log L)$  using FFT (Fast Fourier Transform)
- FFT has complexity  $O(L \log L)$  but critical path depth  $O(\log L)$

**Naive direct computation:** - Computing  $y_k = \sum_{\ell=0}^k K_\ell u_{k-\ell}$  sequentially

- Each  $y_k$  depends on  $y_{k-1}$  being computed first

- **Critical path length:**  $O(L)$  - fully sequential

**Speed-up:** Convolution-based approach reduces critical path from  $O(L)$  to  $O(\log L)$

This is where Claude struggles a bit. It doesn't account for the complexity of the matrix multiplications.

---

### 1.4 (d) Efficient Kernel Computation

Given  $A, B, C$ , we need:  $K = [D, CB, CAB, CA^2B, \dots, CA^{L-1}B]$

**Divide-and-Conquer Strategy** (assuming  $L = 2^N$ ):

1. **Observation:** Computing  $CA^iB$  for all  $i$  naively is  $O(L)$  sequential matrix multiplications

2. **Parallel approach:** Use binary decomposition - Precompute powers of  $A$ :  $A, A^2, A^4, A^8, \dots, A^{2^{N-1}}$  - This takes  $O(\log L)$  sequential steps (each step doubles the power)

3. **Combine:** Any  $A^i$  can be written as a product of at most  $\log L$  of these precomputed powers - Example:  $A^{13} = A^8 \cdot A^4 \cdot A^1$

4. **Parallel computation tree:** All kernel elements can be computed in  $O(\log L)$  depth

**Complexity:**  $O(n^3L)$  work,  $O(n^3 \log L)$  depth (for  $n \times n$  matrix  $A$ )

This is a similar approach to the key, but instead of pre-computing the powers of  $A$  up to  $L/2$ , Claude computes the powers of  $A$  with powers of 2, and breaks all subsequent computations into products of these powers. The complexity analysis is not correct, as the factor it uses to scale for matrix multiplication assumes sequential multiplication rather than the parallelized  $O(\log n)$  path length.

---

## 1.5 (e) Diagonal Matrix Structure

If  $A = \text{diag}(\lambda_1, \dots, \lambda_n)$ :

**Key insight:**  $A^\ell = \text{diag}(\lambda_1^\ell, \dots, \lambda_n^\ell)$

**Efficient computation:**

- No matrix multiplication needed!
- $CA^\ell B$  can be computed element-wise
- For each kernel element  $K_{\ell+1} = CA^\ell B$ , we just compute:  $\sum_{i=1}^n C_i \lambda_i^\ell B_i$
- All  $L$  kernel elements can be computed **independently in parallel**

**Complexity:**  $O(nL)$  work,  $O(1)$  depth (fully parallel)

This assumes that all the powers of  $A$  can be computed in parallel and thus underestimates the time complexity.

---

---

## 1.6 (f) Diagonal-Plus-Low-Rank (DPLR) Structure

Given  $A = I_n + pp^\top$  (rank-1 update to identity):

**Key insight:** Use Sherman-Morrison formula or repeated application:

For  $A^\ell = (I + pp^\top)^\ell$ , we can use:  
- Binomial expansion:  $(I + pp^\top)^\ell = \sum_{k=0}^{\ell} \binom{\ell}{k} (pp^\top)^k$   
- Note:  $(pp^\top)^k = (p^\top p)^{k-1} (pp^\top)$  for  $k \geq 1$   
Let  $\alpha = p^\top p$ , then:

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

**Efficient computation:**

$$\begin{aligned} CA^\ell B &= CB + C \left( \sum_{k=1}^{\ell} \binom{\ell}{k} \alpha^{k-1} \right) pp^\top B \\ &= CB + (Cp)(p^\top B) \sum_{k=1}^{\ell} \binom{\ell}{k} \alpha^{k-1} \end{aligned}$$

The sum can be computed as:  $\frac{(1+\alpha)^\ell - 1}{\alpha}$  when  $\alpha \neq 0$

**Complexity:**

- Precompute:  $Cp, p^\top B, CB, \alpha = p^\top p$  -  $O(n)$
- For each  $\ell$ :  $O(1)$  computation
- **Total:**  $O(n + L)$  work,  $O(1)$  depth for kernel computation

This is dramatically more efficient than the  $O(n^3L)$  naive approach!

In subsequent parts, Claude continues to struggle with complexity and path length analyses (despite being able to handle the more computational aspects of the problems correctly zero-shot). This may be partially my fault for asking it to solve the entire problem in one go – it may have been more helpful to ask it to solve the problem part-by-part, correcting along the way. To amend this, I try to guide Claude into the correct answer with small hints.

## 2 Problem 1 Retry

User: Try double-checking your answer for 1(c). What is the critical path length of matrix multiplication, and how would that be applicable here?

You're absolutely right to question this! Let me reconsider part (c) more carefully.

### 2.1 (c) Efficient Computation with Convolutions - REVISED

If we already have kernel **K**:

Using FFT-based convolution:

- We can compute the convolution  $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$  for all  $k$

- **Work complexity:**  $O(L \log L)$  using FFT

- **Critical path (depth):**  $O(\log L)$  - this part was correct

**Naive direct computation from unrolled recursion:**

Here's where I need to reconsider. The naive approach computes:

-  $x_k = Ax_{k-1} + Bu_{k-1}$  (recurrence relation)

-  $y_k = Cx_k + Du_k$

Each step requires:

- **Matrix-vector multiplication**  $Ax_{k-1}$ : For an  $n \times n$  matrix and  $n$ -dimensional vector, the critical path is  $O(\log n)$  using parallel multiplication and addition trees

- **Vector addition**  $Ax_{k-1} + Bu_{k-1}$ :  $O(\log n)$

- Then computing  $Cx_k$ :  $O(\log n)$

- And  $Cx_k + Du_k$ :  $O(\log n)$

So each time step has critical path  $O(\log n)$ .

Since we must compute  $x_1, x_2, \dots, x_L$  sequentially (each depends on the previous), and we have  $L$  time steps:

**Critical path length:**  $O(L \log n)$

**Comparison:**

- **Convolution approach:**  $O(\log L)$  - depends only on sequence length

- **Naive recurrent approach:**  $O(L \log n)$  - linear in sequence length, logarithmic in state dimension

**Key insight:** The convolution approach parallelizes over the sequence length dimension, while the recurrent approach is inherently sequential in time. The  $\log n$  factor in the naive approach comes from the critical path of each matrix-vector multiplication, but the dominant factor is the  $L$  sequential steps.

Thank you for catching that! The critical difference is that convolution is  $O(\log L)$  while recurrence is  $O(L \log n)$  - a massive improvement when  $L$  is large.

This is better (got the correct naive path length), but still not quite right, since it still doesn't account for complexity of matrix multiplication in the convolutional approach.

User: Wouldn't critical path length of matrix multiplication also be applicable to the critical path length of the parallelized computation?

You're absolutely right! Let me think through this more carefully.

## 2.2 (c) Efficient Computation with Convolutions - CORRECTED

If we already have kernel **K**:

Using FFT-based convolution to compute  $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$ :

But wait - the inputs and outputs are **d-dimensional vectors** (state dimension  $n$ ), not scalars!

So we need to think about what the convolution actually does:

- If  $u_k \in \mathbb{R}^d$  and we want  $y_k \in \mathbb{R}^n$ , then each kernel element  $K_\ell$  is actually an  $n \times d$  matrix

- Computing  $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$  involves:
- $L$  matrix-vector multiplications:  $K_\ell u_{k-\ell}$
- $L$  vector additions to sum them up

**Critical path analysis:** - Each matrix-vector multiplication  $K_\ell u_{k-\ell}$ :  $O(\log d)$  (assuming  $K_\ell$  is  $n \times d$ )

- Summing  $L$  vectors using a parallel reduction tree:  $O(\log L)$

- **Total critical path:**  $O(\log d + \log L) = O(\log(dL))$  This is where Claude struggles, again. It assumes that the total critical path complexity is additive, rather than multiplicative.

Actually, with FFT-based convolution (applying FFT to each dimension):

- **Critical path:**  $O(\log L + \log d)$

**Naive direct computation:**

- Each step:  $x_k = Ax_{k-1} + Bu_{k-1}$  requires matrix-vector multiplication with critical path  $O(\log n)$

- We have  $L$  sequential steps

- **Critical path:**  $O(L \log n)$

**Comparison:**

- **Convolution approach:**  $O(\log L + \log n)$  (or  $O(\log(nL))$ )

- **Naive recurrent approach:**  $O(L \log n)$

The key difference is:

- Convolution: **logarithmic** in  $L$  (parallelizes over time)
- Recurrence: **linear** in  $L$  (sequential in time)

Thank you for pushing me to think this through properly! The matrix-vector operations contribute a  $\log n$  factor to both approaches, but the crucial distinction is  $\log L$  vs  $L$ .

User: Why did you choose to compute the corrected path length by adding  $\log d$  and  $\log L$ ? Double check your thinking here. Also, double check your analysis of the complexities of parts (d)-(f) as well.

You're right to push back on this! Let me think more carefully.

### 2.3 (c) Efficient Computation with Convolutions - RE- ALLY CORRECTED

If we already have kernel  $K$ :

For the convolution  $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$  where  $K_\ell$  are  $n \times d$  matrices:

Each output  $y_k$  requires: - Computing  $L$  matrix-vector products  $K_\ell u_{k-\ell}$ , each with critical path  $O(\log d)$  - Summing these  $L$  vectors with critical path  $O(\log L)$

These happen **sequentially**, not in parallel! First we do the matrix-vector multiplications, then we sum.

**Critical path:**  $O(\log d + \log L)$ ...

Wait, no! If we're computing all the matrix-vector products in parallel, each takes  $O(\log d)$  critical path, and they all happen simultaneously, so that's just  $O(\log d)$ . Then summing  $L$  results takes  $O(\log L)$ . These are sequential operations, so we add them:  $O(\log d + \log L)$ .

Actually, I think the issue is: can we truly parallelize across all  $\ell$  values? Yes, if we have enough processors. So the critical path is indeed  $O(\log d + \log L)$ . **Claude seems to get stuck on the wrong issue, and is making some incorrect assumptions.**

But actually, for **FFT-based convolution**\*\*, we're doing convolution in frequency domain, which changes the analysis. Let me reconsider: - Transform to frequency domain:  $O(\log L)$  depth for FFT - Pointwise multiplication in frequency domain:  $O(\log d)$  for each of  $L$  frequency bins (matrix-vector products)  
- Transform back:  $O(\log L)$  depth

So critical path is still  $O(\log L + \log d)$ .

Hmm, but actually for the pointwise multiplications - we have  $L$  of them and they're independent, so if done in parallel that's just  $O(\log d)$ , not  $O(L \log d)$ .

So: **\*\* $O(\log L + \log d)$ \*\***

**\*\*Naive recurrent approach:\*\***

- $L$  sequential time steps, each with critical path  $O(\log n)$
- **\*\*Critical path:\*\***  $O(L \log n)$

## 2.4 (d)-(f) Complexity Revisions

Let me reconsider these more carefully:

### 2.4.1 (d) Efficient Kernel Computation

Computing  $K_\ell = CA^{\ell-1}B$  for  $\ell = 1, \dots, L$ :

\*\*Divide-and-conquer approach:\*\*

1. Compute powers:  $A, A^2, A^4, \dots, A^{2^{\log L}}$

- Each involves matrix multiplication ( $n \times n$ ): critical path  $O(\log n)$

- Sequential steps:  $O(\log L)$

- \*\*Depth:\*\*  $O(\log L \cdot \log n)$

2. Compute each  $A^i$  using binary representation

- Any  $A^i$  needs at most  $\log L$  matrix multiplications

- Critical path:  $O(\log L \cdot \log n)$

3. Then  $CA^iB$  for each - additional  $O(\log n)$  for matrix-vector operations

\*\*Total critical path:\*\*  $O(\log^2 L \cdot \log n)$  or  $O(\log L \cdot \log n)$  depending on how clever we are **Better now – Claude got the correct critical path, and performed some deeper analysis of the computations needed.**

\*\*Work complexity:\*\*  $O(n^3L)$  (we need all  $L$  kernel values, each involving  $n^3$  operations)

### 2.4.2 (e) Diagonal Matrix Structure

If  $A = \text{diag}(\lambda_1, \dots, \lambda_n)$ :

$$CA^\ell B = \sum_{i=1}^n C_i \lambda_i^\ell B_i$$

For each kernel element:

- Compute  $\lambda_i^\ell$  for all  $i$ : can use repeated squaring, critical path  $O(\log \ell)$

- Multiply by  $C_i B_i$  and sum: critical path  $O(\log n)$

All  $L$  kernel elements computed in parallel:

- \*\*Critical path:\*\*  $O(\log L + \log n)$

- \*\*Work:\*\*  $O(nL \log L)$  (if using repeated squaring for powers)

Actually, if we precompute all powers  $\lambda_i^1, \lambda_i^2, \dots, \lambda_i^L$  using repeated doubling:

- \*\*Critical path:\*\*  $O(\log L)$  to compute all powers **This is better now.**

- Then  $O(\log n)$  to compute each kernel element

- \*\*Total critical path:\*\*  $O(\log L + \log n)$  **This could be simplified with some more asymptotic analysis, but the reasoning is now better.**

- \*\*Work:\*\*  $O(nL)$

### 2.4.3 (f) DPLR Structure

For  $A = I_n + pp^\top$ :

My earlier analysis was trying to use the closed form  $(I + pp^\top)^\ell = I + \frac{(1+\alpha)^{\ell-1}}{\alpha} pp^\top$  where  $\alpha = p^\top p$ .

This gives:  $CA^\ell B = CB + (Cp)(p^\top B) \cdot \frac{(1+\alpha)^{\ell-1}}{\alpha}$

Computing all  $L$  kernel values:

- Precompute:  $Cp$ ,  $p^\top B$ ,  $CB$ ,  $\alpha$  - critical path  $O(\log n)$
  - For each  $\ell$ : compute  $(1+\alpha)^\ell$  - can do all in parallel using doubling, critical path  $O(\log L)$ 
    - Multiply by scalar and add:  $O(1)$  per kernel element
    - \*\*Critical path:\*\*  $O(\log n + \log L)$  \*\*Work:\*\*  $O(n + L)$
- Is this analysis correct now?

User: try thinking through the complexity of 1(c) again. is it not true that there are  $O(\log L)$  steps, and that a matrix multiplication occurs at each step? You're absolutely right! Let me reconsider this more carefully.

## 2.5 (c) Efficient Computation with Convolutions - FINAL ANSWER

\*\*If we already have kernel  $K$ .\*\*

When using convolution to compute outputs, we're computing:

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

where  $K_\ell$  are  $n \times d$  matrices,  $u_k$  are  $d$ -dimensional vectors, and  $y_k$  are  $n$ -dimensional vectors.

Using \*\*FFT-based convolution:\*\*

- FFT of input sequence:  $O(\log L)$  critical path
- Pointwise multiplication in frequency domain: At each of  $O(L)$  frequency bins, we do a matrix-vector multiplication  $K_\ell \cdot \tilde{u}_\ell$ 
  - Each matrix-vector multiplication has critical path  $O(\log d)$
  - But these  $L$  multiplications can all be done in parallel
  - So this contributes  $O(\log d)$  to critical path
  - Inverse FFT:  $O(\log L)$  critical path
  - \*\*Total critical path:  $O(\log L + \log d)$ \*\*

But wait - you're pointing out that FFT itself involves operations on vectors, not just scalars!

Actually, let me reconsider the FFT structure. FFT on a sequence of length  $L$  has:

- $O(\log L)$  stages
  - At each stage, we do pairwise operations (butterfly operations)
  - When the elements are  $d$ -dimensional vectors (or results of matrix-vector products yielding  $n$ -dimensional vectors), each butterfly operation involves vector additions
    - Vector addition of  $n$ -dimensional vectors has critical path  $O(\log n)$
- So FFT on vector-valued sequences:
- $O(\log L)$  stages
  - Each stage involves vector operations with critical path  $O(\log n)$
  - \*\*Critical path:  $O(\log L \cdot \log n)$ \*\*

Similarly, for the pointwise multiplications (matrix-vector products in frequency domain):

- Critical path  $O(\log d)$  for each multiplication
  - Done in parallel across frequency bins
  - Contributes  $O(\log d)$
- \*\*Revised total critical path for convolution:  $O(\log L \cdot \log n + \log d)$ \*\*  
 Or more simply: \*\* $O(\log L \cdot \log n)$ \*\* (assuming  $d, n$  are similar dimensions)  
 \*\*Naive recurrent approach:\*\*
- $L$  sequential time steps
  - Each step:  $x_k = Ax_{k-1} + Bu_{k-1}$  requires matrix-vector multiplication
  - Critical path per step:  $O(\log n)$
  - \*\*Total critical path:  $O(L \cdot \log n)$ \*\*
- \*\*Comparison:\*\*
- \*\*Convolution approach:\*\*  $O(\log L \cdot \log n)$
  - \*\*Naive recurrent approach:\*\*  $O(L \cdot \log n)$

The key speedup is replacing the factor of  $L$  with  $\log L$  - exponential improvement in the sequence length dimension!

Is this the correct analysis you were looking for? **This is now correct, with a lot of help.**

### 3 Problem 3

User: Solve problem 3.

I'll solve Problem 3 on Self-Supervised Linear Purification step by step.

#### 3.1 Problem 3: Self-Supervised Linear Purification

##### 3.1.1 (a) Computing losses for specific encoders

Given: -  $W^{(\alpha)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$  (identity) -  $W^{(\beta)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$  (projects to first dimension)  
 -  $X = \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix}$

##### 3.1.2 i. Computing losses

**For encoder  $\alpha$  (Identity):**

Reconstruction loss:

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

Regularization loss:

$$\|W^{(\alpha)}\|_F^2 = \text{tr}(W^{(\alpha)\top} W^{(\alpha)}) = \text{tr}(I_2) = 1 + 1 = 2$$

**For encoder  $\beta$ :**

First compute  $W^{(\beta)}X$ :

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

Reconstruction loss:

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

$$\begin{aligned} \|X - W^{(\beta)}X\|_F^2 &= 0^2 + 0.02^2 + (-0.01)^2 + 0.01^2 + (-0.02)^2 \\ &= 0.0004 + 0.0001 + 0.0001 + 0.0004 = 0.001 \end{aligned}$$

This matches the given value!

Regularization loss:

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

\*\*Summary table:\*\*

Encoder	Reconstruction Loss	Regularization Loss
0   2		
0.001   1		

### 3.1.3 ii. When does identity have higher loss?

Total loss:  $L_2(W; X, \lambda) = \text{Reconstruction} + \lambda \cdot \text{Regularization}$

For  $W^{(\alpha)}$ :  $L_2^{(\alpha)} = 0 + \lambda \cdot 2 = 2\lambda$

For  $W^{(\beta)}$ :  $L_2^{(\beta)} = 0.001 + \lambda \cdot 1 = 0.001 + \lambda$

We want  $L_2^{(\alpha)} > L_2^{(\beta)}$ :

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

$$\lambda > 0.001$$

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

---

### 3.1.4 (b) Generic square linear encoder

Given:  $X = U\Sigma V^\top$  with singular values  $\sigma_1 > \dots > \sigma_m \geq 0$

### 3.1.5 i. Fill in the optimal weight matrix

The optimal weight matrix is:

$$\hat{W} = U \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & & \\ & \frac{\sigma_2^2}{\sigma_2^2 + \lambda} & & \\ & & \ddots & \\ & & & \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \end{bmatrix} U^\top$$

### 3.1.6 ii. Derive the expression

\*\*Key insight:\*\* Transform to SVD coordinates where the problem decouples.

Let  $\tilde{W} = U^\top WU$ . Then using the hint that Frobenius norm is invariant under orthogonal transformations:

$$\|X - WX\|_F^2 = \|U\Sigma V^\top - WU\Sigma V^\top\|_F^2$$

Multiply on left by  $U^\top$  and right by  $V$ :

$$= \|\Sigma - U^\top WU\Sigma\|_F^2 = \|\Sigma - \tilde{W}\Sigma\|_F^2$$

Similarly:

$$\|W\|_F^2 = \|U^\top WU\|_F^2 = \|\tilde{W}\|_F^2$$

So the objective becomes:

$$L_2(\tilde{W}; \Sigma, \lambda) = \|\Sigma - \tilde{W}\Sigma\|_F^2 + \lambda \|\tilde{W}\|_F^2$$

Since  $\Sigma$  is diagonal with entries  $\sigma_i$ , let  $\tilde{W} = [\tilde{w}_{ij}]$ . Then:

$$\Sigma - \tilde{W}\Sigma = \begin{bmatrix} \sigma_1(1 - \tilde{w}_{11}) & -\tilde{w}_{12}\sigma_2 & \cdots \\ -\tilde{w}_{21}\sigma_1 & \sigma_2(1 - \tilde{w}_{22}) & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

The Frobenius norm squared is:

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

And:

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

The objective decouples! For off-diagonal terms ( $i \neq j$ ):

$$\min_{\tilde{w}_{ij}} [\sigma_j^2 \tilde{w}_{ij}^2 + \lambda \tilde{w}_{ij}^2] = \min_{\tilde{w}_{ij}} [(\sigma_j^2 + \lambda) \tilde{w}_{ij}^2]$$

Clearly  $\tilde{w}_{ij}^* = 0$  for  $i \neq j$ .

For diagonal terms:

$$\min_{\tilde{w}_{ii}} [\sigma_i^2(1 - \tilde{w}_{ii})^2 + \lambda \tilde{w}_{ii}^2]$$

Taking derivative and setting to zero:

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

$$\sigma_i^2 = \sigma_i^2 \tilde{w}_{ii} + \lambda \tilde{w}_{ii}$$

$$\tilde{w}_{ii}^* = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}$$

Therefore:

$$\tilde{W}^* = \text{diag} \left( \frac{\sigma_1^2}{\sigma_1^2 + \lambda}, \dots, \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \right)$$

And transforming back:

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

### 3.1.7 (c) Finding the range of $\lambda$

Given singular values:  $\{10, 8, 4, 1, 0.5, 0.36, 0.16, 0.01\}$

Requirements: 1. \*\*Preserve 802. \*\*Attenuate 50

\*\*Critical singular values:\*\*  $\sigma_3 = 4$  and  $\sigma_4 = 1$

\*\*Condition 1:\*\* For  $\sigma_3 = 4$ :

$$\frac{16}{16 + \lambda} \geq 0.8$$

$$16 \geq 0.8(16 + \lambda)$$

$$16 \geq 12.8 + 0.8\lambda$$

$$3.2 \geq 0.8\lambda$$

$$\lambda \leq 4$$

\*\*Condition 2:\*\* For  $\sigma_4 = 1$ :

$$\frac{1}{1 + \lambda} \leq 0.5$$

$$1 \leq 0.5(1 + \lambda)$$

$$2 \leq 1 + \lambda$$

$$\lambda \geq 1$$

\*\*Answer:\*\*  $1 \leq \lambda \leq 4$

This ensures the purifier preserves the top 3 components (signal) while attenuating the bottom 5 components (noise).

This section is completely correct, with some highlighted key insights from Claude that help with clarity + glean takeaways from the problem. The approach to (b) is slightly different from the key, but seems logically sound.

## 4 Problem 4

User: Solve problem 4.

I'll solve Problem 4 on Ridge-Attention step by step.

Problem 4: Ridge-Attention

(a) Efficient updating of simple averaging

Given:  $m = \frac{1}{n} \sum_{i=1}^n x_i$  (average of  $n$  points)

Want:  $m' = \frac{1}{n+1} \sum_{i=1}^{n+1} x_i$  (average including new point  $x_{n+1}$ )

\*\*Solution:\*\*

Start by multiplying  $m$  by  $n$ :

$$nm = \sum_{i=1}^n x_i$$

Then:

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

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

This allows us to update the average with just one addition and one division, without recomputing the entire sum!

This is entirely correct.

(b) Writing ridge regression terms as sums

Given:  $w^* = (A^\top A + \lambda I)^{-1} A^\top y$  where  $A = \begin{bmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_n^\top \end{bmatrix}$  and  $y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$

\*\*Computing  $A^\top A$ :\*\*

$$A^\top A = [x_1 \ x_2 \ \cdots \ x_n] \begin{bmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_n^\top \end{bmatrix} = \sum_{i=1}^n x_i x_i^\top$$

\*\*Computing  $A^\top y$ :\*\*

$$A^\top y = [x_1 \ x_2 \ \cdots \ x_n] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^n y_i x_i$$

\*\*Answer:\*\*

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

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

This is entirely correct.

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

In non-causal ridge self-attention with context length  $n$  and  $d$ -dimensional query, key, value vectors:

\*\*Setup:\*\* - Keys:  $K_1, \dots, K_n$  (each  $d$ -dimensional) - Values:  $V_1, \dots, V_n$  (each  $d$ -dimensional) - Queries:  $Q_1, \dots, Q_n$  (each  $d$ -dimensional)

For multi-dimensional ridge regression, we need to solve for each query  $Q_i$ :

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

$$\text{where } K = \begin{bmatrix} K_1^\top \\ \vdots \\ K_n^\top \end{bmatrix} \text{ (dimension } n \times d\text{)} \text{ and } V = \begin{bmatrix} V_1^\top \\ \vdots \\ V_n^\top \end{bmatrix} \text{ (dimension } n \times d\text{),}$$

treating each column as a separate regression problem).

\*\*Cost breakdown:\*\*

1. \*\*Compute  $K^\top K$ :\*\* - This is  $d \times d$  matrix - Cost:  $O(nd^2)$  (summing  $n$  outer products of  $d$ -dimensional vectors)

2. \*\*Add  $\lambda I$ :\*\*  $O(d)$

3. \*\*Invert  $(K^\top K + \lambda I)$ :\*\* - Inverting  $d \times d$  matrix:  $O(d^3)$

4. \*\*Compute  $K^\top V$ :\*\* - This is  $d \times d$  matrix (treating all  $d$  dimensions of values together) - Cost:  $O(nd^2)$

5. \*\*Compute  $(K^\top K + \lambda I)^{-1} K^\top V$ :\*\* - Matrix multiply:  $d \times d$  times  $d \times d$ :  $O(d^3)$

6. \*\*For each query  $Q_i$ , compute output:\*\* - We have the weight matrix  $W^* \in \mathbb{R}^{d \times d}$  - Output for query  $i$ :  $W^* Q_i$  costs  $O(d^2)$  - For all  $n$  queries:  $O(nd^2)$

\*\*Total cost:\*\*  $O(nd^2 + d^3 + nd^2 + d^3 + nd^2) = O(nd^2 + d^3)$

Since we assume  $d < n$ , we have  $nd^2 > d^3$ , so:

\*\*Answer:\*\*  $O(nd^2)$

This is entirely correct.

—  
(d) What receives gradients in backprop?

In ridge self-attention, the forward pass involves: - Computing  $(K^\top K + \lambda I)^{-1} K^\top V \cdot Q$

This is a differentiable function of  $K$ ,  $V$ ,  $Q$ , and  $\lambda$ .

\*\*Answer:\*\* All of the following receive gradients: - The ridge  $\lambda$  viewed as a learnable parameter - The keys - The values - The queries

All are part of the computational graph and backprop will successfully pass gradients through the matrix inverse and multiplications.

This is entirely correct.

—  
(e) Cost of Sherman-Morrison Formula application

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

Assuming we already have  $M^{-1}$  (a  $d \times d$  matrix), compute the cost:

\*\*Operations:\*\* 1. Compute  $M^{-1}u$ : matrix-vector multiply,  $O(d^2)$  2. Compute  $v^\top M^{-1}$ : vector-matrix multiply,  $O(d^2)$  3. Compute  $v^\top M^{-1}u$ : inner product (reusing results from steps 1-2),  $O(d)$  4. Compute  $1 + v^\top M^{-1}u$ :  $O(1)$  5. Compute  $\frac{1}{1+v^\top M^{-1}u}$ : scalar division,  $O(1)$  6. Compute  $(M^{-1}u)(v^\top M^{-1})$ : outer product (dyad),  $O(d^2)$  7. Multiply by scalar from step 5:  $O(d^2)$  8. Subtract from  $M^{-1}$ :  $O(d^2)$

\*\*Total cost:\*\*  $O(d^2 + d^2 + d + d^2) = O(d^2)$

\*\*Answer:\*\*  $O(d^2)$

This is entirely correct, but could provide more detail with more explicit use of formulae.

—  
(f) Efficient causal ridge-self-attention

\*\*Key insight:\*\* At position  $t$ , we only use keys and values from positions  $1, \dots, t$ . We can maintain running sums and update incrementally using Sherman-Morrison.

\*\*Algorithm:\*\*

\*\*Initialize:\*\* -  $M_0 = \lambda I$  (the  $d \times d$  matrix before seeing any data) -  $M_0^{-1} = \frac{1}{\lambda} I$  -  $S_0 = 0$  (the  $d \times d$  matrix  $K^\top V = \sum_{i=1}^0 V_i K_i^\top$ )

\*\*For each position  $t = 1, 2, \dots, n$ :

1. \*\*Update the Gram matrix using Sherman-Morrison:\*\* - We need  $M_t = M_{t-1} + K_t K_t^\top$  (a rank-1 update!) - Apply Sherman-Morrison with  $u = v = K_t$ :

$$M_t^{-1} = M_{t-1}^{-1} - \frac{1}{1 + K_t^\top M_{t-1}^{-1} K_t} (M_{t-1}^{-1} K_t)(K_t^\top M_{t-1}^{-1})$$

- Cost:  $O(d^2)$  per step

2. \*\*Update the sum:\*\*

$$S_t = S_{t-1} + V_t K_t^\top$$

- Cost:  $O(d^2)$  per step

3. \*\*Compute the output for query  $Q_t$ :\*\*

$$\text{Output}_t = M_t^{-1} S_t Q_t$$

- First compute  $S_t Q_t$ :  $O(d^2)$  - Then compute  $M_t^{-1}(S_t Q_t)$ :  $O(d^2)$  - Total:  $O(d^2)$

\*\*Total complexity:\*\* - Per position:  $O(d^2)$  - For all  $n$  positions:  $O(nd^2)$  - Critical path:  $O(n \cdot d^2)$  in depth since we must process sequentially in time, but each step is  $O(d^2)$  depth

Wait, let me reconsider the critical path. Each position requires: - Sherman-Morrison update:  $O(d^2)$  work, but critical path is  $O(\log d)$  for the matrix-vector operations - So critical path per step:  $O(\log d)$  - Total critical path:  $O(n \log d)$  (sequential in time)

\*\*What to track:\*\* -  $M_t^{-1}$  (the inverse of the Gram matrix up to position  $t$ ) -  $S_t$  (the sum  $\sum_{i=1}^t V_i K_i^\top$ )

This avoids recomputing everything from scratch at each position!

This is entirely correct.

---

(g) Computing attention weights for ridge-attention

For a given query  $Q$  and ridge-attention output  $\text{Output} = (K^\top K + \lambda I)^{-1} K^\top V \cdot Q$ , we want weights  $w_1, \dots, w_n$  such that:

$$\text{Output} = \sum_{i=1}^n w_i V_i$$

\*\*Solution:\*\*

Let  $W^* = (K^\top K + \lambda I)^{-1} K^\top V$  and the output is  $W^* Q$ .

We can rewrite:

$$(K^\top K + \lambda I)^{-1} K^\top V \cdot Q = (K^\top K + \lambda I)^{-1} \sum_{i=1}^n (K_i^\top Q) V_i$$

Let  $\alpha = (K^\top K + \lambda I)^{-1} K^\top Q$  (a  $d$ -dimensional vector). Then:

$$\text{Output} = \sum_{i=1}^n V_i^\top \alpha$$

So the weight for  $V_i$  is:

$$w_i = K_i^\top (K^\top K + \lambda I)^{-1} Q$$

Alternatively, in matrix form, if we define  $\mathbf{w} = [w_1, \dots, w_n]^\top$ :

$$\mathbf{w} = K(K^\top K + \lambda I)^{-1} Q$$

**\*\*Interpretation:\*\*** These weights can be negative (unlike softmax attention) and they tell us how much each value vector contributes to the final output. They are analogous to the softmax weights in standard attention.

**\*\*Answer:\*\*** The attention weights are:

$$w_i = K_i^\top (K^\top K + \lambda I)^{-1} Q$$

or in vector form:  $\mathbf{w} = K(K^\top K + \lambda I)^{-1} Q$

This is entirely correct, but the explanation is slightly underdetailed.