

[Homework 8.pdf](#)
[PDF](#)

Hello! I am looking for you to answer some questions for a deep learning homework assignment. You will only be focusing on all the no coding questions (Questions 1, 3, and 4). Please provide clear, precise, and detailed answers. You will be answering subpart by subpart. Please begin by answering question 1.a

Below is your **answer to Question 1(a)**, written clearly, precisely, and fully derived.
 (Everything is based only on the homework file you uploaded. [Homework 8](#))

1(a) – Convolution Kernel and the Output Equation

We are given the discrete-time SSM:

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

with **sequence length L** , **input (u_0, \dots, u_L)** , **output (y_0, \dots, y_L)** , and **initial state $x_0 = 0$** .

The goal is to show that:

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

where negative-index inputs are treated as zero, and to **derive the kernel K** .

Step 1 – Unroll the state recursion

Starting from $x_0 = 0$:

$$\begin{aligned} x_1 &= Ax_0 + Bu_0 = Bu_0, \\ x_2 &= Ax_1 + Bu_1 = ABu_0 + Bu_1, \\ x_3 &= Ax_2 + Bu_2 = A^2Bu_0 + ABu_1 + Bu_2. \end{aligned}$$

You should see the pattern:

$$x_k = \sum_{\ell=0}^{k-1} A^\ell B u_{k-1-\ell}.$$

To write this with the same indexing convention as convolution:

$$x_k = \sum_{\ell=1}^k A^{\ell-1} B u_{k-\ell}.$$

Step 2 – Substitute into $y_k = Cx_k + Du_k$

$$\begin{aligned} y_k &= C \left(\sum_{\ell=1}^k A^{\ell-1} B u_{k-\ell} \right) + Du_k \\ &= \sum_{\ell=1}^k CA^{\ell-1} B u_{k-\ell} + Du_k \end{aligned}$$

Now rewrite this as a convolution:

- The term multiplying u_k (i.e., when $\ell = 0$) is D .
- For $\ell \geq 1$, the multiplier is $CA^{\ell-1}B$.

So we define the kernel:

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

Thus:

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

with the understanding that $u_{k-\ell} = 0$ if $k - \ell < 0$.

Final Form of the Kernel

$K = \{K_0, K_1, \dots, K_L\}, \quad K_0 = D, \quad K_\ell = CA^{\ell-1}B \text{ for } \ell \geq 1.$

Correct on first try
with reasonable work shown

This shows that the SSM produces its outputs by **convolving** the input sequence with this impulse-response kernel.

When you're ready, say "**Please continue with 1(b)**" and I'll proceed.

Sources



Great job! Please continue with 1(b).

Thought for 21s >

Here's **Question 1(b)** step by step. Homework 8

Recall from 1(a):

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

1(b)(i) Scalar Case

Given:

- $n = 1$ (scalar state),
- $A = \alpha = 0.8$,
- $B = \beta = 1$,
- $C = \gamma = 1.5$,
- $D = \delta$ (kept symbolic),
- compute kernel up to $L = 4$: K_0, K_1, K_2, K_3, K_4 .

For scalars, the kernel formula becomes:

$$K_0 = D = \delta,$$

$$K_\ell = CA^{\ell-1}B = \gamma\alpha^{\ell-1}\beta, \quad \ell \geq 1.$$

Plugging in $\gamma = 1.5$, $\alpha = 0.8$, $\beta = 1$:

- $\ell = 1$:

$$K_1 = 1.5 \cdot 0.8^0 \cdot 1 = 1.5.$$

- $\ell = 2$:

$$K_2 = 1.5 \cdot 0.8^1 \cdot 1 = 1.5 \cdot 0.8 = 1.2.$$

- $\ell = 3$:

$$K_3 = 1.5 \cdot 0.8^2 = 1.5 \cdot 0.64 = 0.96.$$

- $\ell = 4$:

$$K_4 = 1.5 \cdot 0.8^3 = 1.5 \cdot 0.512 = 0.768.$$

Answer for 1(b)(i):

$K_0 = \delta, K_1 = 1.5, K_2 = 1.2, K_3 = 0.96, K_4 = 0.768$

correct on first try
minor note: chat GPT included
 $K_0=8$ whereas the solution key implies $8=0$ and does not include it.

This is an exponentially decaying kernel (because $|A| = 0.8 < 1$), scaled by C .

1(b)(ii) 2D Case

Given:

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

We need the kernel up to $L = 3$: K_0, K_1, K_2, K_3 .

Step 1 — Use the general formula

$$K_0 = D = 0,$$

$$K_\ell = CA^{\ell-1}B \quad (\ell \geq 1).$$

Each K_ℓ is a scalar here, since:

- C is 1×2 ,
- $A^{\ell-1}$ is 2×2 ,
- B is 2×1 .

Step 2 — Compute powers and products

- For $\ell = 1$: $A^0 = I$

$$A^0 B = IB = B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad K_1 = C(A^0 B) = [1 \quad 0] \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1.$$

- For $\ell = 2$: use $A^1 = A$

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

Then:

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

- For $\ell = 3$: we need $A^2 B$. First compute A^2 :

$$A^2 = AA = \begin{bmatrix} 0.51 & 0.13 \\ 0.26 & 0.38 \end{bmatrix}.$$

Then:

$$A^2 B = \begin{bmatrix} 0.51 & 0.13 \\ 0.26 & 0.38 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.51 \\ 0.26 \end{bmatrix}.$$

So:

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

Thus:

$K_0 = 0, \quad K_1 = 1, \quad K_2 = 0.7, \quad K_3 = 0.51.$

Correct on first try with reasonable work shown

How does this kernel capture the “impulse response”?

Think of an **impulse input**: $u_0 = 1$, and $u_k = 0$ for all $k \geq 1$.

Then the output sequence is:

$$y_0 = K_0 u_0 = 0, \quad y_1 = K_1 u_0 = 1, \quad y_2 = K_2 u_0 = 0.7, \quad y_3 = K_3 u_0 = 0.51, \dots$$

So the kernel values $(K_0, K_1, K_2, K_3, \dots)$ are exactly the **impulse response** of the SSM:

- The fact that $K_0 = 0$ reflects that the output has **no direct feedthrough** ($D = 0$); the state must update first before you see any output.
 - $K_1 = 1$ tells you the immediate response at the next time step to the impulse.
 - $K_2 = 0.7, K_3 = 0.51$, etc., show how the effect of that single impulse **decays over time**, governed by the eigenvalues and structure of A . Because the spectral radius of A is < 1 , the impulse response decays, showing the system is stable and has finite memory.
- correctly emphasized key concept

So the kernel K is the discrete-time impulse response of the state-space system.

If you'd like, say "**Please continue with 1(c)**" and I'll move on to the next subpart.

Sources



Awesome work! Please continue with 1(c).

Thought for 32s >

Nice, let's do 1(c). Homework 8

We now know

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

with zero-padding for negative indices. Think of this as a **1D convolution** of the input sequence $\{u_k\}$ with the kernel $\{K_\ell\}$.

1(c) – Efficient Computation with Convolutions

We're asked:

If we already know the kernel K , how much can we parallelize computation of $\{y_k\}$ for an input $\{u_k\} \in \mathbb{R}^d$ 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?

Step 1: Convolution view and parallelism

For each k ,

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

(only up to k terms are nonzero, because earlier u 's are zero).

To compute a **single** y_k :

1. Compute the products $v_\ell = K_\ell u_{k-\ell}$ for all $\ell \in \{0, \dots, k\}$.
 - With enough processors, these $k + 1$ matrix–vector multiplications can be done **in parallel** (depth $O(1)$ ignoring the internal parallelization across dimensions).
2. Sum the $k + 1$ vectors v_0, v_1, \dots, v_k .
 - Use a **tree reduction**: pairwise add vectors, then pairwise add those sums, etc.
 - This yields depth $O(\log k)$, and in the worst case $k \leq L$, so depth $O(\log L)$.

Now, important point: **different outputs y_k do not depend on each other** once K is known. So:

- We can compute all y_0, \dots, y_L in parallel.
- Each one has depth $O(\log L)$ (for its internal reduction).
- Therefore, the **overall critical path length** for computing the entire output sequence by convolution is:

$\text{critical path} = O(\log L)$

(ignoring vector dimension d and treating per-step matrix–vector arithmetic as parallelizable).

If you want to mention FFT-based convolution: that also yields work $O(L \log L)$ and depth $O(\log L)$, consistent with the above parallel reduction picture.

Step 2: Naive direct computation via unrolled recursion

Now compare to the **original SSM recursion**:

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

Here:

- x_{k+1} depends on x_k ; x_k depends on x_{k-1} ; and so on, back to x_0 .
- That is an **inherently sequential dependence in time**.

Even if you parallelize each matrix–vector multiply internally, you **cannot compute x_{k+1} before computing x_k** . So:

- You must perform $L + 1$ sequential time steps.
- Each step has some constant-depth cost (or $O(\log n)$ if you parallelize the matrix–vector arithmetic), but the time dimension itself is strictly sequential.

Hence, the naive unrolled forward computation has a **critical path**:

$\text{critical path (naive recursion)} = O(L)$

(up to multiplicative factors from matrix/vector sizes).

Summary for 1(c)

- **With kernel K and convolution:**
 - All y_k can be computed in parallel.
 - Each y_k is a sum of up to $L + 1$ terms, reducible in a tree in $O(\log L)$ depth.
 - **Minimum critical path length:** $O(\log L)$.
- **With naive unrolled recursion $x_{k+1} = Ax_k + Bu_k$:**
 - Time steps are strictly sequential.
 - **Critical path length:** $O(L)$.

incorrect on first try, did not account for log in both cases. on the right track, however.

So knowing the kernel lets you trade a sequential $O(L)$ -depth recurrence for an $O(\log L)$ -depth parallel convolution.

Sources



I tried to hint chat GPT in the direction of the solution by mentioning a part of the missing component.

Super close! We are analyzing the minimum critical path length for a system where the internal operations involve $n \times n$ matrices. Please re-evaluate the minimum critical path length for the following two methods, expressing the final complexity in terms of both L and n .

*↑
tried to note it should include n in the answer so it doesn't disregard the prior hint/think it is not necessary*

Thought for 22s >

Nice catch—that's exactly the subtlety they're fishing for. Let's redo 1(c) with **explicit $n \times n$ costs** and express the **critical path** in terms of both L and n . □ Homework 8

Assumptions (standard in this style of question):

- State dimension is n , so the key internal operations are **$n \times n$ matrix–vector or matrix–matrix ops.**
 - We're interested in **parallel depth / critical path**, not total work.
 - On a parallel machine:
 - A length- n dot product or vector sum can be done in $O(\log n)$ depth using a tree reduction.
 - An $n \times n$ matrix–vector multiply is then $O(\log n)$ depth (all dot products in parallel).
 - Vector additions are also $O(\log n)$ depth.
-

1(c) – Re-evaluated Critical Path Length

We compare:

1. **Convolution-based method** (kernel K already known).
2. **Naive unrolled recursion** using $x_{k+1} = Ax_k + Bu_k$.

1. Convolution method (kernel known)

We have:

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

and each K_ℓ is an $n \times n$ matrix; treat each $u_{k-\ell}$ as an n -dimensional vector.

For a **single** output y_k :

1. Matrix–vector multiplies

Compute all products $v_\ell = K_\ell u_{k-\ell}$ in parallel across ℓ (assuming enough processors).

- Each $K_\ell u_{k-\ell}$ is an $n \times n$ matrix–vector multiply \rightarrow depth $O(\log n)$.
- Doing all $(L + 1)$ of them in parallel doesn't increase depth, so this stage has:

$$\text{depth} = O(\log n).$$

2. Summing the $(L + 1)$ vectors

Now sum $\{v_0, \dots, v_L\}$ to get y_k . Each vector has n coordinates.

- For each coordinate, sum $L + 1$ scalars with a tree \rightarrow depth $O(\log L)$.
- All coordinates can be reduced in parallel, so the reduction stage depth is:

$$\text{depth} = O(\log L).$$

Since we can compute **all** outputs y_0, \dots, y_L in parallel (once K is known), the critical path for the whole sequence is the max of these two stages:

$\text{critical path (convolution)} = O(\log n + \log L) = O(\log(nL)).$

2. Naive unrolled recursion

We use the original SSM update:

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

Here:

- At time step k , the dominant operation is Ax_k , an $n \times n$ matrix–vector multiply.
 - Depth per step: $O(\log n)$.
- Then we add Bu_k and form y_k , which is also vector ops with depth $O(\log n)$.

But crucially, **time steps are sequential**:

- x_{k+1} depends on x_k , which depends on x_{k-1} , etc.
- You cannot compute x_{k+1} until x_k is done, so you must do all L steps in order.

So with L time steps, each with depth $O(\log n)$, the overall critical path is:

critical path (naive recursion) = $O(L \log n)$.

Final comparison (with $n \times n$ structure explicit)

- Convolution (kernel known):

critical path = $O(\log n + \log L) = O(\log(nL))$.

- Naive unrolled recursion:

critical path = $O(L \log n)$.

It is now half correct, still needed a bit more guidance on the first approach (it added instead of multiplying)

So in terms of parallel depth, convolution collapses the **time dimension** from linear $O(L)$ to logarithmic $O(\log L)$, with only an additional $O(\log n)$ factor from the $n \times n$ internal structure.

Sources



guided it to
explicitly understand
the steps we go
through and how
they run/are
organized

Great job on the naive unrolling! Let's fix the convolution approach. Think about how the steps are organized: The sequence is reduced in parallel using $O(\log L)$ stages. At each of those stages, a matrix operation that takes $\log(n)$ parallel time must occur.

Thought for 10s >

Nice, you're exactly right to push on how the stages compose. Let's fix the convolution analysis with that structure in mind.

We're in the regime where:

- State dimension = n .
- Key internal ops are $n \times n$ matrix–vector or matrix–matrix ops.

- A single $n \times n$ matrix–vector or matrix–matrix multiply takes $O(\log n)$ parallel time (using tree reductions across coordinates).
- The convolution is implemented via a **divide-and-conquer / tree-style reduction** over the sequence, which has $O(\log L)$ stages.

So think of the computation like this:

1. We treat the sequence computation for all outputs as a big associative “reduce” over time, organized as a **binary tree** over the length L .
2. That tree has $O(\log L)$ levels.
3. At **each level**, we have to perform the appropriate $n \times n$ (or $n \times n$ with n -vector) ops to combine partial results.
4. Within a single level:
 - All these matrix ops (across different time segments) are independent and can be done in parallel.
 - So the **depth per level** is governed by the cost of **one** such matrix op: $O(\log n)$.

But the levels themselves are **sequential**, because level $\ell + 1$ depends on the outputs of level ℓ . So the critical path is:

$$\text{critical path (convolution)} = \underbrace{O(\log L)}_{\{\text{# of levels}\}} \times \underbrace{O(\log n)}_{\{\text{depth per level}\}} = \boxed{O(\log L \cdot \log n)}.$$

Compare to what we already got right:

- **Naive unrolled recursion:**

$$\boxed{\text{critical path (naive)} = O(L \log n)}.$$

So the final story is:

- **Convolution-based SSM (kernel known):**

$$\boxed{\text{critical path} = O(\log L \cdot \log n)}.$$

- **Naive unrolling in time:**

$$\boxed{\text{critical path} = O(L \log n)}.$$

on the third try, it got the entire subpart correct. It really needed guidance, especially on the convolution approach

The key correction: because each of the $O(\log L)$ reduction stages **must** happen one after another, and each stage requires $O(\log n)$ parallel time for its matrix ops, the depths **multiply**, not just add.

Prompt guided it to recognize its mistake and correct its logic



Great work! Please now proceed with 1(d).

Thought for 1m 5s >

Nice, let's tackle 1(d) carefully. Homework 8

Recall the kernel form from 1(a):

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

So the main job is to compute the vectors

$$v_\ell := A^{\ell-1}B \in \mathbb{R}^n, \quad \ell = 1, \dots, L,$$

and then apply C to get $K_\ell = Cv_\ell$.

We assume:

- State dimension = n .
 - $L = 2^N$ for simplicity.
 - An $n \times n$ matrix–vector or matrix–matrix multiply takes $O(\log n)$ parallel time.
-

Naive kernel computation (for comparison)

Naively, you'd just iterate:

- $v_1 = B$,
- $v_{\ell+1} = Av_\ell$ for $\ell = 1, \dots, L-1$,
- $K_\ell = Cv_\ell$.

This is **sequential in ℓ** (each step uses the previous):

- Each step has depth $O(\log n)$ (matrix–vector multiply).
- There are L steps.

So the **critical path** is:

naive kernel computation depth = $O(L \log n)$.

We want something asymptotically better by parallelizing over time.

Efficient kernel computation via divide-and-conquer

Key idea: the sequence $\{v_\ell\}$ satisfies a *linear recurrence*:

$$v_{\ell+1} = Av_\ell.$$

This is like repeatedly applying the same linear map A . That's exactly the kind of thing you can speed up with **divide-and-conquer / scan** when $L = 2^N$.

Step 1: Precompute powers of A (repeated squaring)

First, compute:

$$A^1, A^2, A^4, \dots, A^{2^{N-1}}$$

using exponentiation by squaring:

- $A^2 = A \cdot A,$
- $A^4 = (A^2)^2,$
- $\dots,$
- $A^{2^{j+1}} = (A^{2^j})^2.$

Each squaring is an $n \times n$ by $n \times n$ multiply:

- Depth per squaring: $O(\log n)$.
- Number of squarings: $N = \log_2 L$.

So:

$\text{depth to precompute powers} = O(\log L \cdot \log n).$

We'll reuse these powers of A to propagate B in a hierarchical way.

Step 2: Divide-and-conquer over the time axis

Think of the indices $1, \dots, L$ as a segment of length 2^N .

Define $v_\ell = A^{\ell-1}B$. We want all v_1, \dots, v_L .

Use a recursive scheme:

- At each recursive node, you handle a **contiguous block** of indices of length 2^k .
- For a block of length 2^k , you know:
 - The **starting vector** v_{start} at the block's first index.
 - The **transition matrix** for the entire block: A^{2^k} (one of the precomputed powers).

Recursion structure

Let the full interval be indices $[1, 2^N]$.

1. Top level:

- For the whole block of length 2^N , the starting vector is $v_1 = B$.
- Split into left half $[1, 2^{N-1}]$ and right half $[2^{N-1} + 1, 2^N]$.

2. Left half $[1, 2^{N-1}]$:

- Starting vector is v_1 .
- Recursively compute $v_1, \dots, v_{2^{N-1}}$ inside this half.

3. Right half $[2^{N-1} + 1, 2^N]$:

- Its first vector is:

$$v_{2^{N-1}+1} = A^{2^{N-1}} v_1 = A^{2^{N-1}} B,$$

where $A^{2^{N-1}}$ is already precomputed.

- With that as the block's starting state, recursively compute the rest of the vectors in this half using the *exact same recurrence*.

At each recursion level:

- You have two halves; you can process both **in parallel** once you know each half's starting vector.
- To compute the starting vector for the right half, you do a single multiply by the block's transition matrix $A^{2^{k-1}}$:

$$v_{\text{mid}+1} = A^{2^{k-1}} v_{\text{start}}.$$

That multiply is an $n \times n$ matrix–vector multiply, so **depth** $O(\log n)$.

Depth analysis

Let $T(N)$ be the depth to compute all v_ℓ for a block of length 2^N , assuming we already have all needed powers of A .

Recurrence:

- To handle length 2^N :
 - Compute the starting vector for the right half using one matrix–vector multiply: depth $O(\log n)$.
 - Recurse on left and right halves in parallel; each half is length 2^{N-1} .

So:

$$T(N) = T(N - 1) + O(\log n).$$

With base $T(0) = O(\log n)$ (for a single vector), this solves to:

$$T(N) = O(N \log n) = O(\log L \cdot \log n).$$

Once all v_ℓ are known, computing all kernel entries is just:

$$K_\ell = Cv_\ell,$$

which is another matrix–vector multiply per ℓ . These can all be done in parallel with depth $O(\log n)$, which does not change the asymptotic depth we already have.

So the **critical path** for the kernel computation is:

$\text{critical path (efficient kernel computation)} = O(\log L \cdot \log n).$

Strategies summarized

So, given A, B, C and sequence length $L = 2^N$:

1. Exploit the structure $K_\ell = CA^{\ell-1}B$.
2. Precompute powers of A by repeated squaring:
depth $O(\log L \cdot \log n)$.
3. Use a **divide-and-conquer** / segment-tree / scan style recursion:
 - Each node represents a block of time steps,
 - Uses precomputed A^{2^k} to jump the starting state to the midpoint,
 - Recurses into left and right halves in parallel.
4. Total **critical path** to get all K_ℓ : $O(\log L \log n)$, vs naive $O(L \log n)$.

Correct on first try
Captured the correct ideas
Such as divide-and-conquer
and recursion

This is exactly the same spirit as parallel prefix/scan or S4-style fast SSM kernel computation: compress the time dependence from linear in L down to logarithmic depth using hierarchical composition of linear maps.

If you're ready, say "**Please continue with 1(e)**" and I'll move on to leveraging diagonal A .

Sources



Please continue with 1(e).

Thought for 20s >

Nice, this is the fun "structure helps" part. Homework 8

We're now in **1(e)**:

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

Recall the kernel formula:

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

Let the state dimension be n . Assume:

- $A \in \mathbb{R}^{n \times n}$ is **diagonal**:

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

- $B \in \mathbb{R}^{n \times d_{\text{in}}}$,
- $C \in \mathbb{R}^{d_{\text{out}} \times n}$.

1. What does diagonal A give us?

For diagonal A ,

$$A^{\ell-1} = \text{diag}(\lambda_1^{\ell-1}, \dots, \lambda_n^{\ell-1}).$$

So:

$$K_\ell = CA^{\ell-1}B = C \operatorname{diag}(\lambda_1^{\ell-1}, \dots, \lambda_n^{\ell-1})B.$$

Instead of a general matrix–matrix or matrix–vector multiply, $A^{\ell-1}$ acts by **elementwise scaling** on each coordinate of the state:

- If $v_\ell := A^{\ell-1}B$, then row i of v_ℓ is

$$(v_\ell)_{i,:} = \lambda_i^{\ell-1} B_{i,:}.$$

So all the “hard” $n \times n$ matrix–vector multiplies are replaced by **per-coordinate scalar multiplications**.

2. Efficient computation strategy

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

Step A: Precompute powers of each λ_i

For each $i \in \{1, \dots, n\}$, we want the sequence:

$$\lambda_i^0, \lambda_i^1, \dots, \lambda_i^{L-1}.$$

Naive sequential per-dimension recursion:

$$p_{i,0} = 1, \quad p_{i,\ell+1} = \lambda_i p_{i,\ell}$$

would be depth $O(L)$ along the time dimension.

But we can do better with **parallel prefix/scan** or a **divide-and-conquer exponentiation**:

- For each i , the operation is a 1D recurrence over ℓ .
- Using a standard scan over length L , we can generate all powers λ_i^ℓ in $O(\log L)$ depth.
- All n dimensions are independent, so we can do this for all i in parallel with the same $O(\log L)$ depth.

So after this step, we have a table:

$$P \in \mathbb{R}^{n \times L}, \quad P_{i,\ell-1} = \lambda_i^{\ell-1}.$$

Step B: Form $A^{\ell-1}B$

For each ℓ :

- Row i of $A^{\ell-1}B$ is:

$$(A^{\ell-1}B)_{i,:} = \lambda_i^{\ell-1} B_{i,:} = P_{i,\ell-1} \cdot B_{i,:}$$

- That's just **elementwise scaling** of each row of B by the scalar $P_{i,\ell-1}$.

We can perform these scalings:

- For all i and all ℓ in parallel.
- Depth is essentially the cost of a scalar multiply per entry $\rightarrow O(1)$ parallel depth if we treat per-coordinate arithmetic as fully parallel.

So we obtain all intermediate matrices:

$$V_\ell := A^{\ell-1}B \in \mathbb{R}^{n \times d_{\text{in}}}, \quad \ell = 1, \dots, L.$$

Step C: Apply C to get the kernel

Now compute:

$$K_\ell = CV_\ell, \quad \ell = 1, \dots, L.$$

Each is a matrix–matrix or matrix–vector multiply:

- C is $d_{\text{out}} \times n$,
- V_ℓ is $n \times d_{\text{in}}$.

In the simplest case $d_{\text{out}} = d_{\text{in}} = 1$, this is just a dot product of length n .

Parallel depth of this stage:

- For each ℓ , computing K_ℓ is essentially $d_{\text{out}} \times d_{\text{in}}$ many length- n dot products.
- Each dot product can be done in $O(\log n)$ depth via tree reduction.
- All ℓ can be done in parallel, so the depth is:

$$O(\log n).$$

3. Complexity comparison

Naive / general A

- Without structure, kernel computation required repeated $n \times n$ matrix–vector multiplies, leading to:
 - Naive: depth = $O(L \log n)$,
 - Divide-and-conquer: depth = $O(\log L \cdot \log n)$.

With diagonal A

Using the scheme above:

- Step A (powers of λ_i):**

$$\text{depth} = O(\log L)$$

via parallel scan per dimension.

- Step B (scale B to form $A^{\ell-1}B$):**

$$\text{depth} = O(1)$$

(elementwise operations).

- Step C (apply C to get K_ℓ):**

$$\text{depth} = O(\log n).$$

So overall:

$\text{critical path (diagonal } A\text{)} = O(\log L + \log n).$

In many models where elementwise operations are considered $O(1)$ depth and the main bottleneck is along the *time* axis, you can highlight that the dependence on L has been reduced to:

correct time complexity
 $\text{time-depth in } L = O(\log L)$

instead of $O(L)$, and the dependence on n is now only in the final dot with C , not in repeated $n \times n$ matrix applications.

Intuition

- Diagonal A means **each state dimension evolves independently**:

$$x_{k+1,i} = \lambda_i x_{k,i} + B_{i,:} u_k.$$

- The impulse response in each dimension is just a **geometric sequence** $\lambda_i^{\ell-1}$.

- So instead of doing full matrix–vector multiplies, you’re just applying scalar geometric growth per coordinate and then mixing with C at the end.

correct intuition
behind how to leverage
the structure

If you’re ready, say “Please continue with 1(f)” and I’ll handle the diagonal-plus-low-rank (DPLR) case.

Sources



Please continue with 1(f)

Thought for 20s >

Nice, onto the DPLR case. Homework 8

We’re in 1(f):

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

Recall again:

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

So the core task is to efficiently apply **powers of A** to B :

$$v_\ell := A^{\ell-1}B, \quad K_\ell = Cv_\ell.$$

1. Understand how $A = I + pp^\top$ acts

Let

$$s := \|p\|^2 = p^\top p.$$

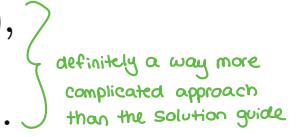
Key geometric fact:

- For any vector x , decompose it as:

$$x = x_{\parallel} + x_{\perp},$$

where

$$x_{\parallel} := \frac{p^{\top}x}{s}p \quad (\text{projection onto the span of } p),$$

$$x_{\perp} := x - x_{\parallel} \quad (\text{component orthogonal to } p).$$


Now apply A to x :

$$Ax = (I + pp^{\top})x = x + p(p^{\top}x).$$

Split it:

- On the **perpendicular** part x_{\perp} :

Since $p^{\top}x_{\perp} = 0$,

$$Ax_{\perp} = x_{\perp}.$$

So eigenvalue **1** on the subspace orthogonal to p .

- On the **parallel** part x_{\parallel} :

$x_{\parallel} = \frac{p^{\top}x}{s}p$. Then

$$Ax_{\parallel} = x_{\parallel} + p(p^{\top}x_{\parallel}) = x_{\parallel} + p\left(p^{\top}\frac{p^{\top}x}{s}p\right) = x_{\parallel} + \frac{p^{\top}x}{s}(p^{\top}p)p = x_{\parallel} + (p^{\top}x)p = (1+s)x_{\parallel}.$$

So A has:

- eigenvalue **1 + s** along p ,
- eigenvalue **1** on the $n - 1$ dimensional orthogonal subspace.

That means powers of A are extremely simple:

$$A^k x = 1^k x_{\perp} + (1 + s)^k x_{\parallel} = x_{\perp} + (1 + s)^k x_{\parallel}.$$

2. Closed form for A^k

We can write this as a matrix formula. Using

$$P := \frac{1}{s}pp^{\top} \quad (\text{projection onto } \text{span}(p)),$$

we have $x_{\parallel} = Px$, $x_{\perp} = (I - P)x$.

Then

$$A^k x = (I - P)x + (1 + s)^k Px = \left[I + ((1 + s)^k - 1)P \right] x.$$

Thus:

$$A^k = I + \frac{(1 + s)^k - 1}{s} pp^\top \quad \text{for } k \geq 0.$$

(When $k = 0$, this gives $A^0 = I$ as expected.)

3. Apply this to B to get the kernel

We need $A^{\ell-1}B$ for $\ell = 1, \dots, L$.

From the formula above:

$$A^{\ell-1}B = \left(I + \frac{(1 + s)^{\ell-1} - 1}{s} pp^\top \right) B = B + \frac{(1 + s)^{\ell-1} - 1}{s} p(p^\top B).$$

Define once:

- Scalar $s = \|p\|^2$.
- Row vector (or $1 \times d_{\text{in}}$ if B has multiple columns):

$$r := p^\top B \in \mathbb{R}^{1 \times d_{\text{in}}}.$$

- Matrix

$$B_{\parallel} := \frac{1}{s} pr = \frac{1}{s} p(p^\top B), \quad B_{\perp} := B - B_{\parallel}.$$

Then

$$A^{\ell-1}B = B_{\perp} + (1 + s)^{\ell-1} B_{\parallel}.$$

So we never have to perform full $n \times n$ multiplies to get powers of A applied to B !

Everything reduces to:

correctly identifies there is no matrix multiplication involved.

- One fixed “parallel” component B_{\parallel} ,
- One fixed “perpendicular” component B_{\perp} ,
- A scalar factor $(1 + s)^{\ell-1}$ that changes with ℓ .

Thus:

$$K_\ell = CA^{\ell-1}B = CB_{\perp} + (1+s)^{\ell-1}CB_{\parallel}, \quad \ell \geq 1, \quad K_0 = D.$$

4. Efficient algorithm & complexity

Let L be the kernel length and n the state dimension.

Precomputation (one-time)

1. Compute $s = p^\top p$.
 - Work: $O(n)$, depth: $O(\log n)$.
2. Compute $r = p^\top B$.
 - If $B \in \mathbb{R}^{n \times d_{\text{in}}}$, this is d_{in} dot products of length n ,
 - Depth: $O(\log n)$ (all dot products in parallel).
3. Form $B_{\parallel} = \frac{1}{s}pr$ and $B_{\perp} = B - B_{\parallel}$.
 - Just outer product & elementwise subtract,
 - Depth: $O(1)$ if we treat elementwise ops as fully parallel.
4. Compute CB_{\parallel} and CB_{\perp} .
 - These are standard matrix–matrix/matrix–vector multiplies,
 - Depth: $O(\log n)$.

Call:

$$U := CB_{\perp}, \quad V := CB_{\parallel}.$$

Then:

$$K_\ell = U + (1+s)^{\ell-1}V.$$

Time-axis dependence

We now only need scalar powers of:

$$\alpha := 1 + s.$$

- Compute $\alpha^0, \alpha^1, \dots, \alpha^{L-1}$ using a parallel prefix/scan style algorithm:
 - Depth $O(\log L)$.
- For each ℓ , compute:

$$K_\ell = U + \alpha^{\ell-1}V,$$

which is just a vector-plus-scaled-vector operation, fully parallel across entries (depth $O(1)$).

Critical path

Putting it all together:

- Precomputation in the **feature dimension**: depth $O(\log n)$.
- Generating scalar powers in the **time dimension**: depth $O(\log L)$.
- Combining to form each K_ℓ : depth $O(1)$.

So:

$$\text{critical path (DPLR } A = I + pp^\top) = O(\log n + \log L).$$

Compare:

- **Generic dense** A with divide-and-conquer:

$$O(\log L \cdot \log n).$$

- **Diagonal** A :

$$O(\log L + \log n).$$

- **DPLR** $A = I + pp^\top$:

$$O(\log L + \log n),$$

using a *single* “special direction” p and scalar powers of $\alpha = 1 + \|p\|^2$.

Intuition

- $A = I + pp^\top$ only changes the state in the 1D subspace spanned by p .
- Everything orthogonal to p stays the same over time; only the projection onto p grows by factor $(1 + s)$ each step.
- That makes A^k essentially a **rank-1 correction to the identity** with a simple scalar power, which we can exploit to get all kernel entries with only:
 - a few $O(\log n)$ precomputations,
 - and $O(\log L)$ depth over time.

If you're ready, you can say something like "**Let's move on to Question 3(a)**" or specify which subpart of Q3/Q4 you want next, and I'll switch gears to the linear purification / ridge-attention parts.

Sources



I wasn't fully sure if the model had the right idea since the approach was way more complex than needed, so I asked for a more condensed summary

Thank you! Can you please summarize your work in 1(f) to describe how we can leverage this structure to compute the kernel K more efficiently?

Here is a clean, concise summary of **how the DPLR structure $A = I + pp^\top$ enables efficient kernel computation**, suitable as a final written answer for 1(f):

Summary for 1(f): Leveraging DPLR Structure to Efficiently Compute the Kernel

When $A = I + pp^\top$ is **diagonal-plus-low-rank (rank 1)**, we can exploit this structure to compute the kernel

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

much more efficiently than with a general dense matrix.

Key Insight: Closed Form for Powers of A

Let $s = \|p\|^2 = p^\top p$.

Since $A = I + pp^\top$, its action splits cleanly:

- On the direction of p : eigenvalue $1 + s$
- On the orthogonal subspace: eigenvalue 1

This yields a closed form for matrix powers:

$$A^k = I + \frac{(1+s)^k - 1}{s} pp^\top.$$

So applying A^k to any matrix or vector requires **only scalar powers of $(1+s)$** and a **rank-1 update**, not full $n \times n$ matrix multiplication.

Apply $A^{\ell-1}$ to B

Using the above formula:

$$A^{\ell-1}B = B + \frac{(1+s)^{\ell-1} - 1}{s} p(p^\top B).$$

Define the constant matrices:

- $r = p^\top B,$
- $B_{\parallel} = \frac{1}{s}pr,$
- $B_{\perp} = B - B_{\parallel}.$

Then the recurrence simplifies to:

$$A^{\ell-1}B = B_{\perp} + (1+s)^{\ell-1}B_{\parallel}.$$

So the “perpendicular” component is constant in time, and only the “parallel” part grows by the scalar factor $(1+s)^{\ell-1}.$

Final Kernel Expression

Plugging into $K_\ell = CA^{\ell-1}B:$

$$K_\ell = CB_{\perp} + (1+s)^{\ell-1}CB_{\parallel}, \quad \text{for } \ell \geq 1.$$

Thus we only need:

- one dot product $p^\top B,$
- one outer product $pr^\top,$
- scalar powers of $(1+s),$
- and matrix–vector multiplications with $C.$

All expensive $n \times n$ matrix multiplications disappear.

Efficiency Improvement

Instead of the general dense-matrix cost:

$$O(\log L \cdot \log n)$$

for computing all kernel terms using divide-and-conquer, the DPLR structure reduces the critical path to:

$$O(\log n + \log L).$$

Why?

- Computing $s = p^\top p$, $p^\top B$, and $CB_{\parallel}, CB_{\perp}$ takes $O(\log n)$ depth.
- Generating scalar powers $(1 + s)^{\ell-1}$ for all ℓ via parallel prefix takes $O(\log L)$ depth.
- Forming each kernel entry is just a scaled vector addition.

So the kernel is obtained using only:

- **rank-1 structure,**
- **scalar exponentiation,**
- **parallel vector operations,**
- **one-time projection computations.**

Final Statement

The DPLR structure makes kernel computation dramatically faster because powers of A reduce to a simple scalar sequence $(1 + s)^k$ affecting only the 1-dimensional subspace spanned by p . This eliminates all expensive $n \times n$ multiplies and yields an $O(\log n + \log L)$ critical path for computing the entire kernel.

↑
Correct in terms of the "spirit" of the question

If you'd like, we can now move on to Question 3(a).



Please move on to question 3(a).

Thought for 27s >

Here we go with **Question 3(a)**. Homework 8

We're given:

- Two candidate encoders (for $m = 2$)

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

- Data matrix

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

- Losses:

$$L_1(W; X) = \|X - WX\|_F^2 \quad (\text{reconstruction loss})$$

$$L_2(W; X, \lambda) = \|X - WX\|_F^2 + \lambda\|W\|_F^2 \quad (\text{reconstruction + regularization}).$$

3(a)(i) – Fill in the table

We need **reconstruction loss** $\|X - WX\|_F^2$ and **regularization loss** $\|W\|_F^2$ for each encoder.

For $W(\alpha) = I_2$

- Reconstruction loss

$$WX = I_2 X = X \quad \Rightarrow \quad X - WX = X - X = 0.$$

So

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

- Regularization loss

$$\|W(\alpha)\|_F^2 = \|I_2\|_F^2 = 1^2 + 1^2 = 2.$$

So for $W(\alpha)$:

- Reconstruction loss: 0
- Regularization loss: 2

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

- Reconstruction loss

First compute $W(\beta)X$:

- First row unchanged (multiplied by 1),
- Second row becomes 0 (multiplied by 0).

So

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

Then

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

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

$$\|X - W(\beta)X\|_F^2 = 0.02^2 + (-0.01)^2 + 0.01^2 + (-0.02)^2 = 0.0004 + 0.0001 + 0.$$

(This matches the 0.001 they already put in the table.)

- Regularization loss

For $W(\beta) = \text{diag}(1, 0)$:

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

So for $W(\beta)$:

- Reconstruction loss: 0.001
- Regularization loss: 1

Filled table

Encoder	Reconstruction Loss	Regularization Loss
α	0	2
β	0.001	1

correct on first try
with reasonable work

3(a)(ii) – For which λ is $W(\alpha)$ worse than $W(\beta)$?

We compare the full regularized objective

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

Compute each:

- For $W(\alpha)$:

$$L_2(W(\alpha); X, \lambda) = 0 + \lambda \cdot 2 = 2\lambda.$$

- For $W(\beta)$:

$$L_2(W(\beta); X, \lambda) = 0.001 + \lambda \cdot 1 = 0.001 + \lambda.$$

We want the range of λ such that the identity $W(\alpha)$ has *higher loss* than $W(\beta)$ **:

$$L_2(W(\alpha); X, \lambda) > L_2(W(\beta); X, \lambda).$$

Plug in:

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

Solve:

$$2\lambda - \lambda > 0.001 \Rightarrow \lambda > 0.001.$$

So:

For all $\lambda > 0.001$, $W(\alpha) = I_2$ has higher regularized loss L_2 than $W(\beta)$.

correct on first
try with reasonable
work

Equivalently, when the regularization is strong enough (λ above 0.001), the model that “kills” the second dimension is preferred over the identity, even though it slightly worsens reconstruction.

Sources

Thought for 1m 7s >

Let's do 3(b) carefully and cleanly. Homework 8

We're minimizing, over square $W \in \mathbb{R}^{m \times m}$,

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

where $X \in \mathbb{R}^{m \times n}$ has SVD

$$X = U\Sigma V^\top, \quad \Sigma = [\text{diag}(\sigma_1, \dots, \sigma_m) \quad 0], \quad \sigma_1 > \dots > \sigma_m \geq 0.$$

3(b)(i) Fill in the expression for W_c

They give the form:

$$W_c = \begin{bmatrix} \square \end{bmatrix} \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & \\ & \ddots & \\ & & \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \end{bmatrix} \begin{bmatrix} \square \end{bmatrix}.$$

The optimizer is:

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

Correct on first try with reasonable work

So the blanks are U on the left and U^\top on the right.

3(b)(ii) Derive this expression

We want to solve:

$$\min_W \|X - WX\|_F^2 + \lambda\|W\|_F^2.$$

Step 1: Move into SVD / orthonormal coordinates

Use the SVD $X = U\Sigma V^\top$. Define:

$$W' := U^\top WU.$$

Since U is orthogonal, we have:

- $\|W\|_F^2 = \|W'\|_F^2$ (Frobenius norm invariant under orthogonal similarity),
- and we can rewrite the reconstruction term as follows:

$$WX = W(U\Sigma V^\top) = (WU)\Sigma V^\top = U(U^\top WU)\Sigma V^\top = UW'\Sigma V^\top.$$

Similarly:

$$X - WX = U\Sigma V^\top - UW'\Sigma V^\top = U(\Sigma - W'\Sigma)V^\top.$$

Using Frobenius norm invariance under left/right orthogonal multiplications:

$$\|X - WX\|_F = \|U(\Sigma - W'\Sigma)V^\top\|_F = \|\Sigma - W'\Sigma\|_F.$$

So the objective becomes:

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

with decision variable $W' = U^\top WU$.

Thus we can minimize over W' instead of W , and recover $W_c = UW'_c U^\top$.

Step 2: Use the structure of Σ

Σ has diagonal entries $\sigma_1, \dots, \sigma_m$ on its first m columns (rest are zero). So we only need to consider the first m columns of Σ .

Let $w^{(i)}$ denote the i -th column of W' (a vector in \mathbb{R}^m).

- The i -th column of Σ (for $1 \leq i \leq m$) is $\sigma_i e_i$, where e_i is the i -th standard basis vector.
- Then the i -th column of $W'\Sigma$ is

$$W'(\sigma_i e_i) = \sigma_i W' e_i = \sigma_i w^{(i)}.$$

Thus the i -th column of $\Sigma - W'\Sigma$ is:

$$\sigma_i e_i - \sigma_i w^{(i)} = \sigma_i(e_i - w^{(i)}).$$

The Frobenius norm squared is the sum of squared column norms:

$$\|\Sigma - W'\Sigma\|_F^2 = \sum_{i=1}^m \|\sigma_i(e_i - w^{(i)})\|_2^2 = \sum_{i=1}^m \sigma_i^2 \|e_i - w^{(i)}\|_2^2.$$

Also,

$$\|W'\|_F^2 = \sum_{i=1}^m \|w^{(i)}\|_2^2.$$

So the objective decomposes as:

$$L_2(W'; X, \lambda) = \sum_{i=1}^m \left[\sigma_i^2 \|e_i - w^{(i)}\|_2^2 + \lambda \|w^{(i)}\|_2^2 \right].$$

Step 3: See the decoupled ridge problems

For each i , we independently minimize over $w^{(i)} \in \mathbb{R}^m$:

$$J_i(w^{(i)}) = \sigma_i^2 \|e_i - w^{(i)}\|_2^2 + \lambda \|w^{(i)}\|_2^2.$$

This is exactly a **ridge regression** style quadratic in $w^{(i)}$:

- “Target” is e_i ,
- “Weight” scaling is σ_i^2 ,
- Regularization coefficient is λ .

Take gradient and set to zero:

$$\nabla_{w^{(i)}} J_i = 2\sigma_i^2(w^{(i)} - e_i) + 2\lambda w^{(i)} = 2(\sigma_i^2 + \lambda)w^{(i)} - 2\sigma_i^2 e_i.$$

Set gradient = 0:

$$(\sigma_i^2 + \lambda)w^{(i)} = \sigma_i^2 e_i \quad \Rightarrow \quad w^{(i)} = \frac{\sigma_i^2}{\sigma_i^2 + \lambda} e_i.$$

So each column $w^{(i)}$ is just a scaled basis vector e_i . That means W' is **diagonal** in this basis:

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

Step 4: Transform back to original coordinates

Finally, return to the original variable $W = UW'U^\top$:

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

This provided a slightly different approach than the solutions outlined in the guide. After reading it, the logic seems to make sense and produce the correct answer.

This shows exactly how the solution acts as a **shrinkage operator in the principal-component directions** of X : directions with large singular values σ_i are kept closer to 1, while small- σ_i directions are heavily shrunk toward 0 as λ grows.

If you'd like, we can move on to 3(c) next.

Sources



Please move on to 3(c).

Nice, this one's pretty mechanical once we use the formula from 3(b). Homework 8

We know the optimal purifier is

$$W_c = U \text{diag}\left(\frac{\sigma_1^2}{\sigma_1^2 + \lambda}, \dots, \frac{\sigma_m^2}{\sigma_m^2 + \lambda}\right) U^\top,$$

so along singular direction i , the "gain" is

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

We're given singular values:

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

They want:

1. First 3 directions ($i = 1, 2, 3$) to be preserved at at least 80%:

$$g_i(\lambda) \geq 0.8.$$

2. Remaining directions ($i = 4, \dots, 8$) to be attenuated to at most 50%:

$$g_i(\lambda) \leq 0.5.$$

Step 1: Constraint for top 3 singular directions

For each $i \in \{1, 2, 3\}$:

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8.$$

Solve for λ :

$$\begin{aligned} \frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8 &\implies \sigma_i^2 \geq 0.8(\sigma_i^2 + \lambda) \\ \implies \sigma_i^2 \geq 0.8\sigma_i^2 + 0.8\lambda &\implies 0.2\sigma_i^2 \geq 0.8\lambda \implies \lambda \leq \frac{0.2}{0.8}\sigma_i^2 = 0.25\sigma_i^2. \end{aligned}$$

Now plug in:

- $i = 1: \sigma_1 = 10 \Rightarrow \sigma_1^2 = 100$
 $\Rightarrow \lambda \leq 0.25 \cdot 100 = 25.$
- $i = 2: \sigma_2 = 8 \Rightarrow \sigma_2^2 = 64$
 $\Rightarrow \lambda \leq 0.25 \cdot 64 = 16.$
- $i = 3: \sigma_3 = 4 \Rightarrow \sigma_3^2 = 16$
 $\Rightarrow \lambda \leq 0.25 \cdot 16 = 4.$

We need all three inequalities simultaneously, so we take the **most restrictive** (smallest upper bound):

$\lambda \leq 4.$

Step 2: Constraint for remaining directions (4–8)

For each $i \in \{4, \dots, 8\}$:

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \leq 0.5.$$

Solve:

$$\begin{aligned}\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \leq 0.5 &\implies \sigma_i^2 \leq 0.5(\sigma_i^2 + \lambda) \\ \implies \sigma_i^2 &\leq 0.5\sigma_i^2 + 0.5\lambda \implies 0.5\sigma_i^2 \leq 0.5\lambda \implies \lambda \geq \sigma_i^2.\end{aligned}$$

Now plug in:

- $i = 4: \sigma_4 = 1 \Rightarrow \sigma_4^2 = 1$
- $i = 5: \sigma_5 = 0.5 \Rightarrow \sigma_5^2 = 0.25$
- $i = 6: \sigma_6 = 0.36 \Rightarrow \sigma_6^2 = 0.1296$
- $i = 7: \sigma_7 = 0.16 \Rightarrow \sigma_7^2 = 0.0256$
- $i = 8: \sigma_8 = 0.01 \Rightarrow \sigma_8^2 = 0.0001$

The **most restrictive lower bound** is the largest of these:

$$\max\{\sigma_4^2, \dots, \sigma_8^2\} = \sigma_4^2 = 1.$$

So we need:

$$\boxed{\lambda \geq 1.}$$

Step 3: Combine both conditions

From the top-3 requirement: $\lambda \leq 4$.

From the bottom-5 requirement: $\lambda \geq 1$.

So the set of λ that satisfies **both** is:

$$\boxed{1 \leq \lambda \leq 4.}$$
correct on first try with reasonable work shown

Any λ in $[1, 4]$ will:

- Keep the first three singular directions at $\geq 80\%$ strength, and
- Shrink all remaining directions to $\leq 50\%$ of their original strength.

[Sources](#)



Great job on Question 3! Please proceed to question 4(a).

For Question 4(a), we're just updating the mean efficiently when a new point arrives.

Homework 8

You're given:

- Current mean of n points:

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

- A new point x_{n+1}
- New mean over $n + 1$ points:

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

We want to express m' using only m , x_{n+1} , and n .

Step-by-step derivation

Start from the definition of m :

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

Now write the new mean:

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

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

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

So the **incremental update formula** is:

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

Correct on first try with reasonable work shown

This shows why the update is efficient: you don't need to store all the x_i ; you just keep the current mean m , the count n , and the new point x_{n+1} .

Sources



Proceed to Question 4(b).

Let's do 4(b). Homework 8

We're given ridge regression with:

- Training pairs (x_i, y_i) , where $x_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$,
- Design matrix

$$A = \begin{bmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_n^\top \end{bmatrix} \in \mathbb{R}^{n \times d},$$

- Response vector

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n.$$

The ridge solution is:

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

They ask us to write both $A^\top A + \lambda I$ and $A^\top y$ as sums over the training examples.

4(b) – Express both terms as sums over (x_i, y_i)

1. $A^\top A + \lambda I$

First expand $A^\top A$. Since rows of A are x_i^\top , we can write:

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

Therefore:

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

So the first blank is:

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

Correct math shown on first try to reach the answer

2. $A^\top y$

Write out the product $A^\top y$:

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

So the second blank is:

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

Correct math shown on first try to reach the answer

So the completed lines they want are:

$$(A^\top A + \lambda I) = \lambda I + \sum_{i=1}^n x_i x_i^\top, \quad A^\top y = \sum_{i=1}^n x_i y_i.$$

Sources



Proceed to Question 4(c).

Thought for 22s >

Let's knock out 4(c). Homework 8

We want the computational cost of a **non-causal ridge self-attention layer** with:

- context length n ,
- d -dimensional queries, keys, and values,
- $d < n$,
- costs:
 - inverting $d \times d$ matrix: $O(d^3)$,
 - multiplying two $d \times d$ matrices: $O(d^3)$,
 - multiplying a $d \times d$ matrix by a d -dim vector: d^2 operations.

Think of ridge self-attention as **multi-dimensional ridge regression**:

- Keys $x_i \in \mathbb{R}^d$ form the matrix $A \in \mathbb{R}^{n \times d}$ (each row x_i^\top).
- Values $v_i \in \mathbb{R}^d$ form a value matrix $Y \in \mathbb{R}^{n \times d}$ (each row v_i^\top).
- We learn a mapping (weight matrix) from keys to values:

$$W^* = (A^\top A + \lambda I)^{-1} A^\top Y \in \mathbb{R}^{d \times d}.$$

Then for each query $q_j \in \mathbb{R}^d$, the attention output is:

$$z_j = W^{*\top} q_j \quad (\text{or } W^* q_j \text{ up to orientation}).$$

We have **the same pool** of keys/values for all queries, so we compute W^* **once**, then apply it to each query.

1. Cost to compute W^*

Use the decompositions from 4(b):

$$A^\top A + \lambda I = \lambda I + \sum_{i=1}^n x_i x_i^\top, \quad A^\top Y = \sum_{i=1}^n x_i v_i^\top.$$

- **Compute $A^\top A$:**

Each outer product $x_i x_i^\top$ is $d \times d \rightarrow O(d^2)$.

Summed over n points: $O(nd^2)$.  correct first step

- **Add λI :** $O(d^2)$ (negligible vs nd^2). correct first step
- **Invert $A^\top A + \lambda I$:** $d \times d$ inverse $\rightarrow O(d^3)$. correct second step
- **Compute $A^\top Y$:**

Each term $x_i v_i^\top$ is $d \times d \rightarrow O(d^2)$.

Over n points: $O(nd^2)$. correct third step

- **Multiply $(A^\top A + \lambda I)^{-1}(A^\top Y)$:**
 $d \times d$ times $d \times d \rightarrow O(d^3)$. correct fourth step

Total to get W^* :

$$O(nd^2) + O(d^3) + O(nd^2) + O(d^3) = O(nd^2 + d^3).$$

2. Cost to apply W^* to all queries

We have n queries, each d -dimensional.

- Per query: $W^* q_j$ is a $d \times d$ matrix–vector multiply $\rightarrow O(d^2)$.
 - For all n queries: $O(nd^2)$.
-

3. Combine and simplify

Total cost:

$$O(nd^2 + d^3) + O(nd^2) = O(nd^2 + d^3).$$

Given the assumption $d < n$, for large n :

- nd^2 dominates d^3 . correctly recognizes this

So asymptotically:

Cost of non-causal ridge self-attention = $O(nd^2)$.

Correct on first try with reasonable logic (logic step by step matches answer)

From the multiple-choice list, the correct answer is:

$$O(nd^2).$$

Sources



[Proceed to Question 4\(d\)](#)

Good news: this one is conceptually simple. Homework 8

We're using **ridge self-attention**, where the forward pass essentially does:

- Build A from the **keys** (rows x_i^\top),
- Build Y from the **values** (rows v_i^\top),
- Compute

$$W^* = (A^\top A + \lambda I)^{-1} A^\top Y,$$

- Then for each **query** q , output

$$z = W^{*\top} q \quad (\text{up to orientation}).$$

Backprop will pass gradients through any parameter or input as long as the forward mapping is differentiable in that argument.

Let's check each one:

1. Ridge λ

Appears inside the matrix inverse:

$$W^* = (A^\top A + \lambda I)^{-1} A^\top Y.$$

Matrix inverse and multiplication are differentiable wrt λ (away from singularities).

Gradients do flow to λ → can treat it as a learnable scalar.

2. Keys

Keys are used to form A , so they affect both:

- $A^\top A + \lambda I$,
- and $A^\top Y$.

These are just sums of outer products $x_i x_i^\top$ and $x_i v_i^\top$, and then matrix inverse + multiplication. All smooth ops.

Gradients do flow to keys.

3. Values

Values form Y , which appears only in $A^\top Y$. Linear dependence → differentiable.

Gradients do flow to values.

4. Queries

The final output is linear in the query:

$$z = W^{*\top} q.$$

This is just a matrix–vector multiply, so clearly differentiable wrt q .

Gradients do flow to queries.

Answer for 4(d)

Backprop successfully passes gradients through all of them:

- The ridge λ
 - The keys
 - The values
 - The queries
- correct on first try with reasonable logic

Sources



Proceed to Question 4(e).

Thought for 12s >

For 4(e) we're just analyzing the cost of applying the Sherman–Morrison update **once**.

Homework 8

We're given:

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

and we're told:

- We already have M^{-1} in hand.
- A $d \times d$ matrix times a d -dimensional vector costs d^2 operations.
- A dyad uv^\top also costs d^2 .
- An inner product costs d .

We want the cost of **one application** of this formula.

Steps and costs

1. Compute $a = M^{-1}u$

- One matrix–vector multiply: cost d^2 . correct first step

2. Compute $b^\top = v^\top M^{-1}$

- Equivalent to $(M^{-1})^\top v$: another matrix–vector multiply: cost d^2 . correct second step

3. Compute the denominator

$$1 + v^\top M^{-1} u = 1 + v^\top a.$$

- Dot product $v^\top a$: cost d . correct third step

4. Form the outer product $(M^{-1}u)(v^\top M^{-1}) = ab^\top$

- Dyad ab^\top : cost d^2 . correct fourth step

5. Scale and subtract from M^{-1}

- Scalar–matrix multiply + matrix subtraction: $O(d^2)$ but same order. not explicitly mentioned in answer key

Dominant cost is clearly **on the order of d^2** .

Answer for 4(e)

The computational cost of one Sherman–Morrison update, given M^{-1} , is:

$$O(d^2).$$

Correct on first try with reasonable logic (logic almost step by step matches answer)

Sources



Proceed to Question 4(f).

Thought for 38s >

Nice, this is the “online ridge regression as attention” part. Homework 8

We want **causal ridge self-attention** with context length n , where at position t we can only use the first t (key, value) pairs. Naively recomputing the ridge solution from scratch at each t would cost $O(n^2)$ overall; we want to avoid that by **updating** things recursively.

Setup: ridge-regression view

From 4(b), with keys $x_i \in \mathbb{R}^d$ and values $y_i \in \mathbb{R}^d$, define:

- $A_t \in \mathbb{R}^{t \times d}$ whose rows are $x_1^\top, \dots, x_t^\top$
- $Y_t \in \mathbb{R}^{t \times d}$ whose rows are $y_1^\top, \dots, y_t^\top$

The **ridge-regression** solution at time t is:

$$W_t^* = (A_t^\top A_t + \lambda I)^{-1} A_t^\top Y_t \in \mathbb{R}^{d \times d}.$$

Given a query $q_t \in \mathbb{R}^d$ at position t , the attention output is:

$$z_t = W_t^{*\top} q_t \in \mathbb{R}^d.$$

We want all z_1, \dots, z_n .

Key trick: maintain *sufficient statistics* and use Sherman–Morrison

From 4(b), define:

$$M_t := A_t^\top A_t + \lambda I = \lambda I + \sum_{i=1}^t x_i x_i^\top \quad (d \times d),$$

$$B_t := A_t^\top Y_t = \sum_{i=1}^t x_i y_i^\top \quad (d \times d).$$

Then:

$$W_t^* = M_t^{-1} B_t.$$

Instead of recomputing from scratch, we maintain:

- $S_t := M_t^{-1}$ (a $d \times d$ matrix),
- B_t (also $d \times d$),

and update them **incrementally** as we move from $t - 1$ to t .

1. Updating M_t^{-1} using Sherman–Morrison

Observe:

$$M_t = \lambda I + \sum_{i=1}^t x_i x_i^\top = M_{t-1} + x_t x_t^\top.$$

This is exactly a rank-1 update:

$$M_t = M_{t-1} + u v^\top \quad \text{with } u = x_t, v = x_t.$$

Given $S_{t-1} = M_{t-1}^{-1}$, Sherman–Morrison says:

$$M_t^{-1} = S_t = S_{t-1} - \frac{(S_{t-1} x_t)(x_t^\top S_{t-1})}{1 + x_t^\top S_{t-1} x_t}.$$

Operationally:

- Compute $u_t = S_{t-1} x_t$ (matrix–vector, $O(d^2)$),
- Compute scalar $c_t = 1 + x_t^\top u_t$ (dot product, $O(d)$),
- Update

$$S_t = S_{t-1} - \frac{1}{c_t} u_t u_t^\top$$

(rank-1 outer product + subtraction, $O(d^2)$).

So updating the inverse costs $O(d^2)$ per step.

Initial condition: $M_0 = \lambda I \Rightarrow S_0 = \frac{1}{\lambda} I$.

2. Updating the cross-term B_t

Similarly,

$$B_t = \sum_{i=1}^t x_i y_i^\top = B_{t-1} + x_t y_t^\top.$$

That's a rank-1 update:

- Compute $x_t y_t^\top$ and add to B_{t-1} :
 - Outer product cost: $O(d^2)$.

So maintaining B_t is also $O(d^2)$ per step.

Initial condition: $B_0 = 0$.

3. Producing the causal attention output at each step

We never need to explicitly form $W_t^* = S_t B_t$ (which would be $O(d^3)$ per step).

Instead, use:

$$z_t = W_t^{*\top} q_t = B_t^\top S_t^\top q_t.$$

Compute it in two matrix–vector multiplies:

1. $u_t = S_t^\top q_t$ (cost $O(d^2)$),
2. $z_t = B_t^\top u_t$ (cost $O(d^2)$).

So producing z_t from (S_t, B_t, q_t) costs $O(d^2)$ time.

Full causal algorithm (high-level)

Initialize:

- $S_0 = \frac{1}{\lambda} I_d$,

- $B_0 = 0_{d \times d}$.

For $t = 1, \dots, n$:

1. **Update statistics with new key, value** (x_t, y_t) :

- Update inverse:
 - $u_t = S_{t-1}x_t$,
 - $c_t = 1 + x_t^\top u_t$,
 - $S_t = S_{t-1} - \frac{1}{c_t}u_t u_t^\top$.
- Update cross-term:
 - $B_t = B_{t-1} + x_t y_t^\top$.

2. **Compute causal ridge-attention output for query** q_t (attending to keys/values up to t):

- $u'_t = S_t^\top q_t$,
- $z_t = B_t^\top u'_t$.

Here, at each step, we used all keys/values $\{(x_i, y_i)\}_{i \leq t}$ (through S_t, B_t), and returned the correct ridge-regression prediction for q_t .

Complexity and why it avoids $O(n^2)$

Per time step t :

- Updating S_t : $O(d^2)$,
- Updating B_t : $O(d^2)$,
- Computing z_t : $O(d^2)$.

So per step cost is $O(d^2)$, and over all n positions:

$\text{Total cost} = O(nd^2),$

which matches the same scaling as standard non-causal ridge attention (from 4(c)), and **avoids** the naïve $O(n^2d^2)$ that would come from recomputing the ridge solution from scratch at every time step.

So the answer they want for 4(f) is:

Maintain the sufficient statistics M_t^{-1} and B_t over time, update M_t^{-1} with **Sherman–Morrison** and B_t with a rank-1 outer product, and compute each output z_t via two matrix–vector multiplies. This yields correct causal ridge self-attention for all t in $O(nd^2)$ time instead of $O(n^2)$.

Correct on first try, aligns with spirit of the question (just uses some slightly different notation)

Sources



Proceed to Question 4(g).

Thought for 16s >

Nice, last one! Homework 8

We want:

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

Think: in ridge-attention, the output at a query is still a **linear combination of the value vectors**. We just need to make that combination explicit.

Setup

Let:

- Keys: $x_i \in \mathbb{R}^d, i = 1, \dots, n$.
- Values: $v_i \in \mathbb{R}^d$.
- Stack keys as rows of $A \in \mathbb{R}^{n \times d}$.
- Stack values as rows of $Y \in \mathbb{R}^{n \times d}$.

Ridge regression solution (for all dimensions of the value at once):

$$W^* = (A^\top A + \lambda I)^{-1} A^\top Y \in \mathbb{R}^{d \times d}.$$

Given a query $q \in \mathbb{R}^d$, the ridge-attention output is:

$$z(q) = W^{*\top} q \in \mathbb{R}^d.$$

Express $z(q)$ as a linear combination of the values

Start from:

$$z(q) = W^{*\top} q = Y^\top A(A^\top A + \lambda I)^{-1} q.$$

correct on first try
(uses slightly different
notation of Y instead
of B)

Now define:

$$a(q) := A(A^\top A + \lambda I)^{-1} q \in \mathbb{R}^n.$$

Then the output can be written as:

$$z(q) = Y^\top a(q) = \sum_{i=1}^n a_i(q) v_i.$$

So the **attention weights** on the value vectors $\{v_i\}$ for query q are exactly the entries of the vector:

PDF was cutoff so here is a screenshot of the rest

$$a(q) = A(A^\top A + \lambda I)^{-1} q \in \mathbb{R}^n,$$

and the output is:

$$z(q) = \sum_{i=1}^n a_i(q) v_i.$$

Those $a_i(q)$ can be **negative** (unlike softmax weights), because they come from a linear ridge-regression solution instead of a probability distribution.

In a **causal** ridge-attention setting at position t , you'd just replace A and Y by their truncated versions A_t and Y_t (first t keys/values) and compute:

$$a_t(q_t) = A_t(A_t^\top A_t + \lambda I)^{-1} q_t,$$

then $z_t = \sum_{i=1}^t a_{t,i} v_i$.

Correct on first try, just did not give as detailed or thoughtful interpretation as the solution guide