

### 1. SSM Convolution Kernel

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

$$\begin{aligned} \dot{x}_k &= Ax_k + Bu_k \\ y_k &= Cx_k + Du_k \end{aligned}$$

(a) Convolution Kernel and Output Equation: Given that the sequence length is 2, input  $\{u_0, \dots, u_2\}$ , output  $\{y_0, \dots, y_2\}$  and assume  $n_x = 3$ , show that the output  $y_k$  can be expressed as a convolution of the input sequence  $\{u_i\}_{i=0}^2$  with kernel  $K$ :

$$y_k = \sum_{j=0}^{k-1} K_{k-j} u_j$$

where any  $u_{i,j}$  with a negative index is set to 0 (zero-padding). Note that  $K$

I'm working on a deep learning assignment. Please explain your reasoning step by step instead of just giving me the answer.

HW 9

Gemini 3 pro

Show thinking ▾

This derivation connects the state-space representation (recurrence) to the convolutional representation (parallelizable). This is a foundational concept in understanding how modern architectures like Mamba or Linear Recurrent Units (LRUs) work efficiently.

Here is the step-by-step reasoning to derive the convolution form and find the kernel  $K$ .

#### Step 1: "Unroll" the Recursion

To understand the pattern, it is best to calculate the first few time steps manually, starting from  $k = 0$ .

Given:

- 1.  $x_{k+1} = Ax_k + Bu_k$
- 2.  $y_k = Cx_k + Du_k$
- 3.  $x_0 = 0$  (Initial state is zero)

Time step  $k = 0$ :

- State:  $x_0 = 0$
- Output:

$$y_0 = C(0) + Du_0 = \mathbf{Du}_0$$

Time step  $k = 1$ :

- State: We need  $x_1$ . Using the update rule:  $x_1 = Ax_0 + Bu_0 = A(0) + Bu_0 = Bu_0$ .
- Output:

$$y_1 = Cx_1 + Du_1 = C(Bu_0) + Du_1 = \mathbf{CBu}_0 + \mathbf{Du}_1$$

Time step  $k = 2$ :

- State: We need  $x_2$ . Using the update rule:  $x_2 = Ax_1 + Bu_1$ . Substitute  $x_1 = Bu_0$ :

$$x_2 = A(Bu_0) + Bu_1 = ABu_0 + Bu_1$$

#### • Output:

$$y_2 = Cx_2 + Du_2 = C(ABu_0 + Bu_1) + Du_2$$

$$y_2 = \mathbf{CABu}_0 + \mathbf{CBu}_1 + \mathbf{Du}_2$$

Time step  $k = 3$ :

- State:  $x_3 = Ax_2 + Bu_2 = A(ABu_0 + Bu_1) + Bu_2 = A^2Bu_0 + ABu_1 + Bu_2$ .
- Output:

$$y_3 = \mathbf{CA}^2\mathbf{Bu}_0 + \mathbf{CABu}_1 + \mathbf{CBu}_2 + \mathbf{Du}_3$$

#### Step 2: Identify the Pattern

Let's look at the result for  $y_3$  and identify the coefficients for each input  $u$ :

$$y_3 = (CA^2B)u_0 + (CAB)u_1 + (CB)u_2 + (D)u_3$$

Notice that  $y_k$  is a weighted sum of current and past inputs  $u$ . If we try to write  $y_k$  generally based on this pattern:

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

Here,  $D$  acts on the current input, and terms like  $CA^nB$  act on previous inputs.

#### Step 3: Convert to Convolution Notation

The problem asks us to express this as a convolution:

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

In a convolution, we iterate over a "lag" or "shift" variable  $\ell$ .

- When  $\ell = 0$ , we look at  $u_{k-0} = u_k$ .
- When  $\ell = 1$ , we look at  $u_{k-1}$ .
- When  $\ell = 2$ , we look at  $u_{k-2}$ .

Let's rearrange our pattern from Step 2 to group by the "lag" from the current time  $k$ .

Looking at  $y_3$  again:

$$y_3 = \underbrace{D}_{Lag=0} u_3 + \underbrace{CB}_{Lag=1} u_2 + \underbrace{CAB}_{Lag=2} u_1 + \underbrace{CA^2B}_{Lag=3} u_0$$

We can map the lag  $\ell$  to the coefficients:

- $\ell = 0$  (term  $u_k$ ): Coefficient is  $D$ .

We can map the lag  $\ell$  to the coefficients:

- $\ell = 0$  (term  $u_k$ ): Coefficient is  $D$ .
- $\ell = 1$  (term  $u_{k-1}$ ): Coefficient is  $CB$ .
- $\ell = 2$  (term  $u_{k-2}$ ): Coefficient is  $CAB$ .
- $\ell = 3$  (term  $u_{k-3}$ ): Coefficient is  $CA^2B$ .
- ...
- General  $\ell$  (term  $u_{k-\ell}$ ): Coefficient is  $CA^{\ell-1}B$ .

#### Step 4: Define the Kernel $K$

Based on the mapping in Step 3, we can define the kernel  $K = \{K_\ell\}_0^L$  piecewise.

The term for  $\ell = 0$  is distinct because it comes from the direct feedthrough matrix  $D$ . The terms for  $\ell > 0$  come from the state recurrence and involve powers of  $A$ .

The Kernel  $K$  is:

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

#### Final Verification

Let's plug this kernel back into the convolution sum to ensure it matches the problem statement.

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

Separate the  $\ell = 0$  term from the rest:

$$y_k = K_0 u_k + \sum_{\ell=1}^L K_\ell u_{k-\ell}$$

Substitute our found  $K$ :

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

Note regarding the upper limit  $L$ : The problem states that  $u$  with a negative index is 0. If  $\ell > k$ , then the index  $k - \ell$  is negative, so  $u_{k-\ell} = 0$ . This means the sum effectively stops at  $\ell = k$ , which matches our derivation in Step 2 exactly.

**Solution Summary:** The output  $y_k$  is the convolution of the input  $u$  and the kernel  $K$ , where  $K$  is defined as:

$$K = \{D, CB, CAB, CA^2B, \dots, CA^{L-1}B\}$$

excellent structure

Gemini correctly identified the recursive expansion and the mapping to a discrete convolution.

(b) Concrete Examples:  
1. Scalar Case: Let  $\alpha = 1$ , and set  $A = \alpha$ ,  $B = \beta$ ,  $C = \gamma$ ,  $D = \delta$ . Use  $\alpha = 0.8$ ,  $\beta = 1$ ,  $\gamma = 1.5$ , and  $\delta = 0$ . Let  $u = [1, 0, 0, 0]^T$  be the input vector.  
 $A = \begin{pmatrix} 0.8 & 0 \\ 0 & 0.8 \end{pmatrix}$ ,  $B = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ ,  $C = \begin{pmatrix} 1 & 0 \end{pmatrix}$ ,  $D = 0$ .  
Compute kernels up to  $L = 4$  and briefly discuss how the kernel expresses the "impulse response".



Based on the derivation from Part (a), we established that the kernel  $K$  is defined as:

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

Let's compute the concrete examples step by step.

#### i. Scalar Case ( $n = 1$ )

Given:

- $A = \alpha = 0.8$
- $B = \beta = 1$
- $C = \gamma = 1.5$
- $D = \delta = 0$  (No numerical value is provided in the prompt, so we treat it as a variable).
- Target: Compute  $K$  up to  $L = 4$  (meaning  $\ell = 0, 1, 2, 3, 4$ ).

Calculations:

1.  $\ell = 0$ :

$$K_0 = D = \delta$$

2.  $\ell = 1$ :

$$K_1 = CB = \gamma \cdot \beta = 1.5 \cdot 1 = \mathbf{1.5}$$

3.  $\ell = 2$ :

$$K_2 = CAB = \gamma \cdot \alpha \cdot \beta = 1.5 \cdot 0.8 \cdot 1 = \mathbf{1.2}$$

4.  $\ell = 3$ :

$$K_3 = CA^2B = \gamma \cdot \alpha^2 \cdot \beta = 1.5 \cdot (0.8)^2 \cdot 1$$

$$K_3 = 1.5 \cdot 0.64 = \mathbf{0.96}$$

5.  $\ell = 4$ :

$$K_4 = CA^3B = \gamma \cdot \alpha^3 \cdot \beta = 1.5 \cdot (0.8)^3 \cdot 1$$

$$K_4 = 1.5 \cdot 0.512 = \mathbf{0.768}$$

$$K_4 = CA^3B = \gamma \cdot \alpha^3 \cdot \beta = 1.5 \cdot (0.8)^3 \cdot 1$$

$$K_4 = 1.5 \cdot 0.512 = \mathbf{0.768}$$

Summary for Scalar Case:

$$K = \{\delta, 1.5, 1.2, 0.96, 0.768\}$$

(Note: You can observe the geometric decay  $1.5 \times 0.8^{\ell-1}$  for  $\ell \geq 1$ .)

ii. 2D Case ( $n = 2$ )

Given:

- $A = \begin{pmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{pmatrix}$
- $B = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
- $C = \begin{pmatrix} 1 & 0 \end{pmatrix}$
- $D = 0$
- Target: Compute  $K$  up to  $L = 3$ .

Calculations:

1.  $\ell = 0$ :

$$K_0 = D = \mathbf{0}$$

2.  $\ell = 1$ :

$$K_1 = CB = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1(1) + 0(0) = \mathbf{1}$$

3.  $\ell = 2$ :

$$K_2 = CAB$$

First, compute  $AB$  (which extracts the first column of  $A$  because  $B$  is a standard basis vector):

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

Now multiply by  $C$ :

$$K_2 = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0.7 \\ 0.2 \end{pmatrix} = \mathbf{0.7}$$

4.  $\ell = 3$ :

$$K_3 = CA^2B = CA(AB)$$

b-ii

Matrix multiplication and ordering are all correct

Gemini caught that  $AB$  just picks out  $A$ 's first column

Interpretation correct and conceptually clear.

Gemini connected SSM dynamics to signal - processing

(b-ii)

values match the official solution exactly; Gemini noticed the geometric decay  $1.5 \times 0.8^{\ell-1}$

but lack of  $1.5 \times 0.8^4$  solution.

First, compute  $AB$  (which extracts the first column of  $A$  because  $B$  is a standard basis vector):

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

Now multiply by  $C$ :

$$K_2 = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0.7 \\ 0.2 \end{pmatrix} = \mathbf{0.7}$$

4.  $\ell = 3$ :

$$K_3 = CA^2B = CA(AB)$$

We already calculated  $AB$  in the previous step. We just need to multiply  $A$  by that result again.

$$A(AB) = \begin{pmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{pmatrix} \begin{pmatrix} 0.7 \\ 0.2 \end{pmatrix}$$

Row 1:  $(0.7 \times 0.7) + (0.1 \times 0.2) = 0.49 + 0.02 = 0.51$  Row 2:  $(0.2 \times 0.7) + (0.6 \times 0.2) = 0.14 + 0.12 = 0.26$  So,  $A^2B = \begin{pmatrix} 0.51 \\ 0.26 \end{pmatrix}$  Now multiply by  $C$ :

$$K_3 = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0.51 \\ 0.26 \end{pmatrix} = \mathbf{0.51}$$

Summary for 2D Case:

$$K = \{0, 1, 0.7, 0.51\}$$

Discussion: How the Kernel Captures "Impulse Response"

In signal processing, the **impulse response** of a system is its output when the input is an "impulse" (a Kronecker delta:  $u_0 = 1$  and all other  $u_k = 0$ ).

If we plug this specific input into our convolution formula  $y_k = \sum_{\ell=0}^k K_\ell u_{k-\ell}$ :

- At  $k = 0$ :  $y_0 = K_0 u_0 = K_0(1) = K_0$
- At  $k = 1$ :  $y_1 = K_1 u_0 + K_0 u_1 = K_1(1) + K_0(0) = K_1$
- At  $k = 2$ :  $y_2 = K_2 u_0 + K_1 u_1 + K_0 u_2 = K_2(1) + 0 + 0 = K_2$

Conclusion: The kernel sequence  $K = \{K_0, K_1, K_2, \dots\}$  is exactly the **impulse response** of the State-Space Model. It represents how the system "remembers" a momentary input spike as it propagates forward in time, decaying according to the eigenvalues of the matrix  $A$ .

### (c) Efficient Computation with Convolutions

This section highlights the fundamental efficiency advantage of SSMs (like Mamba) that can be viewed as convolutions.

#### 1. Parallelization of the Output Sequence:

- **Convolution View:** If we know the kernel  $K$ , the calculation of any specific output  $y_k$  depends *only* on the inputs  $u_0, \dots, u_k$  and the kernel weights. It does **not** depend on the previous output  $y_{k-1}$  or state  $x_{k-1}$ .
- **Result:** We can compute every element of the output sequence  $\{y_0, \dots, y_L\}$  **simultaneously (in parallel)**. There is no sequential dependency between time steps in the convolution form.

#### 2. Minimum Critical Path Length:

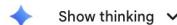
- **Convolution:** To compute  $y_k$ , we perform a dot product of size roughly  $k$  (or  $L$ ). Using a parallel reduction tree (summing numbers in pairs), this takes logarithmic time. Alternatively, using the Fast Fourier Transform (FFT) theorem ( $y = \text{IFFT}(\text{FFT}(u) \cdot \text{FFT}(K))$ ), the entire sequence can be computed with a critical path dominated by the FFT steps.
- **Critical Path:**  $O(\log L)$  ✓

#### 3. Comparison with Naive, Direct Computation:

- **Recurrent View (Naive):** The unrolled recursion is  $x_{k+1} = Ax_k + Bu_k$ . To compute  $x_k$ , you *must* wait for  $x_{k-1}$  to be finished. To get  $x_{k-1}$ , you wait for  $x_{k-2}$ , and so on.
- **Result:** This forces a strictly sequential execution. You cannot calculate the end of the sequence until you have stepped through the entire history.
- **Critical Path:**  $O(L)$  (Linear time). ✓

(d) **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.

correct, but  
forgot to multiply  
 $O(\log N)$  for the  
parallelize matrix  
multiplication.



Here is the step-by-step reasoning for Part (d): Efficient Kernel Computation.

#### The Challenge: Sequential Dependency

The naive way to compute the kernel  $K = \{D, CB, CAB, CA^2B, \dots, CA^{L-1}B\}$  is to multiply by  $A$  iteratively:

1. Start with  $x = B$ .
2. Compute  $Ax$ .
3. Compute  $A(Ax)$ .
4. ... and so on.

This is a **serial process** with  $O(L)$  steps. To utilize modern hardware (GPUs/TPUs), we need to parallelize this.

#### Strategy 1: Recursive Doubling (Divide and Conquer)

This strategy directly uses the hint that  $L = 2^N$ . We can compute the powers of  $A$  using a "doubling" approach rather than one by one.

**Reasoning:** If we have computed the sequence of powers for length  $L/2$ , we can reuse it to compute the second half.

Let  $P_k$  be the sequence of powers  $\{A^0, A^1, \dots, A^{k-1}\}$ . To find  $P_{2k}$  (length  $2k$ ):

### Strategy 1: Recursive Doubling (Divide and Conquer)

This strategy directly uses the hint that  $L = 2^N$ . We can compute the powers of  $A$  using a "doubling" approach rather than one by one.

**Reasoning:** If we have computed the sequence of powers for length  $L/2$ , we can reuse it to compute the second half.

Let  $P_k$  be the sequence of powers  $\{A^0, A^1, \dots, A^{k-1}\}$ . To find  $P_{2k}$  (length  $2k$ ):

1. Take the existing sequence  $P_k$ .
2. Take a copy of  $P_k$  and multiply every element by  $A^k$ .
3. Concatenate them:  $P_{2k} = P_k \cup (A^k \cdot P_k)$ .

**Algorithm:**

1. **Initialize:** List =  $\{I\}$ . (Length 1)
2. **Step 1:** Multiply List by  $A^1 \rightarrow \{A\}$ . Concatenate  $\rightarrow \{I, A\}$ . (Length 2)
3. **Step 2:** Multiply List by  $A^2 \rightarrow \{A^2, A^3\}$ . Concatenate  $\rightarrow \{I, A, A^2, A^3\}$ . (Length 4)
4. **Step 3:** Multiply List by  $A^4 \rightarrow \{A^4, \dots, A^7\}$ . Concatenate  $\rightarrow \{I, \dots, A^7\}$ . (Length 8)
5. Repeat  $N$  times until length is  $L = 2^N$ .

**Efficiency:**

- **Depth (Critical Path):** We only need  $N = \log_2 L$  steps of matrix multiplication.
- **Parallelism:** The multiplication of the list by the scalar  $A^k$  can be done for all elements in parallel.

### Strategy 2: Diagonalization (Structural Efficiency)

This is the most common strategy used in modern efficient SSMs (like S4).

**Reasoning:** Matrix multiplication is expensive ( $O(n^3)$  or  $O(n^2)$ ). If we can make  $A$  diagonal, matrix multiplication becomes scalar multiplication.

Assume  $A$  is diagonalizable ( $A = V\Lambda V^{-1}$ ). The kernel term becomes:

$$K_\ell = CA^{\ell-1}B = C(V\Lambda^{\ell-1}V^{-1})B$$

Let  $\tilde{C} = CV$  and  $\tilde{B} = V^{-1}B$ .

$$K_\ell = \tilde{C}\Lambda^{\ell-1}\tilde{B} = \sum_{j=1}^n \tilde{c}_j(\lambda_j)^{\ell-1}\tilde{b}_j$$

**Efficiency:**

- Instead of computing matrix powers, we effectively compute **scalar powers**  $\lambda_j^{\ell-1}$ .
- **Fully Parallel:** We can compute the value for every time step  $\ell \in \{0, \dots, L\}$  simultaneously. There is no dependency between  $K_\ell$  and  $K_{\ell+1}$ .
- **Complexity:**  $O(1)$  parallel depth.

Gemini  $\Rightarrow$  pro  
gives  $\Rightarrow$  strategy  
the strategy I  
is same as the answer.

### Strategy 3: Fast Fourier Transform (FFT)

We can also compute the kernel in the frequency domain.

**Reasoning:** The sequence of powers corresponds to the impulse response of the system. By the convolution theorem, we can compute this via the Transfer Function  $H(z)$ .

$$H(z) = C(zI - A)^{-1}B + D$$

**Algorithm:**

1. Evaluate the transfer function  $H(z)$  at  $L$  roots of unity (this can be done in parallel).
2. Apply the **Inverse Fast Fourier Transform (IFFT)** to the result to get the time-domain kernel  $K$ .

**Efficiency:**

- **Depth:**  $O(\log L)$  due to the FFT algorithm.
- **Total Operations:**  $O(L \log L)$ .

### Summary of Solution for (d)

To compute  $K$  efficiently given  $L = 2^N$ :

1. **Structure the Matrix:** If possible, diagonalize  $A$ . This reduces the problem to calculating geometric sequences of scalars, which can be done instantly in parallel for all  $L$ .
2. **Recursive Doubling:** If  $A$  cannot be diagonalized, use the divide-and-conquer strategy. Compute the first half of the powers, compute  $A^{L/2}$ , and multiply the first half by this factor to generate the second half. This requires only  $O(\log L)$  sequential steps.

### (e) Adding Structure to $A$ (Diagonal Case)

**Problem:** Suppose  $A$  is a diagonal matrix. How can we leverage this to compute  $K$  efficiently?

**1. Analyze the Matrix Powers:** If  $A$  is diagonal, it can be written as:

$$A = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$$

Raising a diagonal matrix to a power  $k$  is simply raising each diagonal element to that power:

$$A^k = \text{diag}(\lambda_1^k, \lambda_2^k, \dots, \lambda_n^k)$$

**2. Substitute into the Kernel Definition:** Recall the kernel term for  $\ell > 0$  is  $K_\ell = CA^{\ell-1}B$ .

Let's define the vectors explicitly:

- $C = [c_1 \ c_2 \ \dots \ c_n]$
- $B = [b_1 \ b_2 \ \dots \ b_n]^\top$

The product  $A^{\ell-1}B$  acts element-wise:

$$A^{\ell-1}B = \begin{bmatrix} \lambda_1^{\ell-1}b_1 \\ \lambda_2^{\ell-1}b_2 \\ \vdots \\ \lambda_n^{\ell-1}b_n \end{bmatrix}$$

Multiplying by  $C$  (the dot product) gives:

$$K_\ell = \sum_{j=1}^n c_j \lambda_j^{\ell-1} b_j$$

### 3. Efficiency Gains:

- **Scalar Reduction:** We transformed a matrix-vector multiplication (cost  $O(n^2)$ ) into a vector dot product (cost  $O(n)$ ).
- **Parallelism:** Each term in the sum is just a geometric sequence  $g_j(\ell) = (c_j b_j) \cdot \lambda_j^{\ell-1}$ . We can compute these sequences independently and in parallel for all  $j$ .

### 3. Efficiency Gains:

- **Scalar Reduction:** We transformed a matrix-vector multiplication (cost  $O(n^2)$ ) into a vector dot product (cost  $O(n)$ ).
- **Parallelism:** Each term in the sum is just a geometric sequence  $g_j(\ell) = (c_j b_j) \cdot \lambda_j^{\ell-1}$ . We can compute these sequences independently and in parallel for all  $j$ .
- **Total Complexity:** Computing the full kernel  $K$  of length  $L$  takes  $O(nL)$  operations, which is significantly faster than the naive  $O(n^2L)$ .

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

**Problem:** Suppose  $A = I_n + pp^\top$ , where  $p \in \mathbb{R}^n$ . How can we leverage this?

**1. Analyze the Structure of  $A$ :** The matrix  $A$  is a rank-1 update to the identity matrix. To compute the kernel, we need a closed form for  $A^k$ . Let's look at the eigenvalues and eigenvectors of  $A$ :

- **Eigenvector 1:** The vector  $p$  itself.

$$Ap = (I + pp^\top)p = p + p(p^\top p) = p(1 + \|p\|^2)$$

So,  $p$  is an eigenvector with eigenvalue  $\lambda_1 = 1 + \|p\|^2$ .

- **Other Eigenvectors:** Any vector  $v$  orthogonal to  $p$  (i.e.,  $p^\top v = 0$ ).

$$Av = (I + pp^\top)v = v + p(0) = v$$

These vectors have eigenvalue  $\lambda_{2\dots n} = 1$ .

**2. Derive powers  $A^k$ :** Because we know how  $A$  acts on  $p$  and vectors perpendicular to  $p$ , we can decompose any vector  $x$  into a component parallel to  $p$  and a component orthogonal to  $p$ . This leads to the closed-form expansion for the matrix power:

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

(Note: You can verify this for  $k = 1$ , which returns  $I + pp^\top$ . For  $k = 0$ , it returns  $I$ .)

Gemini 3 provides correct answers, but not emphasizes the critical path =  $\log L$ .

**3. Substitute into the Kernel Definition:** We need  $K_\ell = CA^{\ell-1}B$ . Let scalar  $\alpha = \|p\|^2$  and scalar factor  $s_\ell = \frac{(1+\alpha)^{\ell-1}-1}{\alpha}$ .

Substitute  $A^{\ell-1} = I + s_\ell pp^\top$  into the kernel equation:

$$K_\ell = C(I + s_\ell pp^\top)B$$

$$K_\ell = CB + s_\ell(Cp)(p^\top B)$$

**4. Efficiency Gains:** Notice that  $CB$ ,  $Cp$ , and  $p^\top B$  are all just **scalars**.

- **Pre-computation ( $O(n)$ ):**

1. Compute scalar  $w = C \cdot B$  (dot product).
2. Compute scalar  $u = C \cdot p$  (dot product).
3. Compute scalar  $v = p \cdot B$  (dot product).
4. Compute scalar  $\alpha = p \cdot p$ .

- **Kernel Generation ( $O(1)$  per step):**

$$K_\ell = w + \left( \frac{(1 + \alpha)^{\ell-1} - 1}{\alpha} \right) uv$$

**Conclusion:** By leveraging the DPLR structure, we reduced the complexity of computing each kernel element from  $O(n^2)$  (matrix multiplication) to  $O(1)$  (constant time scalar arithmetic) after a linear time setup. This allows the model to handle very large state dimensions  $n$  without slowdowns.

Show thinking ▾

Here are the step-by-step solutions for **Part (a)** of the Self-Supervised Linear Purification problem.

### i. Computing Reconstruction and Regularization Losses

We need to fill in the table for the two encoders,  $W^{(\alpha)}$  (Identity) and  $W^{(\beta)}$  (Projection).

#### 1. Analyze the Loss Definitions

From the problem description:

- **Reconstruction Loss:**  $\|\mathbf{X} - W\mathbf{X}\|_F^2$
- **Regularization Loss:**  $\lambda\|W\|_F^2$  (as indicated by the brace in Equation 2)

#### 2. Encoder $\alpha$ (Identity Matrix)

$$W^{(\alpha)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

- **Reconstruction Loss:** Since  $W^{(\alpha)} = I$ , the reconstruction is perfect:  $W^{(\alpha)}\mathbf{X} = \mathbf{X}$ .

$$\|\mathbf{X} - I\mathbf{X}\|_F^2 = \|\mathbf{0}\|_F^2 = 0$$

- **Regularization Term:** We calculate the squared Frobenius norm of the identity matrix:

- **Regularization Term:** We calculate the squared Frobenius norm of the identity matrix:

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

The Regularization Loss is therefore  $2\lambda$ .

#### 3. Encoder $\beta$ (Projection Matrix)

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

- **Reconstruction Loss:** (Given as 0.001, but let's verify) This matrix keeps the first row of  $\mathbf{X}$  and zeros out the second row. The error is exactly the "missing" second row of  $\mathbf{X}$ : Row 2 of  $\mathbf{X}$  is  $[0.02, -0.01, 0.01, -0.02]$ .

$$\text{Loss} = (0.02)^2 + (-0.01)^2 + (0.01)^2 + (-0.02)^2$$

$$= 0.0004 + 0.0001 + 0.0001 + 0.0004 = 0.001$$

(Matches the table).

- **Regularization Term:**

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

The Regularization Loss is therefore  $1\lambda$  (or just  $\lambda$ ).

Completed Table:

Accurate  
well justified and pedagogically clear.

3. Solve for  $\lambda$  Subtract  $\lambda$  from both sides:

$$\lambda > 0.001$$

**Conclusion:** For any regularization parameter  $\lambda > 0.001$ , the trivial identity solution  $W^{(\alpha)}$  will have a higher total loss than the projection solution  $W^{(\beta)}$ . This demonstrates how regularization forces the model to learn a "simpler" (lower norm) representation rather than just copying the input.



explain the logic behind

Completed Table:

Encoder	Reconstruction Loss	Regularization Loss ( $\text{Term} \times \lambda$ )
$\alpha$	0	$2\lambda$ (from $\ W\ _F^2 =$ )
$\beta$	0.001	$\lambda$ (from $\ W\ _F^2 =$ )

Export to Sheets



#### ii. Finding the Critical Value of $\lambda$

We want to find where the identity matrix  $W^{(\alpha)}$  performs worse (higher loss) than the projection matrix  $W^{(\beta)}$ .

#### 1. Set up the Inequality

We require:

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

Substitute the total loss formula (Reconstruction + Regularization):

$$(\text{Recon}_\alpha + \lambda\|W^{(\alpha)}\|_F^2) > (\text{Recon}_\beta + \lambda\|W^{(\beta)}\|_F^2)$$

#### 2. Substitute the Values

Using the values computed in Part i:

$$0 + \lambda(2) > 0.001 + \lambda(1)$$

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

Based on the problem statement provided in the images, here is the step-by-step derivation and solution for Part (b).

### i. Fill in the Empty Matrices

The optimizing weight matrix  $\widehat{W}$  is given by the singular value decomposition matrices of the data  $\mathbf{X}$ .

The expression is:

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

Filled-in matrices:

- Left Matrix:  $\mathbf{U}$
- Right Matrix:  $\mathbf{U}^\top$

So the full equation is:

$$\widehat{W} = \left[ \begin{array}{c} \mathbf{U} \\ \vdots \end{array} \right] \left[ \begin{array}{ccc} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & \\ & \ddots & \\ & & \frac{\sigma_n^2}{\sigma_n^2 + \lambda} \end{array} \right] \left[ \begin{array}{c} \mathbf{U}^\top \\ \vdots \end{array} \right]$$

### ii. Derivation of the Expression

We want to find the matrix  $W$  that minimizes the regularized loss:

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

**Step 1: Use SVD and Orthogonal Invariance** Let the Singular Value Decomposition (SVD) of the data be  $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ , where:

- $\mathbf{U}$  is an  $m \times m$  orthogonal matrix ( $\mathbf{U}^\top \mathbf{U} = \mathbf{I}$ ).
- $\Sigma$  is an  $m \times n$  diagonal matrix containing singular values  $\sigma_i$ .
- $\mathbf{V}^\top$  is an  $n \times n$  orthogonal matrix.

Recall the property of the Frobenius norm: it is invariant under multiplication by orthogonal matrices. That is,  $\|\mathbf{A}\|_F = \|\mathbf{Q}\mathbf{A}\|_F = \|\mathbf{A}\mathbf{Q}\|_F$  for orthogonal  $\mathbf{Q}$ .

Let's simplify the first term (Reconstruction Loss):

$$\|\mathbf{X} - W\mathbf{X}\|_F^2 = \|\mathbf{U}\Sigma\mathbf{V}^\top - W\mathbf{U}\Sigma\mathbf{V}^\top\|_F^2$$

Multiply from the right by  $\mathbf{V}$  (which removes  $\mathbf{V}^\top$ ):

$$= \|\mathbf{U}\Sigma - W\mathbf{U}\Sigma\|_F^2$$

Multiply from the left by  $\mathbf{U}^\top$ :

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

**Step 2: Rewrite the Objective in terms of  $\widehat{W}$**  Substituting  $\widehat{W}$  into our simplified loss equation:

$$\mathcal{L}(\widehat{W}) = \|\Sigma - \widehat{W}\Sigma\|_F^2 + \lambda\|\widehat{W}\|_F^2$$

This effectively decouples the problem. Because  $\Sigma$  is diagonal, the matrix multiplication  $\widehat{W}\Sigma$  scales the columns of  $\widehat{W}$  by the singular values  $\sigma_j$ . Let  $\widehat{W}_{ij}$  be the element at row  $i$  and column  $j$  of  $\widehat{W}$ . The term  $(\Sigma - \widehat{W}\Sigma)$  has elements:

- Diagonal ( $i = j$ ):  $\sigma_i - \widehat{W}_{ii}\sigma_i$
- Off-diagonal ( $i \neq j$ ):  $0 - \widehat{W}_{ij}\sigma_j$

The squared Frobenius norm is the sum of the squares of all elements:

$$\mathcal{L} = \sum_{i=1}^m \left[ (\sigma_i - \widehat{W}_{ii}\sigma_i)^2 + \sum_{j \neq i} (\widehat{W}_{ij}\sigma_j)^2 \right] + \lambda \sum_{i,j} \widehat{W}_{ij}^2$$

**Step 3: Minimize Component-wise (Decoupled Ridge Regression)** We can minimize this sum by minimizing each  $\widehat{W}_{ij}$  independently.

- **For off-diagonal terms ( $i \neq j$ ):** The loss terms involving  $\widehat{W}_{ij}$  are:  $(\widehat{W}_{ij}\sigma_j)^2 + \lambda\widehat{W}_{ij}^2 = \widehat{W}_{ij}^2(\sigma_j^2 + \lambda)$ . Since  $\sigma^2 \geq 0$  and  $\lambda > 0$ , the minimum occurs at  $\widehat{W}_{ij} = 0$ . Thus,  $\widehat{W}$  is diagonal.
- **For diagonal terms ( $i = j$ ):** Let  $w_i = \widehat{W}_{ii}$ . The loss terms are:

$$J(w_i) = (\sigma_i - w_i\sigma_i)^2 + \lambda w_i^2$$

$$J(w_i) = \sigma_i^2(1 - w_i)^2 + \lambda w_i^2$$

Take the derivative with respect to  $w_i$  and set to 0:

$$\frac{dJ}{dw_i} = -2\sigma_i^2(1 - w_i) + 2\lambda w_i = 0$$

$$-\sigma_i^2 + w_i\sigma_i^2 + \lambda w_i = 0$$

$$w_i(\sigma_i^2 + \lambda) = \sigma_i^2$$

$$w_i = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}$$

**Step 4: Reconstruct  $\widehat{W}$**  We found that the optimal  $\widehat{W}$  is a diagonal matrix with entries  $\frac{\sigma_i^2}{\sigma_i^2 + \lambda}$ . Recall our definition  $\widehat{W} = \mathbf{U}^\top W\mathbf{U}$ . To recover  $W$ , we invert this transformation:

The squared Frobenius norm is the sum of the squares of all elements:

$$\mathcal{L} = \sum_{i=1}^m \left[ (\sigma_i - \widehat{W}_{ii}\sigma_i)^2 + \sum_{j \neq i} (\widehat{W}_{ij}\sigma_j)^2 \right] + \lambda \sum_{i,j} \widehat{W}_{ij}^2$$

**Step 3: Minimize Component-wise (Decoupled Ridge Regression)** We can minimize this sum by minimizing each  $\widehat{W}_{ij}$  independently.

- **For off-diagonal terms ( $i \neq j$ ):** The loss terms involving  $\widehat{W}_{ij}$  are:  $(\widehat{W}_{ij}\sigma_j)^2 + \lambda\widehat{W}_{ij}^2 = \widehat{W}_{ij}^2(\sigma_j^2 + \lambda)$ . Since  $\sigma^2 \geq 0$  and  $\lambda > 0$ , the minimum occurs at  $\widehat{W}_{ij} = 0$ . Thus,  $\widehat{W}$  is diagonal.

- **For diagonal terms ( $i = j$ ):** Let  $w_i = \widehat{W}_{ii}$ . The loss terms are:

$$J(w_i) = (\sigma_i - w_i\sigma_i)^2 + \lambda w_i^2$$

$$J(w_i) = \sigma_i^2(1 - w_i)^2 + \lambda w_i^2$$

Take the derivative with respect to  $w_i$  and set to 0:

$$\frac{dJ}{dw_i} = -2\sigma_i^2(1 - w_i) + 2\lambda w_i = 0$$

$$-\sigma_i^2 + w_i\sigma_i^2 + \lambda w_i = 0$$

$$w_i(\sigma_i^2 + \lambda) = \sigma_i^2$$

$$w_i = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}$$

**Step 4: Reconstruct  $\widehat{W}$**  We found that the optimal  $\widehat{W}$  is a diagonal matrix with entries  $\frac{\sigma_i^2}{\sigma_i^2 + \lambda}$ . Recall our definition  $\widehat{W} = \mathbf{U}^\top W\mathbf{U}$ . To recover  $W$ , we invert this transformation:

$$W = \mathbf{U}\widehat{W}\mathbf{U}^\top$$

Substituting our optimal diagonal matrix:

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

This matches the form in equation (5), confirming the filled-in matrices are  $\mathbf{U}$  and  $\mathbf{U}^\top$ .

*outstanding work — precise  
well organized and easy to follow.  
The derivation is spot-on, and adds a  
bit more intuition about why regularization  
shrinks each mode would make it  
perfect.*

Based on the derivation in part (b), the eigenvalues of the optimal purification matrix  $\widehat{W}$  act as **scaling factors** for the singular directions of the data. Specifically, for the  $i$ -th singular value  $\sigma_i$ , the corresponding direction is scaled by:

$$s_i(\lambda) = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}$$

We need to find the range of  $\lambda$  that satisfies the preservation and attenuation constraints.

#### Step 1: Identify Critical Singular Values

Since the scaling factor  $s_i(\lambda)$  increases as  $\sigma_i$  increases, the constraints are bounded by the "worst-case" singular values in each group:

1. **Preservation Group ( $i = 1, 2, 3$ )**: We want  $s_i \geq 0.8$ . This is hardest to satisfy for the **smallest** singular value in this group.
  - Critical Value:  $\sigma_3 = 4$ .
2. **Attenuation Group ( $i = 4, \dots, 8$ )**: We want  $s_i \leq 0.5$ . This is hardest to satisfy for the **largest** singular value in this group.
  - Critical Value:  $\sigma_4 = 1$ .

#### Step 2: Solve for the Preservation Constraint

We require the scaling factor for  $\sigma_3$  to be at least 80%:

$$\frac{\sigma_3^2}{\sigma_3^2 + \lambda} \geq 0.8$$

Substitute  $\sigma_3 = 4$ :

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

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

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

$$3.2 \geq 0.8\lambda$$

$$\lambda \leq \frac{3.2}{0.8}$$

$$\lambda \leq 4$$

*Accurate.*

*But its explanation lacked the intuitive commentary a human might add about why those specific singular values define the limits.*

#### Step 3: Solve for the Attenuation Constraint

We require the scaling factor for  $\sigma_4$  to be at most 50%:

$$\frac{\sigma_4^2}{\sigma_4^2 + \lambda} \leq 0.5$$

Substitute  $\sigma_4 = 1$ :

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

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

$$2 \leq 1 + \lambda$$

$$\lambda \geq 1$$

#### Final Answer

Combining the two inequalities, the valid set of hyperparameter values is:

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

or expressed as an interval:

$$\lambda \in [1, 4]$$

Accuracy  $\approx 90\%$  (only a small mistake)

Overall, the AI exhibits excellent technical competence and instructional potential — precise, brief and insightful. With slightly more human-style intuition and contextual explanation, its work would be indistinguishable from that of a top-performing graduate-level teaching assistant.