

Multi-part question with detailed explanations



SSM Convolution Kernel

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

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

- (a) **Convolution Kernel and the Output Equation.** Given that the sequence length is L (input: (u_0, \dots, u_L) , output: (y_0, \dots, y_L)) and assume $x_0 = 0$, show that the output y_k can be expressed as a convolution of the input sequence $\{u_\ell\}_0^L$ with a kernel $K = \{K_\ell\}_0^L$:

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

where any $u_{\leq 0}$ with a negative index is set to 0 (zero-padding). Also, find K .

(b) **Concrete Examples.**

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

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

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

- (c) **Efficient Computation with Convolutions.** If we already know the kernel K , how much can we parallelize the computation of the output sequence $\{y_k\}$ for an input sequence $\{u_k\} \in \mathbb{R}^d$ of length L ? What is the minimum critical path length of the computation? What about a naive, direct computation of y_k from the unrolled recursion?
- (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.
- (e) **Adding structure to A .** Suppose A is a diagonal matrix. How can we leverage this structure to compute the kernel K more efficiently?
- (f) **Diagonal-plus-low-rank (DPLR) structure** Now if A has the following form:

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

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

please answer all parts a to f. make sure to explain your thoughts and give complete responses.

12/10/2025, 5:57:18 PM

Thought process: Systematically derived SSM convolution kernels and parallelization strategies.

This is a comprehensive problem about State-Space Models (SSMs) and their convolution kernel representation. Let me work through each part carefully.

The SSM is:

- $x_{k+1} = Ax_k + Bu_k$
- $y_k = Cx_k + Du_k$

With $x_0 = 0$.

Let me solve each part:

```
## Part (a): Convolution Kernel and the Output Equation
```

I need to show that $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$ and find K .

Starting from the state equation with $x_0 = 0$:

- $x_1 = Ax_0 + Bu_0 = Bu_0$
- $x_2 = Ax_1 + Bu_1 = ABu_0 + Bu_1$
- $x_3 = Ax_2 + Bu_2 = A^2Bu_0 + ABu_1 + Bu_2$
- In general: $x_k = \sum_{i=0}^{k-1} A^{k-1-i}Bu_i = \sum_{j=0}^{k-1} A^j Bu_{k-1-j}$

So:

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

Let me reindex. If $\ell = k-1-j$, then $j = k-1-\ell$, and when $j=0$, $\ell=k-1$; when $j=k-1$, $\ell=0$.

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

I'll carefully rewrite the convolution representation, focusing on the kernel definition and indexing transformation.

I'll rearrange the terms to match the convolution form, defining the kernel coefficients systematically. The goal is to express y_k as a sum of input-kernel products that clearly shows how past inputs contribute to the current output.

By substituting $\ell = k - i$ and adjusting the summation limits, I can rewrite the output equation to reveal the convolution structure. This transformation allows me to express the kernel coefficients K_ℓ explicitly, with K_0 representing the direct input coefficient and subsequent K_ℓ capturing the system's dynamic response.

The kernel coefficients emerge naturally: K_0 is the direct input gain, while K_{ℓ} for $\ell \geq 1$ represents the weighted contribution of past inputs through the system's impulse response.

With zero-padding assumed for inputs at negative indices, the output can be compactly represented as a convolution sum, highlighting the system's linear time-invariant characteristics.

The kernel sequence follows a recursive pattern: K_0 is the direct term, while subsequent kernels scale exponentially with the state matrix's power. For the scalar scenario with $\alpha = 0.8$, $\beta = 1$, $\gamma = 1.5$, the kernel decays geometrically, reflecting the system's dampening behavior.

Calculating kernels for $L = 4$ reveals the exponential decay, where each successive kernel diminishes by a factor of 0.8 , demonstrating the system's transient response dynamics.

In the 2D case, the matrix computations become more intricate. The kernel calculation involves matrix multiplication, with K_1 directly computed as the output matrix's first element, and subsequent kernels requiring progressive matrix transformations.

For K_2 , I multiply the control matrix C with the system matrix AB , yielding 0.7 . The process involves carefully tracking matrix dimensions and performing precise matrix operations.

Calculating K_3 requires squaring the system matrix A , which involves complex matrix multiplication. The squared matrix captures the system's quadratic interaction, with each element computed through careful cross-multiplication of rows and columns.

The impulse response reveals how the system dynamically reacts to a single input spike, tracing the kernel's evolution over sequential time steps. This provides insight into the system's fundamental behavior under minimal excitation.

The convolution computation suggests an efficient method for predicting system outputs by leveraging the precomputed kernel sequence, enabling streamlined signal processing calculations.

Parallel computation strategies using Fast Fourier Transform (FFT) dramatically reduce computational complexity, achieving near-logarithmic critical path length. This contrasts sharply with naive sequential approaches that require linear time progression.

Kernel computation fundamentally involves iterative matrix transformations, where each successive kernel represents a progressive system state derived from cascading linear operations.

For diagonal matrices, power computation becomes trivially efficient. By element-wise exponential operations, matrix powers transform directly through coordinate-wise transformations, eliminating complex multiplication overhead.

Parallel computation techniques like prefix scanning and divide-and-conquer strategies enable logarithmic-time power calculations, fundamentally restructuring computational approaches from sequential to massively parallel paradigms.

The key insight emerges in scalar decomposition: representing matrix operations as independent coordinate transformations dramatically simplifies computational complexity, enabling independent parallel processing of each matrix element's power sequence.

By expressing K_{ℓ} as a sum of geometric sequences, we transform a potentially expensive matrix computation into efficient scalar operations. Each term's $\lambda_i^{\ell-1}$ can be rapidly calculated through recurrence or fast Fourier techniques, reducing algorithmic overhead substantially.

The diagonal-plus-low-rank structure introduces intriguing matrix manipulation possibilities. A rank-1 perturbation of the identity matrix creates unique computational opportunities, particularly when considering matrix power expansions.

The Sherman-Morrison formula reveals elegant matrix inversion properties, though our focus shifts to direct power computation. The rank-1 nature of p^T enables recursive power calculations through strategic binomial decomposition.

By introducing $r = \|p\|^2$ and deriving α_ℓ , we establish a compact representation of matrix powers that scales efficiently with computational complexity.

The computation reduces to a simple scalar multiplication involving C_p and $p^T B$, enabling streamlined matrix transformations. This approach allows expressing K_{ℓ} as a linear combination of base terms with a scalar scaling factor derived from the vector's norm.

The kernel calculation leverages this elegant decomposition, transforming matrix operations into lightweight scalar arithmetic that minimizes computational overhead.

SSM Convolution Kernel - Complete Solution

I'll work through each part systematically, explaining the derivations and insights.

Part (a): Convolution Kernel and the Output Equation

Goal: Show that $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$ and find K .

Derivation:

Starting with $x_0 = 0$ and iterating the state equation:

$$x_1 = Ax_0 + Bu_0 = Bu_0$$

$$x_2 = Ax_1 + Bu_1 = ABu_0 + Bu_1$$

$$x_3 = Ax_2 + Bu_2 = A^2Bu_0 + ABu_1 + Bu_2$$

By induction, the general form is: $\sum_{i=0}^{k-1} A^{k-1-i} Bu_i$

Substituting into the output equation: $y_k = Cx_k + Du_k = C\sum_{i=0}^{k-1} A^{k-1-i} Bu_i + Du_k$

Let $\ell = k - i$, so when i ranges from 0 to $k - 1$, ℓ ranges from k down to 1: $y_k = \sum_{\ell=1}^k CA^{\ell-1}Bu_{k-\ell} + Du_k = \sum_{\ell=0}^{k-1} K_\ell u_{k-\ell}$

The Convolution Kernel is:

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

With zero-padding (any $u_{<0} = 0$), we can extend the sum to L : $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$

Part (b): Concrete Examples

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

With $A = \alpha = 0.8$, $B = \beta = 1$, $C = \gamma = 1.5$, and assuming $D = 0$:

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

Computing kernel up to $L = 4$:

ℓ	K_ℓ	Calculation
0	0	$D = 0$
1	1.5	$1.5 \cdot 0.8^0 = 1.5$
2	1.2	$1.5 \cdot 0.8^1 = 1.2$
3	0.96	$1.5 \cdot 0.8^2 = 0.96$
4	0.768	$1.5 \cdot 0.8^3 = 0.768$

The kernel decays geometrically since $|\alpha| < 1$, showing stable system behavior.

(ii) 2D Case

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

Computing K_0 : $K_0 = D = 0$

Computing K_1 : $K_1 = CB = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \boxed{1}$

Computing K_2 :

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

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

Computing K_3 :

$$A^2 = \begin{pmatrix} 0.7 & 0.1 \\ 0.2 & 0.6 \end{pmatrix}^2 = \begin{pmatrix} 0.51 & 0.13 \\ 0.26 & 0.38 \end{pmatrix}$$

$$A^2B = \begin{pmatrix} 0.51 \\ 0.26 \end{pmatrix}$$

$$K_3 = CA^2B = \boxed{0.51}$$

Impulse Response Interpretation: The kernel $K = \{K_0, K_1, K_2, K_3, \dots\}$ is the impulse response of the system. If we apply an impulse input $u_0 = 1, u_k = 0$ for $k \geq 1$, the output sequence is exactly:
 $y_k = K_k$

This captures how the system "remembers" and responds to past inputs over time. The decaying values ($1 \rightarrow 0.7 \rightarrow 0.51$) show that the influence of past inputs diminishes, characteristic of a stable system with eigenvalues inside the unit circle.

Part (c): Efficient Computation with Convolutions

With Known Kernel K (Convolution Approach):

The computation $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$ is a **discrete convolution**.

Using FFT:

1. Compute $\hat{K} = \text{FFT}(K)$ in $O(L \log L)$
2. Compute $\hat{u} = \text{FFT}(u)$ in $O(L \log L)$
3. Element-wise multiply: $\hat{y} = \hat{K} \odot \hat{u}$ in $O(L)$
4. Compute $y = \text{IFFT}(\hat{y})$ in $O(L \log L)$

Parallelization: All L outputs can be computed simultaneously!

- **Critical path length:** $O(\log L)$ (depth of FFT butterfly network)
- **Total work:** $O(L \log L)$

Naive Direct Computation (Unrolled Recursion):

Computing via the original recurrence: $x_{k+1} = Ax_k + Bu_k$, $y_k = Cx_k + Du_k$

This is inherently **sequential** since x_{k+1} depends on x_k .

- **Critical path length:** $O(L)$ — must compute states one by one
- **Total work:** $O(Ln^2)$ for matrix-vector multiplications

Comparison:

Method	Critical Path	Total Work	Parallelizable?
FFT Convolution	$O(\log L)$	$O(L \log L)$	Highly
Direct Recursion	$O(L)$	$O(Ln^2)$	Poorly

Part (d): Efficient Kernel Computation

Challenge: Computing $K_\ell = CA^{\ell-1}B$ for $\ell = 1, \dots, L$ requires powers of A .

Strategy 1: Parallel Prefix (Scan) for Matrix Powers

Since matrix multiplication is associative, we can use a **parallel scan** algorithm:

For $L = 2^N$:

1. **Level 0:** Start with $A^1 = A$
2. **Level 1:** Compute $A^2 = A \cdot A$
3. **Level 2:** Compute $A^4 = A^2 \cdot A^2$
4. Continue until $A^{2^{N-1}}$

Then use a **down-sweep** to fill in all intermediate powers.

- **Parallel depth:** $O(\log L) = O(N)$
- **Total work:** $O(L \cdot n^\omega)$ where ω is matrix multiplication exponent

Strategy 2: Divide and Conquer

Compute $\{A^0, A^1, \dots, A^{L/2-1}\}$ and $\{A^{L/2}, \dots, A^{L-1}\}$ recursively.

- The second half: $A^{L/2+j} = A^{L/2} \cdot A^j$
- Once $A^{L/2}$ is known (via repeated squaring), all second-half powers can be computed in parallel.

Strategy 3: Eigendecomposition (if applicable)

If $A = V\Lambda V^{-1}$ where Λ is diagonal: $\$A^\ell = V\Lambda^\ell V^{-1}$ Each diagonal entry λ_i^ℓ can be computed independently.

Part (e): Adding Structure to A (Diagonal Case)

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

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

Let $C = (c_1, \dots, c_n)$ and $B = (b_1, \dots, b_n)^T$. Then:

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

Key insight: The kernel is a **sum of geometric sequences!**

$$K_\ell = \sum_{i=1}^n \gamma_i \lambda_i^{\ell-1} \quad \text{where } \gamma_i = c_i b_i$$

Efficient Computation:

For each eigenvalue λ_i :

- The sequence $\{\lambda_i^0, \lambda_i^1, \dots, \lambda_i^{L-1}\}$ can be computed:

- Sequentially: $O(L)$ scalar multiplications
- Or via FFT techniques if computing in frequency domain

Total complexity:

- Setup: $O(n)$ to compute $\gamma_i = c_i b_i$
- Kernel computation: $O(nL)$ scalar operations (vs. $O(Ln^3)$ for general matrices)

Parallelization:

- All n geometric sequences can be computed in parallel
- Each sequence can use parallel prefix multiplication: $O(\log L)$ depth

This is a **massive speedup** from matrix operations to scalar operations!

Part (f): Diagonal-Plus-Low-Rank (DPLR) Structure

Given: $A = I_n + pp^T$ where $p \in \mathbb{R}^n$

Key Observation:

pp^T is a **rank-1 matrix**, and its powers have special structure: $(pp^T)^k = (p^T p)^{k-1} (pp^T) = \|p\|^{2(k-1)} pp^T$ for $k \geq 1$

Computing A^ℓ :

Using the binomial theorem: $A^\ell = (I + pp^T)^\ell = \sum_{k=0}^{\ell} \binom{\ell}{k} (pp^T)^k$

$$\begin{aligned} &= I + \sum_{k=1}^{\ell} \binom{\ell}{k} \|p\|^{2(k-1)} pp^T \\ &= I + \frac{pp^T}{\|p\|^2} \sum_{k=1}^{\ell} \binom{\ell}{k} \|p\|^{2k} \\ &= I + \frac{pp^T}{\|p\|^2} [(1 + \|p\|^2)^\ell - 1] \end{aligned}$$

Let $r = \|p\|^2$. Then: $A^\ell = I + \alpha_\ell \cdot pp^T$ where $\alpha_\ell = \frac{(1+r)^\ell - 1}{r}$

Efficient Kernel Computation:

$$K_\ell = CA^{\ell-1}B = C(I + \alpha_{\ell-1}pp^T)B = CB + \alpha_{\ell-1}(Cp)(p^T B)$$

Precompute once ($O(n)$ time):

- $\beta_0 = CB$ (scalar)
- $\beta_1 = Cp$ (scalar)
- $\beta_2 = p^T B$ (scalar)
- $r = \|p\|^2$ (scalar)

Then each kernel element:

$$K_\ell = \beta_0 + \beta_1 \beta_2 \cdot \frac{(1+r)^{\ell-1} - 1}{r}$$

Complexity Analysis:

Operation	Naive (dense A)	DPLR Structure
Setup	—	$O(n)$
Each K_ℓ	$O(n^2)$ matrix mult	$O(1)$ scalar ops
All L kernels	$O(Ln^3)$	$O(n + L)$

This is the key insight behind efficient SSM implementations like S4 (Structured State Spaces):

By constraining A to have DPLR structure, kernel computation becomes dramatically more efficient, enabling practical training and inference for very long sequences.

12/10/2025, 5:58:58 PM



Self-Supervised Linear Purification

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

$$\mathcal{L}_1(W; \mathbf{X}) = \|\mathbf{X} - W\mathbf{X}\|_F^2 \quad (1)$$

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

$$\mathcal{L}_2(W; \mathbf{X}, \lambda) = \underbrace{\|\mathbf{X} - W\mathbf{X}\|_F^2}_{\text{Reconstruction Loss}} + \lambda \underbrace{\|W\|_F^2}_{\text{Regularization Loss}} \quad (2)$$

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

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

$$W^{(\alpha)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad W^{(\beta)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad (3)$$

The training data matrix \mathbf{X} is also given to you as follows:

$$\mathbf{X} = \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix} \quad (4)$$

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

Encoder	Reconstruction Loss	Regularization Loss
α	—	—
β	0.001	—

- ii. For what values of the regularization parameter λ is the identity matrix $W^{(\alpha)}$ get higher loss \mathcal{L}_2 in (2), as compared to $W^{(\beta)}$?

- (b) Now consider a generic square linear encoder $W \in \mathbb{R}^{m \times m}$ and the regularized objective \mathcal{L}_2 reproduced below for your convenience:

$$\mathcal{L}_2(W; \mathbf{X}, \lambda) = \underbrace{\|\mathbf{X} - W\mathbf{X}\|_F^2}_{\text{Reconstruction Loss}} + \lambda \underbrace{\|W\|_F^2}_{\text{Regularization Loss}}$$

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

- i. You are given that the optimizing weight matrix for the regularized objective \mathcal{L}_2 above takes the following form. **Fill in the empty matrices below.**

$$\widehat{W} = \left[\begin{array}{c} \\ \end{array} \right] \cdot \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & & \\ & \frac{\sigma_2^2}{\sigma_2^2 + \lambda} & & \\ & & \ddots & \\ & & & \frac{\sigma_n^2}{\sigma_n^2 + \lambda} \end{bmatrix} \cdot \left[\begin{array}{c} \\ \end{array} \right] \quad (5)$$

- ii. Derive the above expression.

(Hint: Can you understand $\mathcal{L}_2(W; \mathbf{X}, \lambda)$ as a sum of m completely decoupled ridge-regression problems?)

(Hint: The Frobenius norm is equal to $\|A\|_F^2 := \text{tr}(AA^T)$, and it is invariant under orthogonal transform. That is, $\|A\|_F^2 = \|UAV^T\|_F^2$ for any orthogonal matrices U, V , and any rectangular matrix A , as long as U, A, V have compatible shapes.)

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

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

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

(Hint: What are the two critical singular values to focus on?)

please answer all parts a to c. make sure to explain your thoughts and give complete responses.

12/10/2025, 6:04:22 PM

Thought process: Systematically derived optimal weight matrices and computed regularization bounds.

Let me analyze this problem on Self-Supervised Linear Purification. I'll work through each part carefully.

The setup:

- Linear encoder with square weight matrix $W \in \mathbb{R}^{m \times m}$
- Training data $\mathbf{X} \in \mathbb{R}^{m \times n}$
- Loss $\mathcal{L}_1(W; \mathbf{X}) = \|\mathbf{X} - W\mathbf{X}\|_F^2$
- Regularized loss $\mathcal{L}_2(W; \mathbf{X}, \lambda) = \|\mathbf{X} - W\mathbf{X}\|_F^2 + \lambda \|W\|_F^2$

Let me work through each part.

Part (a)

For $m = 2$:

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

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

Part (a)(i): Compute reconstruction and regularization losses

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

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

- Regularization Loss: The identity matrix's Frobenius norm squared is simply the

sum of squared diagonal elements, which equals 2

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

I'll continue the calculation by completing the matrix multiplication of $W^{(\beta)} \mathbf{X}$, which involves the first matrix row multiplied by the input matrix columns.

The resulting matrix shows how $W^{(\beta)}$ transforms the input, with the first row capturing some input values and the second row zeroed out.

- Reconstruction Loss: Calculating the squared Frobenius norm of the difference between the original input and the reconstructed input, which yields a small value of 0.001
- Regularization Loss: For $W^{(\beta)}$, the Frobenius norm squared is 1, representing the squared magnitude of the matrix elements

I'll continue analyzing the loss comparison by examining the λ values that would cause $W^{(\alpha)}$ to have higher total loss than $W^{(\beta)}$

. By solving the inequality, I find that λ must be greater than 0.001 to achieve this condition.

The problem now shifts to considering a generic weight matrix W with a singular value decomposition of \mathbf{X} , preparing to explore the optimal weight matrix structure under specific constraints.

I'll examine how the weight matrix transforms when decomposed using singular value decomposition, focusing on how each singular value contributes to the optimal solution. The key insight involves manipulating the matrix through orthogonal transformations while preserving its fundamental properties.

By introducing a transformed coordinate system through U^T , I can simplify the problem's complexity and derive a more elegant representation of the optimal weight matrix. This approach allows me to understand how regularization interacts with the matrix's underlying structural components.

The diagonal structure becomes critical, revealing how each singular value contributes to the reconstruction loss. By carefully tracking these transformations, I can develop a precise characterization of the optimal weight matrix's behavior across different singular value scales.

The loss function simplifies to a weighted sum of squared singular values, scaled by $(1 - w_i)^2$. This suggests the reconstruction error depends directly on how much each singular dimension is compressed or modified.

I need to verify whether the optimal solution truly maintains a diagonal structure, which will require examining the loss function's gradient and constraints more carefully.

Breaking down the Frobenius norm reveals a detailed interaction between the weight matrix \tilde{W} and the singular value matrix Σ . The key observation is how each matrix entry contributes to the overall reconstruction error, particularly focusing on the diagonal elements where σ_j appears.

The computation shows that when Σ is diagonal, the matrix multiplication $\tilde{W}\Sigma$ simplifies to scaling singular values by corresponding weight matrix entries. This provides insight into how individual singular dimensions are transformed.

The regularization term $\|\tilde{W}\|_F^2$ captures the overall complexity of the weight matrix, penalizing large off-diagonal entries. By introducing λ , the optimization balances reconstruction accuracy with model complexity.

For off-diagonal terms, the loss function suggests a nuanced trade-off where each weight's contribution is scaled by both the singular value and a regularization constant, effectively controlling the model's sensitivity to different matrix dimensions.

Solving for diagonal entries reveals an elegant shrinkage mechanism. The optimal diagonal weights are proportional to the squared singular values, scaled by a regularization factor that prevents overfitting. This approach ensures that less significant components are effectively damped while preserving the most informative matrix features.

The resulting diagonal matrix will adaptively adjust its entries based on the underlying data's structural characteristics, providing a principled approach to matrix estimation.

For the specific problem, I'll need to find a λ that satisfies the dual constraints on the first three and last five singular values. This requires carefully balancing the regularization parameter to achieve the desired shrinkage behavior across different matrix components.

Using $\sigma_3 = 4$ as the critical point for the first condition, I derive $\lambda \leq 4$ by algebraically manipulating the shrinkage fraction. For the second condition with $\sigma_4 = 1$, I notice an impossible mathematical scenario where the denominator constraint leads to a nonsensical inequality.

The mathematical derivation suggests the need to reconsider the second condition's formulation or adjust the approach to finding an appropriate λ value that maintains the desired singular value transformations.

By systematically verifying the boundary conditions $\lambda = 1$ and $\lambda = 4$, I confirm the singular value constraints are satisfied. The critical checks validate that both σ_3 and σ_4 meet their respective threshold requirements within this interval.

The final result establishes $\lambda \in [1, 4]$, representing the range where the singular value transformations maintain the specified mathematical constraints.

Self-Supervised Linear Purification - Complete Solution

Part (a): Simplified Case Analysis

(a)(i) Computing Losses for Both Encoders

Given:

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

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

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

Reconstruction Loss:

$$W^{(\alpha)}\mathbf{X} = I \cdot \mathbf{X} = \mathbf{X}$$

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

Regularization Loss:

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

For $W^{(\beta)}$ (Projection onto first coordinate):

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

Reconstruction Loss:

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

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

Regularization Loss:

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

Completed Table:

Encoder	Reconstruction Loss	Regularization Loss
α	0	2
β	0.001	1

(a)(ii) When Does $W^{(\alpha)}$ Have Higher Loss?

Computing the total regularized loss for each:

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

$$\mathcal{L}_2(W^{(\beta)}; \mathbf{X}, \lambda) = 0.001 + \lambda \cdot 1 = 0.001 + \lambda$$

For $W^{(\alpha)}$ to have **higher** loss than $W^{(\beta)}$: $2\lambda > 0.001 + \lambda$ $\Rightarrow \lambda > 0.001$

$$\boxed{\lambda > 0.001}$$

Interpretation: When regularization is strong enough ($\lambda > 0.001$), the identity matrix is penalized more heavily due to its larger Frobenius norm, even though it perfectly reconstructs the data. The "purifier" $W^{(\beta)}$ discards the low-variance second component, achieving a better trade-off.

Part (b): General Solution with SVD**(b)(i) Fill in the Matrices**

Given $\mathbf{X} = U\Sigma V^T$, the optimal weight matrix is:

$$\widehat{W} = U \cdot \text{diag} \left(\frac{\sigma_1^2}{\sigma_1^2 + \lambda}, \frac{\sigma_2^2}{\sigma_2^2 + \lambda}, \dots, \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \right) \cdot U^T$$

The three matrices are:

- **Left:** U (left singular vectors of \mathbf{X})
 - **Middle:** Diagonal matrix with entries $\frac{\sigma_i^2}{\sigma_i^2 + \lambda}$
 - **Right:** U^T
-

(b)(ii) Derivation

Step 1: Transform to SVD Coordinates

Let $\tilde{W} = U^T W U$, so $W = U \tilde{W} U^T$.

Since the Frobenius norm is invariant under orthogonal transformations: $\|W\|_F^2 = \|U^T W U\|_F^2 = \|\tilde{W}\|_F^2$

Step 2: Simplify the Reconstruction Loss

$$\begin{aligned} \|\mathbf{X} - W\mathbf{X}\|_F^2 &= \|U\Sigma V^T - U\tilde{W}U^T U\Sigma V^T\|_F^2 \\ &= \|U(\Sigma - \tilde{W}\Sigma)V^T\|_F^2 = \|\Sigma - \tilde{W}\Sigma\|_F^2 \end{aligned}$$

Step 3: Decoupled Optimization

Let \tilde{w}_{ij} be the entries of \tilde{W} . The reconstruction term is: $\|\Sigma - \tilde{W}\Sigma\|_F^2 = \sum_{j=1}^m \sigma_j^2 (1 - \tilde{w}_{jj})^2 + \sum_{j \neq k} \tilde{w}_{jk}^2 (\sigma_k^2 + \lambda)$

The total loss becomes: $\mathcal{L} = \sum_{j=1}^m \sigma_j^2 (1 - \tilde{w}_{jj})^2 + \sum_{j \neq k} \tilde{w}_{jk}^2 (\sigma_k^2 + \lambda)$

Off-diagonal terms: For $j \neq k$, the term $\tilde{w}_{jk}^2 (\sigma_k^2 + \lambda)$ is minimized when $\tilde{w}_{jk} = 0$.

Diagonal terms: For each j , we minimize: $f(\tilde{w}_{jj}) = \sigma_j^2 (1 - \tilde{w}_{jj})^2 + \lambda \tilde{w}_{jj}^2$

Taking the derivative and setting to zero: $\frac{df}{d\tilde{w}_{jj}} = -2\sigma_j^2 (1 - \tilde{w}_{jj}) + 2\lambda \tilde{w}_{jj} = 0$ $\Rightarrow \sigma_j^2 (1 - \tilde{w}_{jj}) = \lambda \tilde{w}_{jj}$

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

This is exactly **ridge regression shrinkage**! Each singular direction is treated as an independent 1D regression problem, and the solution shrinks coefficients toward zero proportionally to λ .

Part (c): Designing λ for Selective Preservation

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

Requirements:

- **Preserve $\geq 80\%$** for first 3 directions: $\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8$ for $i = 1, 2, 3$
- **Attenuate to $\leq 50\%$** for remaining directions: $\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \leq 0.5$ for $i = 4, \dots, 8$

Critical singular values:

- $\sigma_3 = 4$ (smallest to preserve)
- $\sigma_4 = 1$ (largest to attenuate)

Upper bound from preservation constraint (using $\sigma_3 = 4$):

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

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

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

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

$$\lambda \leq 4$$

Lower bound from attenuation constraint (using $\sigma_4 = 1$):

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

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

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

$$1 \leq 0.5 + 0.5\lambda$$

$$\lambda \geq 1$$



Verification at boundaries:

λ	$\sigma_3 = 4$ preservation	$\sigma_4 = 1$ attenuation
1	$\frac{16}{17} \approx 0.94 \checkmark$	$\frac{1}{2} = 0.50 \checkmark$
4	$\frac{16}{20} = 0.80 \checkmark$	$\frac{1}{5} = 0.20 \checkmark$

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

Interpretation: The regularization parameter must be large enough to suppress the noise directions (small singular values) but small enough to preserve the signal directions (large singular values). This range leverages the **spectral gap** between $\sigma_3 = 4$ and $\sigma_4 = 1$ to achieve selective filtering.

12/10/2025, 6:05:37 PM



- $O(n^2d^3)$
- $O(n^2d^2)$
- $O(n^2)$
- $O(1)$

(HINT: Do not forget that for a single d -dimensional query vector \mathbf{q} , attention needs to return a d -dimensional result.)

- (d) Assume that a ridge self-attention layer is used in a Transformer architecture and there is a downstream loss. **For which of these will backprop successfully pass gradients if we use ridge self-attention?**
- The ridge λ viewed as a learnable parameter for the self-attention layer.
 - The keys
 - The values
 - The queries
- (e) Now step back. There is a nice trick (called the Sherman–Morrison Formula) by which one can update the inverse of an invertible matrix to which you make a rank-1 update. Let M be an invertible square $d \times d$ matrix and let \mathbf{u}, \mathbf{v} be two d -dimensional vectors. Then:

$$(M + \mathbf{u}\mathbf{v}^\top)^{-1} = M^{-1} - \frac{1}{1 + \mathbf{v}^\top M^{-1} \mathbf{u}} (M^{-1}\mathbf{u})(\mathbf{v}^\top M^{-1}) \quad (7)$$

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

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

- (f) Consider implementing causal ridge-self-attention with a context length of n but where the pool of (key, value) vectors that one is querying at position t consists only of the t (key, value) pairs so far.

Describe explicitly how you would compute causal ridge-self-attention in a computationally efficient manner. Leverage your decomposition of the ridge-regression formula in part (b) of this problem together with the Sherman-Morrison formula from (7) to avoid having to do $O(n^2)$ computations while still calculating causal ridge-self-attention outputs correctly for all n positions in the context.

(HINT: Think recursively. What do you need to track from one time step to the next to avoid repeating work?)

- (g) Many people consider important the ability to visualize the attention weights. For traditional softmax-attention, these are the outputs of the softmax for a given query vector. They tell you the exact amounts by which the attention outputs at this position are linear combinations of the values being fed in at this and other positions.

For ridge-attention and a given query vector, how would you compute the (possibly negative) weights associated to each of the value vectors in the context?

Ridge-Attention

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

- (a) To start, let's think about why it is possible to efficiently update simple averaging. Let $m = \frac{1}{n} \sum_{i=1}^n x_i$ be the average of n points. Use m, x_{n+1}, n and simple arithmetic operations to compute $m' = \frac{1}{n+1} \sum_{i=1}^{n+1} x_i$ — the average of all points including the new point x_{n+1} .
(HINT: Start by multiplying m by n .)

- (b) Let us now shift to thinking about traditional ridge-regression with n training pairs (\mathbf{x}_i, y_i) where \mathbf{x}_i

are d -dimensional vectors and y_i are scalars. Let the matrix $A = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix}$ and vector $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ so that

we can find the familiar closed-form solution

$$\mathbf{w}_* = (A^\top A + \lambda I)^{-1} A^\top \mathbf{y} \quad (6)$$

that allows us to make scalar predictions on any new test input \mathbf{x} by computing $\mathbf{w}_*^\top \mathbf{x}$.

First, write the two terms $(A^\top A + \lambda I)$ and $A^\top \mathbf{y}$ as sums involving the \mathbf{x}_i and y_i .

i.e. Complete:

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

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

- (c) Suppose we wanted to do ridge-self-attention (non-causal – “encoder-style”) with a context length of n and d -dimensional query, key, and value vectors. Recall that this is the style of attention where each of the n queries is applied to the same pool of n (key, value) pairs. The goal is to calculate multi-dimensional ridge-regression predictions, after “training” on the pool of (key, value) pairs, and given the query as a kind of “test” input. (Note: the keys are playing the role of the A matrix in ridge-regression, each query is like the \mathbf{x} we are testing on, and in place of the scalar y_i , we have an entire value vector for a multi-dimensional ridge-regression problem so the \mathbf{y} vector is replaced with a matrix that has a row for each value vector.)

Assume that the cost of inverting a $d \times d$ matrix is $O(d^3)$ and the cost of multiplying two such matrices is also $O(d^3)$. Assume that a $d \times d$ matrix times either a d -dimensional row or column vector costs d^2 operations. You should assume $d < n$.

What is the computational cost of a non-causal ridge self-attention layer?

- $O(d^4)$
- $O(nd^2)$

please answer all parts a to g. make sure to explain your thoughts and give complete responses.

12/10/2025, 6:05:52 PM

Thought process: Systematically deconstructed ridge regression mechanics and attention mechanisms.

This is a comprehensive problem about Ridge-Attention, which connects ridge

regression with attention mechanisms in transformers. Let me work through each part carefully.

Part (a): Efficient Update of Simple Averaging

We have $m = \frac{1}{n} \sum_{i=1}^n x_i$ and need to compute $m' = \frac{1}{n+1} \sum_{i=1}^{n+1} x_i$.

Starting with $mn = \sum_{i=1}^n x_i$

So $\sum_{i=1}^{n+1} x_i = mn + x_{n+1}$

Therefore:

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

Part (b): Ridge Regression Decomposition

The ridge regression solution is $\mathbf{w}^* = (\mathbf{A}^T \mathbf{A} + \lambda \mathbf{I})^{-1} \mathbf{A}^T \mathbf{y}$

where \mathbf{A} has rows \mathbf{x}_i^T .

$\mathbf{A}^T \mathbf{A} = \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^T$ (sum of outer products)

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

Part (c): Computational Cost of Non-Causal Ridge Self-Attention

In ridge self-attention:

- Keys play role of \mathbf{x}_i (the \mathbf{A} matrix rows)
- Values play role of \mathbf{y}_i (but now it's a matrix \mathbf{V} instead of vector \mathbf{y})
- Query is the test input

The formula becomes: $\text{output} = (\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{K}^T \mathbf{V}$ applied to query

I'll analyze the computational complexity for a single query's processing.

Calculating $\mathbf{K}^T \mathbf{K}$ involves computing outer products of key vectors, which requires $O(nd^2)$ operations. Adding the regularization term $\lambda \mathbf{I}$ takes $O(d^2)$ time. Matrix inversion of the $d \times d$ matrix demands $O(d^3)$ computational effort. The final step of computing $\mathbf{K}^T \mathbf{V}$ involves summing matrix-vector products across all key-value pairs.

For multiple queries, the first five steps remain consistent, with the query application step scaling to $O(nd^2)$. The overall computational complexity

simplifies to $\$O(nd^2)$, given that the dimension d is smaller than the number of samples n .

Regarding differentiability, the key components - regularization parameter λ , key vectors, and value vectors - all participate in differentiable operations within the matrix computations.

The Sherman-Morrison formula provides an efficient matrix inversion technique. By precomputing M^{-1} , we can perform matrix-vector and vector-matrix multiplications efficiently, with computational steps ranging from $\$O(d)$ to $\$O(d^2)$.

For causal attention, we strategically limit key and value computations to preceding positions. At each step, the output depends on a recursive computation involving keys, values, and a regularization parameter λ . The key innovation involves incrementally updating matrix computations as new positions are processed.

The computational approach leverages a recursive structure where K_t builds upon K_{t-1} by incorporating the latest key vector, enabling efficient incremental updates to the attention mechanism.

By maintaining running computations of key-key and key-value matrices, we transform a traditionally quadratic problem into a linear-time algorithm, reducing computational complexity from $\$O(n^2)$ to $\$O(nd^2)$.

Ridge-attention modifies standard softmax attention by introducing a regularization term λ . Instead of direct dot product weights, we compute an inverse matrix W that incorporates both key interactions and regularization, enabling more stable and controlled attention mechanisms.

The output computation shifts from simple weighted averaging to a more sophisticated matrix multiplication, where W acts as a learned, regularized projection matrix transforming key-value interactions through the query vector.

For multi-dimensional outputs, this approach allows each value to be influenced by the entire key matrix, creating a more nuanced representation that balances between direct similarity and structural regularization.

The prediction framework reveals how ridge regression principles can be elegantly applied to attention mechanisms, providing a principled approach to managing high-dimensional feature interactions.

By computing weights through a matrix transformation involving kernel similarity

and regularization, we derive a flexible mechanism for value weighting that captures complex relationships across input dimensions.

The resulting weight vector \mathbf{w} represents a sophisticated mapping that transforms input queries into weighted value representations, enabling adaptive and context-sensitive transformations.

Ridge-Attention - Complete Solution

Part (a): Efficient Update of Simple Averaging

Goal: Express $m' = \frac{1}{n+1} \sum_{i=1}^{n+1} x_i$ using m, x_{n+1}, n .

Derivation:

Starting with $m = \frac{1}{n} \sum_{i=1}^n x_i$, multiply both sides by n : $mn = \sum_{i=1}^n x_i$

Now for the new average: $m' = \frac{1}{n+1} \sum_{i=1}^{n+1} x_i = \frac{1}{n+1} \left(\sum_{i=1}^n x_i + x_{n+1} \right) = \frac{mn + x_{n+1}}{n+1}$

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

This is an $O(1)$ update (for scalars) or $O(d)$ for d -dimensional vectors, rather than recomputing the entire sum.

Part (b): Ridge Regression Decomposition

Goal: Express $(A^T A + \lambda I)$ and $A^T \mathbf{y}$ as sums.

Given that A has rows \mathbf{x}_i^T :

$$A^T A = [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \cdots \quad \mathbf{x}_n] \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix} = \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^T$$

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

For $A^T \mathbf{y}$: $A^T \mathbf{y} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^n y_i \mathbf{x}_i$

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

Part (c): Computational Cost of Non-Causal Ridge Self-Attention

Setup:

- n queries, keys, and values (all d -dimensional)
- Keys play the role of A matrix rows ($\mathbf{x}_i \rightarrow \mathbf{k}_i$)
- Values replace scalar y_i with vectors \mathbf{v}_i
- Query is the test input

The ridge self-attention formula: For each query \mathbf{q} , the output is: $\text{output} = \underbrace{(\mathbf{K}^T \mathbf{K} + \lambda I)^{-1}}_{d \times d} \underbrace{\mathbf{K}^T \mathbf{V}}_{d \times d} \cdot \mathbf{q}$

Cost breakdown:

Step	Operation	Cost
1	Compute $\mathbf{K}^T \mathbf{K} = \sum_{i=1}^n \mathbf{k}_i \mathbf{k}_i^T$	$O(nd^2)$
2	Add λI	$O(d^2)$
3	Invert $d \times d$ matrix	$O(d^3)$
4	Compute $\mathbf{K}^T \mathbf{V} = \sum_{i=1}^n \mathbf{k}_i \mathbf{v}_i^T$	$O(nd^2)$
5	Multiply $(\mathbf{K}^T \mathbf{K} + \lambda I)^{-1} \cdot \mathbf{K}^T \mathbf{V}$	$O(d^3)$
6	Apply to all n queries	$O(nd^2)$

Total: $O(nd^2 + d^3)$

Since $d < n$, this simplifies to:

$$O(nd^2)$$

Part (d): Gradient Backpropagation

Which components successfully pass gradients?

All operations in ridge self-attention are differentiable:

Component	Appears in	Backprop?
λ	$(K^T K + \lambda I)^{-1}$	<input checked="" type="checkbox"/> Yes - matrix inverse is differentiable w.r.t. λ
Keys	$K^T K$ and $K^T V$	<input checked="" type="checkbox"/> Yes - outer products are differentiable
Values	$K^T V$	<input checked="" type="checkbox"/> Yes - linear operation
Queries	Final multiplication	<input checked="" type="checkbox"/> Yes - linear operation

All four: λ , keys, values, and queries

Explanation: Unlike softmax attention which has discrete argmax-like behavior, ridge regression has a smooth, closed-form solution. Matrix inversion is differentiable (with derivative involving the inverse itself), making all parameters learnable.

Part (e): Sherman-Morrison Computational Cost

Given: M^{-1} already computed. Find cost of computing $(M + \mathbf{u}\mathbf{v}^T)^{-1}$.

Formula:

$$(M + \mathbf{u}\mathbf{v}^T)^{-1} = M^{-1} - \frac{1}{1 + \mathbf{v}^T M^{-1} \mathbf{u}} (M^{-1} \mathbf{u})(\mathbf{v}^T M^{-1})$$

Cost analysis:

Step	Operation	Cost
1	$M^{-1} \mathbf{u}$ (matrix-vector)	$O(d^2)$
2	$\mathbf{v}^T M^{-1}$ (vector-matrix)	$O(d^2)$
3	$\mathbf{v}^T M^{-1} \mathbf{u}$ (inner product using step 1)	$O(d)$
4	$(M^{-1} \mathbf{u})(\mathbf{v}^T M^{-1})$ (outer product)	$O(d^2)$
5	Scalar division and matrix subtraction	$O(d^2)$

Total:

$O(d^2)$

This is much cheaper than $O(d^3)$ for full matrix inversion!

Part (f): Efficient Causal Ridge Self-Attention

Challenge: At position t , we need output using only keys/values from positions $1, \dots, t$.

Key Insight: The matrices can be updated incrementally!

Define running sums:

$$S_t = K_t^T K_t + \lambda I = \lambda I + \sum_{i=1}^t \mathbf{k}_i \mathbf{k}_i^T$$

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

Recursive updates:

$$S_t = S_{t-1} + \mathbf{k}_t \mathbf{k}_t^T \quad (\text{rank-1 update!})$$

$$C_t = C_{t-1} + \mathbf{k}_t \mathbf{v}_t^T$$

Algorithm:

Initialize: $S_0 = \lambda I$, $S_0^{-1} = (1/\lambda)I$, $C_0 = \mathbf{0}$

For $t = 1$ to n :

1. Update $S_t = S_{t-1} + \mathbf{k}_t \mathbf{k}_t^T$ $[O(d^2)]$
2. Update S_t^{-1} using Sherman-Morrison on S_{t-1}^{-1} $[O(d^2)]$
3. Update $C_t = C_{t-1} + \mathbf{k}_t \mathbf{v}_t^T$ $[O(d^2)]$
4. Compute $\text{output}_t = S_t^{-1} C_t$ $[O(d^2)]$

Complexity:

- Per position: $O(d^2)$
- Total for n positions: $O(nd^2)$

Total: $O(nd^2)$ instead of $O(n^2d^2)$ for naive causal attention

What to track: At each step, maintain S_t^{-1} (the inverse) and C_t (the key-value correlation matrix).

Part (g): Computing Ridge-Attention Weights

Goal: Find weights w_i such that $\text{output} = \sum_{i=1}^n w_i \mathbf{v}_i$.

Derivation:

The ridge-attention output for query \mathbf{q} is: $\text{output} = (\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{K}^T \mathbf{V} \cdot \mathbf{q}$

Let $\mathbf{a} = (\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{q}$, a d -dimensional vector.

Then: $\text{output} = \mathbf{K}^T \mathbf{V} \cdot \mathbf{a} = \sum_{i=1}^n \mathbf{k}_i \mathbf{v}_i^T \mathbf{a} = \sum_{i=1}^n (\mathbf{k}_i^T \mathbf{v}_i) \mathbf{a}$

Wait, let me reconsider. Looking at dimensions more carefully:

If K is $n \times d$ (rows are \mathbf{k}_i), then $\mathbf{K}^T \mathbf{K}$ is $d \times d$, and $\mathbf{K}^T \mathbf{V}$ is $d \times d$.

The output is: $(\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{K}^T \mathbf{V} \mathbf{q}$

Actually, thinking about it as a weighted sum of values: $\text{output} = \mathbf{V}^T (\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{q}$

where V has rows \mathbf{v}_i^T , so V^T has columns \mathbf{v}_i .

Let $\mathbf{a} = (\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{q}$ and $\mathbf{s} = K \mathbf{a}$ where K has rows \mathbf{k}_i^T .

Then $s_i = \mathbf{k}_i^T \mathbf{a}$ and: $\text{output} = \mathbf{V}^T \mathbf{s} = \sum_{i=1}^n s_i \mathbf{v}_i^T \mathbf{a} = \sum_{i=1}^n (\mathbf{k}_i^T \mathbf{v}_i) \mathbf{a}$

The attention weight for value \mathbf{v}_i is:

$$w_i = \mathbf{k}_i^T (\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{q}$$

Or in vector form: $\boxed{\mathbf{w} = \mathbf{K}(\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{q}}$

Key differences from softmax attention:

1. Weights can be **negative** (unlike softmax which is always positive)
2. Weights **don't sum to 1** (no normalization constraint)
3. Weights depend on the **global key structure** through $(\mathbf{K}^T \mathbf{K} + \lambda \mathbf{I})^{-1}$, not just pairwise query-key similarity

12/10/2025, 6:07:07 PM

 for 1c, your critical path length is correct, but how did you compute the total work? in particular, a matmul will take log(n)

12/10/2025, 6:20:54 PM

Thought process: Reconsidered work complexity accounting for logarithmic matrix multiplication depth.

The user is asking about part (c) from the first SSM Convolution Kernel problem. Let me re-read the question.

Part (c) asks: "If we already know the kernel K , how much can we parallelize the computation of the output sequence $\{y_k\}$ for an input sequence $\{u_k\} \in R^d$ of length L ? What is the minimum critical path length of the computation? What about a naive, direct computation of y_k from the unrolled recursion?"

The user is pointing out that in my analysis, I said the total work for FFT convolution is $O(L \log L)$, but they're noting that a matrix multiplication (or parallel reduction) takes $O(\log n)$ depth but the work is different.

Let me reconsider this more carefully.

For the convolution approach using FFT:

- The convolution $y_k = \sum_{l=0}^{L-1} K_l u_{k-l}$

If we're dealing with scalar inputs/outputs:

- FFT of length L sequence: $O(L \log L)$ work, $O(\log L)$ depth
- Element-wise multiplication: $O(L)$ work, $O(1)$ depth
- IFFT: $O(L \log L)$ work, $O(\log L)$ depth
- Total work: $O(L \log L)$, Critical path: $O(\log L)$

But wait, the problem says $u_k \in R^d$, so the inputs are d-dimensional vectors. And K_l would be matrices (or the output is computed via matrix-vector products).

Actually, looking back at the SSM setup:

- $x_{k+1} = Ax_k + Bu_k$ where $x \in R^n$, $u \in R^d$
- $y_k = Cx_k + Du_k$ where $y \in R^p$

So $K_l = CA^{l-1}B$ would be a $p \times d$ matrix.

The convolution computation involves matrix-vector multiplications for each term. With K_l being $p \times d$ and u_{k-l} being $d \times 1$, each matrix-vector product takes $O(pd)$ work. Summing $L+1$ terms means the total work becomes $O(L * pd)$, which is

significant for large matrices and sequence lengths.

If we compute directly without FFT, the computational complexity increases. For each output y_k , we perform L matrix-vector products and then sum them, resulting in $O(L^2 \cdot pd)$ total work. The FFT approach could potentially reduce this computational burden by working on each kernel and input entry more efficiently.

The naive direct computation through unrolled recursion reveals a critical path length of $O(L)$, with total work scaling to $O(Ln^2)$ for matrix-vector multiplications. This highlights the computational challenges in sequential processing.

When examining the convolution $y_k = \sum_{\ell=0}^{L-1} K_\ell u_{k-\ell}$, we see interesting parallel computation possibilities. For scalar operations, each output requires L multiplications and $L-1$ additions, with all L outputs potentially computed simultaneously. This leads to a total work complexity of $O(L^2)$.

For matrix-valued kernels, the complexity increases. Each matrix-vector product becomes $O(n^2)$, and with L such operations per output, the computational demands grow significantly. The potential for parallel reduction trees offers some optimization, reducing the critical path depth to $O(\log L)$.

The key insights involve distinguishing between total work and critical path length, highlighting how parallel architectures can dramatically reshape computational strategies across different problem scales.

For convolution-based computation, each output can be generated independently, with matrix-vector products parallelizable and vector reduction achievable in logarithmic depth. The critical path depends on matrix multiplication characteristics, potentially ranging from constant to logarithmic complexity.

FFT approaches offer logarithmic depth with work scaling proportionally to problem size. In contrast, sequential recursion methods like state updates demonstrate inherent linear path dependencies, revealing fundamental differences in computational structure.

The core challenge lies in understanding how parallel computation transforms traditional sequential complexity metrics, particularly for matrix operations where depth and work exhibit nuanced relationships. I'll compute the computational complexity for different convolution and recursion approaches, focusing on work and depth trade-offs. The convolution method using a known kernel reveals interesting parallel computation strategies. With L outputs, matrix-vector products can be parallelized, achieving $O(L^2 \cdot pd)$ work and $O(\log L)$ depth.

The FFT approach dramatically reduces computational complexity, achieving $O(pd \cdot L \log L)$ work with $O(\log L)$ depth - significantly more efficient than the direct convolution method.

For naive unrolled recursion, sequential computation dominates, requiring $O(L(n^2 + nd))$ work and $O(L \cdot \log n)$ depth. Each step involves matrix-vector multiplications that limit parallel execution.

The key insight is the trade-off between work and depth across these computational strategies, with FFT offering the most promising complexity profile for large-scale convolution problems.

You raise a great point. Let me redo the analysis more carefully, properly distinguishing **work** (total operations) from **depth** (critical path length).

Part (c) Revised: Parallel Computation Analysis

Setup

- Input sequence $\{u_k\} \in \mathbb{R}^d$ of length L
- Kernel $K_\ell = CA^{\ell-1}B$ where K_ℓ is a $p \times d$ matrix
- Output $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$

Method 1: Direct Parallel Convolution (with known K)

For each output $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$:

Step 1: Compute all L matrix-vector products $K_\ell u_{k-\ell}$ in parallel

- Each matrix-vector product ($p \times d$ matrix times $d \times 1$ vector):
 - Work: $O(pd)$
 - Depth: $O(\log d)$ (parallel inner products)
- All L products computed in parallel

Step 2: Sum the L resulting vectors using parallel reduction tree

- Work: $O(Lp)$ additions
- Depth: $O(\log L)$ levels

Per output y_k :

- Work: $O(Lpd)$
- Depth: $O(\log d + \log L)$

For all L outputs (computed in parallel):

Metric	Value
Total Work	$O(L^2pd)$
Critical Path (Depth)	$O(\log L + \log d) = O(\log L)$

Method 2: FFT-based Convolution

If we treat each component separately and use FFT:

Metric	Value
Total Work	$O(pd \cdot L \log L)$
Critical Path	$O(\log L)$

The FFT approach has **less work** by a factor of $L / \log L$.

Naive Direct Recursion: $x_{k+1} = Ax_k + Bu_k$

This is inherently sequential since x_{k+1} depends on x_k .

Per time step:

- Ax_k : Work $O(n^2)$, Depth $O(\log n)$
- Bu_k : Work $O(nd)$, Depth $O(\log d)$
- Addition: Work $O(n)$, Depth $O(1)$

For all L steps (sequential dependency):

Metric	Value
Total Work	$O(L(n^2 + nd))$
Critical Path (Depth)	$O(L \log n)$

Summary Comparison

Method	Total Work	Critical Path
Direct Parallel Convolution	$O(L^2pd)$	$O(\log L)$
FFT Convolution	$O(pdL \log L)$	$O(\log L)$
Naive Recursion	$O(L(n^2 + nd))$	$O(L \log n)$

Key insight: The convolution approaches achieve **exponentially better parallelism** ($O(\log L)$ vs $O(L \log n)$ depth), though the naive recursion may have less total work when $n \ll L$.

12/10/2025, 6:22:19 PM

 no, this is still not really correct. what i meant was that in parallel you can do matrix multiplication in $\log n$

12/10/2025, 6:23:05 PM

Thought process: Reconsidered parallel matrix multiplication depth versus total work distinction.

The user is pointing out that matrix multiplication itself can be parallelized. Let me think about this more carefully.

For multiplying two $n \times n$ matrices:

- Total work: $O(n^3)$ (or $O(n^\omega)$ with fast algorithms)
- Parallel depth: $O(\log n)$ - because each entry is an inner product of two vectors, which can be done in $O(\log n)$ depth with a parallel reduction tree

So for the naive recursion $x_{k+1} = Ax_k + Bu_k$:

The issue is that we have a sequential dependency: $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \dots \rightarrow x_L$

Each step requires:

- Matrix-vector product Ax_k : This is n inner products, each of length n
 - Work: $O(n^2)$
 - Depth: $O(\log n)$ (parallel reduction for inner products)
- Matrix-vector product Bu_k : n inner products, each of length d
 - Work: $O(nd)$
 - Depth: $O(\log d)$
- Vector addition:
 - Work: $O(n)$
 - Depth: $O(1)$

Since we need to do this for L sequential steps, the critical path is $O(L \cdot \log n)$.

But wait, I think the user is asking about something else. Let me reconsider.

Actually, I think there might be a way to parallelize the recursion using parallel prefix/scan algorithms. Since matrix multiplication is associative, we can use parallel prefix.

The state at time k is:

$$x_k = A^{k-1}B_0 + A^{k-2}B_1 + \dots + B_{k-1}$$

I can represent this augmented system as a matrix transformation. By introducing an additional dimension, we create a structured way to track the state progression, allowing for potential parallel computation strategies. This approach transforms the sequential recursion into a more parallelizable matrix operation, potentially reducing computational bottlenecks.

The parallel prefix method offers an elegant solution for efficiently computing matrix sequences. By strategically multiplying matrices in logarithmic parallel steps, we can dramatically reduce computational complexity. The technique enables computing all state transformations with a critical path of $O(\log L \cdot \log n)$, making large-scale state calculations more tractable.

The computational work remains $O(L \cdot n^3)$, with matrix multiplications being the primary computational constraint. This approach provides a powerful framework for analyzing complex dynamical systems with potential parallel processing advantages.

Turning attention to the convolution approach reveals additional nuanced computational considerations. Matrix-vector multiplication depth and work become critical metrics, with inner product calculations scaling logarithmically with input dimension. Parallel reduction techniques further optimize the computational strategy, offering a depth of $O(\log L + \log d)$ for multiple output computations.

The problem demands a comparative analysis between convolution and direct recursion methods, examining parallelization potential and critical path length. FFT-based convolution introduces logarithmic depth transformations, with element-wise operations providing constant-time complexity.

Dimensional scaling introduces additional computational complexity, requiring careful analysis of work and depth across input and output spaces.

The naive recursion approach reveals intriguing parallel computation possibilities. Matrix-vector products can achieve logarithmic depth, transforming the sequential $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_L$ dependency into a potentially more efficient computation strategy.

Parallel prefix techniques emerge as a powerful optimization mechanism, potentially reducing critical path length from $O(L \cdot \log n)$ to $O(\log L \cdot \log n)$ through sophisticated matrix product decomposition.

For the convolution method, parallel computation becomes even more compelling. By leveraging $O(\log d)$ depth matrix-vector products and $O(\log L)$ vector

summation, we can compute all L outputs concurrently, achieving a critical path of $O(\log L + \log d)$.

The naive recursion reveals significant computational inefficiency. With sequential dependencies across L time steps and each matrix-vector product requiring $O(\log n)$ depth, the critical path balloons to $O(L \log n)$.

Comparing methods reveals a stark performance difference. Convolution offers $O(\log L)$ critical path, while naive recursion demands $O(L \log n)$. Work complexity further highlights convolution's advantages, though total computational cost depends on specific matrix multiplication strategies.

Parallel convolution requires computing L^2 interdependent terms, with each matrix-vector product consuming $O(pd)$ resources. This approach yields $O(L^2 pd)$ total work and maintains a lean $O(\log L)$ depth.

FFT convolution provides an alternative, trading work complexity for potentially more efficient computation at $O(pd \cdot L \log L)$ with consistent $O(\log L)$ depth.

The key insight emerges in understanding how parallelism and sequential dependencies dramatically impact computational efficiency across different recursive strategies. I see the complexity nuances for parallel convolution computation. The critical path depends on matrix multiplication depth, which is $O(\log n)$. For the convolution method, we achieve $O(\log L)$ critical path, while naive recursion requires $O(L \log n)$ depth. Total work remains constant across parallel approaches - $O(pd \cdot L \log L)$ for FFT convolution, with each dimension contributing computational complexity. The key distinction lies in how operations are distributed across parallel processors, not in reducing total computational steps.

The matrix multiplication's parallel characteristics reveal an interesting computational trade-off: while depth can be logarithmic, the actual work remains quadratic or cubic. This means parallel algorithms optimize time complexity without fundamentally changing the total number of operations required.

My analysis suggests the user wants clarity on distinguishing between parallel depth and total computational work, emphasizing that parallelism changes execution time but not total computational complexity.

For different input dimensions, the work scales proportionally: scalar inputs have $O(L \log L)$ work, while d -dimensional inputs increase this to $O(dL \log L)$. The state-space model introduces additional complexity with matrix kernels, requiring more nuanced computational analysis.

The key insight is tracking how dimensionality and kernel structure impact the overall computational complexity, particularly when transitioning from scalar to multi-dimensional scenarios.

For the full state-space model, the convolution involves $p \cdot d$ independent 1D convolutions, each with $O(L \log L)$ FFT work. This results in a total computational complexity of $O(pd \cdot L \log L)$.

The naive recursion approach reveals computational costs for matrix operations: state transition, input processing, and output generation. When dimensions are roughly equivalent, the FFT method becomes more efficient, scaling as $O(d^2 L \log L)$ compared to the naive approach's quadratic complexity.

Critically, the computational trade-offs involve work versus parallel depth, with each matrix operation having logarithmic parallel complexity while maintaining consistent overall work.

You're absolutely right. Let me redo this analysis more carefully, properly distinguishing **work** (total operations) from **depth** (critical path), and accounting for the fact that matrix-vector multiplication has $O(\log n)$ depth when parallelized.

Part (c) Revised: Proper Parallel Complexity Analysis

Key Parallel Costs for Basic Operations

Operation	Work	Depth
Matrix-vector product ($n \times n$) \times ($n \times 1$)	$O(n^2)$	$O(\log n)$
Vector addition (length n)	$O(n)$	$O(1)$
Summing L vectors via parallel reduction	$O(Ln)$	$O(\log L)$

Method 1: Convolution with Known Kernel K

For $y_k = \sum_{\ell=0}^L K_\ell u_{k-\ell}$ where K_ℓ is $p \times d$:

Computing a single output y_k :

- L matrix-vector products $K_\ell u_{k-\ell}$ (all in parallel):
 - Work: $O(Lpd)$
 - Depth: $O(\log d)$

- Sum L resulting p -vectors via parallel reduction:
 - Work: $O(Lp)$
 - Depth: $O(\log L)$

For all L outputs (computed in parallel):

Metric	Value
Work	$O(L^2pd)$
Depth	$O(\log L + \log d)$

With FFT: Work reduces to $O(pd \cdot L \log L)$, Depth remains $O(\log L)$

Method 2: Naive Direct Recursion

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

The sequential dependency $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_{L-1}$ **cannot be parallelized.**

Per time step:

- Ax_k : Work $O(n^2)$, Depth $O(\log n)$
- Bu_k : Work $O(nd)$, Depth $O(\log d)$
- Addition: Work $O(n)$, Depth $O(1)$

For all L steps (sequential):

Metric	Value
Work	$O(L(n^2 + nd))$
Depth	$O(L \cdot \log n)$

Summary

Method	Work	Depth (Critical Path)
Direct Parallel Convolution	$O(L^2pd)$	$O(\log L)$
FFT Convolution	$O(pdL \log L)$	$O(\log L)$
Naive Recursion	$O(Ln^2)$	$O(L \log n)$

Key insight: The naive recursion may have **less total work** (when n is small relative to L), but its **depth is exponentially worse**: $O(L \log n)$ vs $O(\log L)$. Each sequential step contributes $O(\log n)$ to the critical path due to the parallelized matrix-vector multiplication.

12/10/2025, 6:24:18 PM



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

Solution: Once the convolution kernel K is known, computing the discrete convolution $\{y\} = K * \{u\}$ can be performed by zero-padding in $\mathcal{O}(\log L \log n)$ time (critical path length of the computation graph assuming matrix multiplication takes $\mathcal{O}(\log n)$).

In contrast, naive unrolling would require, summing over all previous inputs $\ell \leq k$. Over $k = 1, \dots, L$, this leads to at least $\mathcal{O}(L)$ recurrent steps which cannot be parallelized, resulting in a total complexity of $\mathcal{O}(L \log n)$ (critical path length).

unfortunately, this is not correct. here is the correct solution

12/10/2025, 6:29:54 PM

Thought process: Recalibrated critical path analysis for convolution algorithms.

The user is showing me the correct solution for part (c). Let me analyze it carefully.

The correct solution states:

1. **With known kernel K (FFT convolution):** Critical path length is $\mathcal{O}(\log L \log n)$
 - This accounts for the FFT having depth $\mathcal{O}(\log L)$, and each step involving matrix operations that have depth $\mathcal{O}(\log n)$
2. **Naive unrolled recursion:** Critical path length is $\mathcal{O}(L \log n)$
 - There are $\mathcal{O}(L)$ sequential recurrent steps
 - Each step involves matrix-vector multiplication with depth $\mathcal{O}(\log n)$
 - Total critical path: $\mathcal{O}(L \log n)$

So my earlier analysis was close but I had:

- FFT convolution: $\mathcal{O}(\log L)$ - this was missing the $\log n$ factor
- Naive recursion: $\mathcal{O}(L \log n)$ - this was correct

The key insight I missed is that in the FFT-based convolution, there are $\mathcal{O}(\log L)$ levels in the FFT butterfly network, and at each level, we're doing operations that involve matrix multiplications (or at least operations of similar complexity), each contributing $\mathcal{O}(\log n)$ to the depth.

So the correct answers are:

- **FFT Convolution:** $\mathcal{O}(\log L \log n)$ critical path

- **Naive Recursion:** $O(L \log n)$ critical path

This makes sense - the FFT approach is better by a factor of $L / \log L$ in the critical path.

Thank you for the correction. Let me understand the proper solution:

Part (c): Correct Analysis

With Known Kernel K (FFT Convolution)

The discrete convolution $\{y\} = K * \{u\}$ can be computed via FFT with zero-padding.

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

- The FFT has $O(\log L)$ levels in its butterfly network
 - At each level, matrix operations contribute $O(\log n)$ depth
 - Total: $O(\log L \cdot \log n) = O(\log L \log n)$
-

Naive Unrolled Recursion

Computing y_k directly from $x_{k+1} = Ax_k + Bu_k$ requires summing over all previous inputs $\ell \leq k$.

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

- There are $O(L)$ recurrent steps that **cannot be parallelized** (sequential dependency)
 - Each step requires matrix-vector multiplication with depth $O(\log n)$
 - Total: $O(L \cdot \log n) = O(L \log n)$
-

Summary

Method	Critical Path
FFT Convolution	$O(\log L \log n)$
Naive Recursion	$O(L \log n)$

Key insight: The FFT approach achieves an exponential improvement in parallelism, reducing the critical path by a factor of $L / \log L$ compared to the sequential recursion.