Note: I had originally signed up for HWK 7 GPT 5, but someone covered it, without adding to the deconflictation form; so I changed to HWK 8 GPT 5.1 Thinking (to my knowledge it used just thinking not normal mode). My formatting here is based on or at least influenced by participation assignments that other students did.

Non-coding questions

To compare correctness, I used the solutions and then attempted to prompt it further once into getting the correct answer (based on what the solutions said the correct answer was).

Question 1

a) First Try - solved the question
b) First Try - solved the question
c) Missed the O(logn) factor from matrix multiplication on the first try, succeeded the second time
d) Brought up Divide and conquer, it doesn't seem 100% the same as what the solutions have, but I will give this credit. Also it missed the O(logn) factor from matrix multiplication. The second time it got O(logL * Tmm) but Tmm is a matmat it previously said in c's answer should be logn, so total is O(logL * logn) which is correct, but solution did not word it that way.
e) Missed this on both runs
f) Missed this on both runs

Question 3

a) First Try - solved the question
b) First Try - solved the question
c) The derivation might be different/incorrect but it does get to the right answer, so I will give it credit.

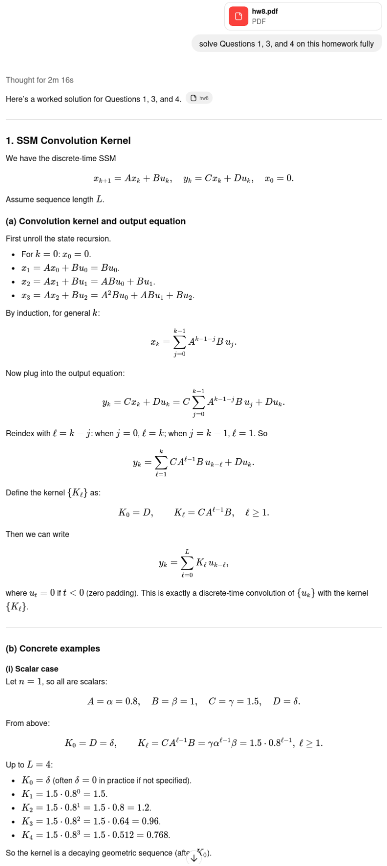
Question 4

a) First Try - solved the question correctly
b) First Try - solved the question correctly
c) First Try - solved the question correctly
d) First Try - solved the question correctly
e) First Try - solved the question correctly
f) GPT says that the answer matches in substance to the solutions when re-prompted with the solutions

g) GPT says that the answer matches in substance to the solutions when re-prompted with the solutions

Thoughts: This performed much better than the discussion I had heard previously on GPT's ability to solve problems.

Log:



solve Questions 1, 3, and 4 on this homework fully

Thought for 2m 16s

Here's a worked solution for Questions 1, 3, and 4.

---

1. SSM Convolution Kernel

We have the discrete-time SSM

$$x_{k+1} = Ax_k + Bu_k, \quad y_k = Cx_k + Du_k, \quad x_0 = 0.$$

Assume sequence length L.

(a) Convolution kernel and output equation

First unroll the state recursion.

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

By induction, for general k:

$$x_k = \sum_{j=0}^{k-1} A^{k-1-j} B\, u_j.$$

Now plug into the output equation:

$$y_k = Cx_k + Du_k = C \sum_{j=0}^{k-1} A^{k-1-j} B\, u_j + Du_k.$$

Reindex with $\ell = k - j$: when $j = 0$, $\ell = k$; when $j = k - 1$, $\ell = 1$. So

$$y_k = \sum_{\ell=1}^{k} CA^{\ell-1} B\, u_{k-\ell} + Du_k.$$

Define the kernel $\{K_\ell\}$ as:

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

Then we can write

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

where $u_t = 0$ if $t < 0$ (zero padding). This is exactly a discrete-time convolution of $\{u_k\}$ with the kernel $\{K_\ell\}$.

(b) Concrete examples

(i) Scalar case

Let $n = 1$, so all are scalars:

$$A = \alpha = 0.8, \quad B = \beta = 1, \quad C = \gamma = 1.5, \quad D = \delta.$$

From above:

$$K_0 = D = \delta, \qquad K_\ell = CA^{\ell-1}B = \gamma\alpha^{\ell-1}\beta = 1.5 \cdot 0.8^{\ell-1},\ \ell \geq 1.$$

Up to $L = 4$:

- $K_0 = \delta$ (often $\delta = 0$ in practice if not specified).
- $K_1 = 1.5 \cdot 0.8^0 = 1.5$.
- $K_2 = 1.5 \cdot 0.8^1 = 1.5 \cdot 0.8 = 1.2$.
- $K_3 = 1.5 \cdot 0.8^2 = 1.5 \cdot 0.64 = 0.96$.
- $K_4 = 1.5 \cdot 0.8^3 = 1.5 \cdot 0.512 = 0.768$.

So the kernel is a decaying geometric sequence (afte K_0).

So the kernel is a decaying geometric sequence (after K_0).

(ii) 2D case
Given

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

Kernel formula:

$$K_0 = D = 0, \qquad K_\ell = CA^{\ell-1}B,\ \ell \geq 1.$$

We just need the first columns of $A^{\ell-1}$:

- $\ell = 1$: $A^0 = I$, so

$$K_1 = CIB = CB = [1\ 0] \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1.$$

- $\ell = 2$: $K_2 = CAB$. First compute

$$AB = \text{first column of } A = \begin{bmatrix} 0.7 \\ 0.2 \end{bmatrix} \Rightarrow K_2 = [1\ 0] \begin{bmatrix} 0.7 \\ 0.2 \end{bmatrix} = 0.7.$$

- $\ell = 3$: $K_3 = CA^2B$. First

$$A^2 = \begin{bmatrix} 0.51 & 0.13 \\ 0.26 & 0.38 \end{bmatrix} \Rightarrow A^2B = \text{first column of } A^2 = \begin{bmatrix} 0.51 \\ 0.26 \end{bmatrix},$$

  so $K_3 = [1\ 0] \begin{bmatrix} 0.51 \\ 0.26 \end{bmatrix} = 0.51.$

Thus up to $L = 3$:

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

Impulse-response interpretation.
If the input is an impulse (e.g., $u_0 = 1$, all others 0), then $y_0 = K_0$, $y_1 = K_1$, etc. The kernel $\{K_\ell\}$ is exactly the system's impulse response: how the output decays over time after a single spike at the input. Here we see a decaying sequence $1, 0.7, 0.51, \ldots$, reflecting how the 2D state contracts under A while being read out by C.

(c) Efficient computation with convolutions

Assume we already know the kernel $K_0, \ldots, K_L$, and the input sequence $u_0, \ldots, u_L \in \mathbb{R}^d$. The output is

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

- **Parallelization with convolution:**
  For each fixed k, each product $K_\ell u_{k-\ell}$ is independent, so all multiplications can be done in parallel. Then we just need to sum up $L + 1$ vectors of dimension equal to $\dim(y_k)$.

  Using a parallel tree reduction, summing $L + 1$ terms has critical path length $O(\log L)$. So the **minimum critical path length** (depth) for computing the whole output sequence (assuming enough processors) is $O(\log L)$.

- **Naive direct recursion:**
  If we instead compute x_k from the recursion $x_{k+1} = Ax_k + Bu_k$ and then $y_k = Cx_k + Du_k$, each state x_{k+1} depends on x_k. That chain has length L, so the critical path length is $O(L)$ — fundamentally sequential in time.

So using the convolution formulation massively increases parallelizability: $O(\log L)$ vs $O(L)$.

(d) Efficient kernel computation

We want $K_0 = D$ and $K_\ell = CA^{\ell-1}B$ for $\ell = 1, \ldots, L$.

Key ideas for efficiency and parallelism:

1. **Binary exponentiation of A:**
   Precompute

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

   where $L = 2^N$. This takes $O(\log L)$ matrix multiplications of cost $O(n^3)$ each.
2. **Divide-and-conquer on the sequence of states:**

   Consider the impulse input $u_0 = 1$, others 0. Then the sequence of states is

1. **Binary exponentiation of A:**
   Precompute
   $$A, A^2, A^4, \dots, A^{2^{N-1}},$$
   where $L = 2^N$. This takes $O(\log L)$ matrix multiplications of cost $O(n^3)$ each.
2. **Divide-and-conquer on the sequence of states:**

   Consider the impulse input $u_0 = 1$, others 0. Then the sequence of states is
   $$x_1 = B,\ x_2 = AB,\ x_3 = A^2B, \dots,\ x_L = A^{L-1}B.$$
   We can compute all x_k via a divide-and-conquer recursion (like a parallel prefix with matrix A):
   - Compute all states for the first half of the time window.
   - Use $A^{L/2}$ to “jump” those states to the second half.

   This yields an $O(L)$ total work algorithm with $O(\log L)$ **depth** instead of L sequential steps.
3. **FFT-based convolution (if needed):**
   If you only need the action of the kernel as a convolution on many sequences, you can compute the kernel and then perform convolutions using FFT in $O(L \log L)$ time.

(e) A diagonal A

Suppose A is diagonal:
$$A = \operatorname{diag}(a_1, \dots, a_n).$$
Then
$$A^{\ell-1} = \operatorname{diag}(a_1^{\ell-1}, \dots, a_n^{\ell-1}),$$
so each coordinate is just a geometric sequence.

If B is a vector and C is a row vector,
$$K_\ell = CA^{\ell-1}B = \sum_{i=1}^{n} C_i B_i a_i^{\ell-1}.$$
So:

- Each scalar sequence $a_i^{\ell-1}$ can be computed very cheaply (few multiplies per time step).
- You can precompute $C_i B_i$ and then form K_ℓ as a sum of n scalar geometric sequences.
- This reduces kernel computation from generic matrix multiplications to scalar multiplications, with $O(nL)$ work and easy parallelization over coordinates.

(f) DPLR structure: $A = I_n + pp^\top$

Now A has the form
$$A = I_n + pp^\top, \qquad p \in \mathbb{R}^n.$$
Multiplying any vector x by A is:
$$Ax = x + p(p^\top x).$$
This costs only:

- One inner product $p^\top x$: $O(n)$,
- One scaled vector add $p(p^\top x)$: $O(n)$.

So each multiplication by A is $O(n)$ instead of $O(n^2)$.

For kernel computation:

- Recall $K_\ell = CA^{\ell-1}B$.
- We can compute the sequence $v_\ell = A^{\ell-1}B$ by iterating
  $$v_1 = B, \quad v_{\ell+1} = Av_\ell = v_\ell + p(p^\top v_\ell),$$
  with cost $O(n)$ per step.
- Then $K_\ell = Cv_\ell$ is just an inner product for each ℓ.

Total work is $O(nL)$ instead of repeatedly doing $O(n^3 \log L)$ matrix exponentiation. The low-rank structure makes each application of A cheap and thus the kernel sequence cheap.

---

3. Self-Supervised Linear Purification

We have

3. Self-Supervised Linear Purification

We have

$$L_1(W;X) = \|X - WX\|_F^2, \quad L_2(W;X,\lambda) = \|X - WX\|_F^2 + \lambda\|W\|_F^2.$$

(a) Small 2D example

Given

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

and

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

(i) Reconstruction & regularization losses

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

- $W^{(\alpha)}X = X$, so

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

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

So:

- Encoder α:
  Reconstruction loss = 0
  Regularization loss = 2.

2. **For $W^{(\beta)}$:**

- This zeroes the second row, so

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

  Then

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

  The reconstruction loss is

$$0.02^2 + (-0.01)^2 + 0.01^2 + (-0.02)^2 = 0.0004 + 0.0001 + 0.0001 + 0.0004 = 0.001.$$

- Frobenius norm:
  $\|W^{(\beta)}\|_F^2 = 1^2 + 0^2 + 0^2 + 0^2 = 1.$

So the table is:

Encoder	Reconstruction Loss	Regularization Loss
α	0	2
β	0.001	1

(ii) Values of λ where $W^{(\alpha)}$ has higher L_2

Compute:

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

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

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

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

- For $\lambda > 0.001$, the identity has *higher* regularized loss than $W^{(\beta)}$.
- At $\lambda = 0.001$, they are equal.
- For $\lambda < 0.001$, the identity is preferred.

(b) General solution in SVD coordinates

We know $X \in \mathbb{R}^{m \times n}$ with $n > m$, SVD:

$$X = U\Sigma V^\top,$$

where Σ has singular values $\sigma_1 > \cdots > \sigma_m \geq 0$.

We want to minimize

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

(i) Fill in the expression for W

We want to minimize

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

(i) Fill in the expression for W_c

The optimizing matrix has the form

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

So the missing matrices are U (left and right).

(ii) Derivation

Use invariance of Frobenius norm under orthogonal transforms.

Let

$$\widetilde{W} = U^\top W U.$$

Then:

1. **Transform the reconstruction term:**

$$X - WX = U\Sigma V^\top - WU\Sigma V^\top.$$

Pre-multiply by $U^\top$ and post-multiply by V (both orthogonal and norm-preserving):

$$\|X - WX\|_F^2 = \|\Sigma - U^\top W U\, \Sigma\|_F^2 = \|\Sigma - \widetilde{W}\Sigma\|_F^2.$$

2. **Transform the regularization term:**

$$\|W\|_F^2 = \|U^\top W U\|_F^2 = \|\widetilde{W}\|_F^2.$$

So the objective becomes

$$L_2(W) = \|\Sigma - \widetilde{W}\Sigma\|_F^2 + \lambda \|\widetilde{W}\|_F^2.$$

Write Σ explicitly as diag$(\sigma_1, \ldots, \sigma_m)$ (with possibly extra zero columns if $n > m$), and denote $\widetilde{W} = (\widetilde{w}_{ij})$.

Then

$$(\Sigma - \widetilde{W}\Sigma)_{ij} = \begin{cases} \sigma_i(1 - \widetilde{w}_{ii}), & i = j, \\ -\widetilde{w}_{ij}\sigma_j, & i \neq j. \end{cases}$$

Therefore

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

Add the regularization:

$$L_2 = \sum_{i=1}^m \sigma_i^2 (1 - \widetilde{w}_{ii})^2 + \sum_{i \neq j} (\sigma_j^2 + \lambda) \widetilde{w}_{ij}^2 + \sum_{i=1}^m \lambda \widetilde{w}_{ii}^2.$$

This **decouples** over all entries $\widetilde{w}_{ij}$:

- For off-diagonal $i \neq j$, the coefficient is $\sigma_j^2 + \lambda > 0$. Minimizer is $\widetilde{w}_{ij}^\star = 0$.
- For each diagonal entry $\widetilde{w}_{ii}$, the problem reduces to 1D:

$$f_i(a) = \sigma_i^2 (1 - a)^2 + \lambda a^2,$$

where $a = \widetilde{w}_{ii}$.

Differentiate and set to zero:

$$f_i'(a) = 2\sigma_i^2 (a - 1) + 2\lambda a = 0 \Rightarrow a(\sigma_i^2 + \lambda) = \sigma_i^2 \Rightarrow a = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}.$$

So $\widetilde{W}$ is diagonal:

$$\widetilde{W} = \mathrm{diag}\left(\frac{\sigma_1^2}{\sigma_1^2 + \lambda}, \ldots, \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \right).$$

Transform back:

$$W_c = U\widetilde{W}U^\top = U \,\mathrm{diag}\Big(\frac{\sigma_i^2}{\sigma_i^2 + \lambda}\Big) U^\top.$$

(c) Choosing λ for a specific spectrum

(c) Choosing λ for a specific spectrum

Given σ_i for $X \in \mathbb{R}^{8\times n}$:

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

The shrinkage factor along singular direction i is

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

We want:

1. For the first 3 singular directions, preserve at least 80%:
$$f_i(\lambda) \geq 0.8, \quad i = 1, 2, 3.$$
2. For directions 4–8, attenuate to at most 50%:
$$f_i(\lambda) \leq 0.5, \quad i = 4, \dots, 8.$$

First condition:

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8 \Rightarrow \lambda \leq 0.25\,\sigma_i^2$$

(as derived: $\lambda \leq \frac{1}{4}\sigma_i^2$).

Compute σ_i^2 for $i = 1, 2, 3$:

- $\sigma_1^2 = 100 \Rightarrow \lambda \leq 25.$
- $\sigma_2^2 = 64 \Rightarrow \lambda \leq 16.$
- $\sigma_3^2 = 16 \Rightarrow \lambda \leq 4.$

So to satisfy **all three**, we need

$$\lambda \leq 4.$$

Second condition:

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

Compute σ_i^2 for $i = 4, \dots, 8$:

- $\sigma_4 = 1 \Rightarrow \sigma_4^2 = 1.$
- $\sigma_5 = 0.5 \Rightarrow 0.25.$
- $\sigma_6 = 0.36 \Rightarrow 0.1296.$
- $\sigma_7 = 0.16 \Rightarrow 0.0256.$
- $\sigma_8 = 0.01 \Rightarrow 0.0001.$

We need $\lambda \geq \max_{i\geq 4} \sigma_i^2 = 1.$

So both sets of constraints are satisfied if

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

Any λ in $[1, 4]$ gives at least 80% preservation along the top 3 singular directions and at most 50% strength along the remaining ones.

---

4. Ridge-Attention

(a) Incremental update of an average

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

We want

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

Compute:

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

So

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

This shows we can update the average using old average m, new point x_{n+1}, and n.

So

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

This shows we can update the average using old average m, new point x_{n+1}, and n.

(b) Ridge regression in sum form

We have matrix $A \in \mathbb{R}^{n\times d}$ whose rows are $x_i^\top$, and $y \in \mathbb{R}^n$ with entries y_i. The ridge solution is

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

We need to write the pieces as sums.

- First:

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

  So:

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

- Second:

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

So the blanks are:

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

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

(c) Cost of non-causal ridge self-attention

Non-causal ridge self-attention:

- Context length n.
- Keys $x_i \in \mathbb{R}^d$.
- Values $v_i \in \mathbb{R}^d$ (say same dimension).
- All n queries use the same set of (key,value) pairs.

We effectively:

1. Build

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

   — cost $O(nd^2)$.
2. Build

$$B = A^\top Y$$

   where Y stacks the value vectors — also $O(nd^2)$.
3. Invert the $d \times d$ matrix S: cost $O(d^3)$.
4. Multiply $S^{-1}B$: cost $O(d^3)$.
5. Apply the resulting linear map to each of the n query vectors: each application is a matrix–vector multiply $O(d^2)$, so total $O(nd^2)$.

Given $d < n$, the term nd^2 dominates d^3 asymptotically, so the overall complexity is:

$$\boxed{O(nd^2)}.$$

So the correct option is $O(nd^2)$.

(d) Backprop through ridge self-attention

We use ridge self-attention inside a Transformer, with a downstream loss.

The computation of the ridge solution is:

$$w^\star(\lambda, \text{keys}, \text{values}) = (A^\top A + \lambda I)^{-1} A^\top y.$$

The computation of the ridge solution is:

$$w^\star(\lambda, \mathrm{keys}, \mathrm{values}) = (A^\top A + \lambda I)^{-1} A^\top y,$$

and then the attention outputs depend on $w^\star$ and the query vectors.

This is a composition of differentiable operations (matrix multiplications, inverse, etc.). Thus backprop can propagate gradients to:

- The **ridge** λ (if treated as a learnable parameter).
- The **keys** (they appear inside A).
- The **values** (they appear inside y or Y).
- The **queries** (used when evaluating the prediction on test inputs).

So **all four** items listed will receive gradients and can be learned.

(e) Cost of one Sherman–Morrison update

Sherman–Morrison:

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

Assuming we already know M^{-1}, and:

- Matrix–vector multiply costs $O(d^2)$,
- Dyad $uv^\top$ costs $O(d^2)$,
- Inner product costs $O(d)$.

Steps:

1. $z = M^{-1}u$: $O(d^2)$.
2. $w^\top = v^\top M^{-1}$ (or $M^{-1\top} v$): $O(d^2)$.
3. Scalar $\alpha = 1 + v^\top z$: $O(d)$.
4. Dyad $zw^\top$: $O(d^2)$.
5. Scale and subtract from M^{-1}: $O(d^2)$.

Dominated by $O(d^2)$, so the correct choice is:

$$\boxed{O(d^2)}.$$

(f) Efficient causal ridge self-attention

We want **causal** ridge self-attention with context length n:

- At time t, only keys/values from positions $1, \ldots, t$ are available and used.
- We want to avoid recomputing ridge regression from scratch at each t (which would be $O(n^2 d^2)$).

Let $k_t \in \mathbb{R}^d$ be the key, $v_t \in \mathbb{R}^d$ be the value at time t.

Define:

- $S_t = \lambda I + \sum_{i=1}^{t} k_i k_i^\top \in \mathbb{R}^{d \times d}$.
- $P_t = S_t^{-1}$ (the inverse).
- $B_t = \sum_{i=1}^{t} k_i v_i^\top \in \mathbb{R}^{d \times d}$, so the ridge weight matrix at time t is $W_t = P_t B_t$ (up to orientation).

Initialization:

- $S_0 = \lambda I, \quad P_0 = \lambda^{-1} I, \quad B_0 = 0.$

Recursive update at each time step t:

1. **Update covariance:**

   $$S_t = S_{t-1} + k_t k_t^\top.$$

2. **Update inverse using Sherman–Morrison** (rank-1 update with $u = v = k_t$):

   $$P_t = S_t^{-1} = P_{t-1} - \frac{P_{t-1} k_t k_t^\top P_{t-1}}{1 + k_t^\top P_{t-1} k_t}.$$

   This is $O(d^2)$.

3. **Update cross-term B_t:**

   $$B_t = B_{t-1} + k_t v_t^\top.$$

   Again $O(d^2)$ if values are d-dimensional.

4. **Compute output for query q_t:**

   The multi-output ridge solution at time t is:

   $$W_t = P_t B_t.$$

   The prediction for query q_t is

$$B_t = B_{t-1} + k_t v_t^\top.$$

Again $O(d^2)$ if values are d-dimensional.

4. **Compute output for query q_t:**

   The multi-output ridge solution at time t is:

   $$W_t = P_t B_t.$$

   The prediction for query q_t is

   $$y_t = W_t^\top q_t \quad (\text{or } y_t = q_t^\top W_t, \text{depending on conventions}).$$

   Computing y_t costs $O(d^2)$.

Thus per timestep cost is $O(d^2)$, and for a context of length n the total is $O(nd^2)$, which is much better than recomputing ridge from scratch each time in $O(n^2 d^2)$.

The key idea: maintain S_t^{-1} and B_t recursively and use Sherman–Morrison for the rank-1 update $k_t k_t^\top$.

(g) Attention weights in ridge-attention

For traditional softmax attention, the output at a query q is

$$\sum_{i=1}^{n} \alpha_i(q) v_i$$

with $\alpha_i(q)$ given directly by the softmax over key–query scores.

For ridge-attention, the ridge regression prediction with keys K (rows = $k_i^\top$), values V, and query q is:

1. Compute

   $$S = K^\top K + \lambda I, \quad P = S^{-1}.$$

2. Multi-output ridge weight matrix:

   $$W^\star = PK^\top V.$$

3. Prediction at query q:

   $$\hat{y}(q) = q^\top W^\star = q^\top PK^\top V.$$

We want to express this as a combination of values:

$$\hat{y}(q) = \sum_{i=1}^{n} w_i(q) v_i.$$

Let the vector of **ridge-attention weights** be

$$w(q)^\top = q^\top PK^\top \in \mathbb{R}^{1\times n}.$$

Then:

$$\hat{y}(q) = w(q)^\top V = \sum_{i=1}^{n} w_i(q) v_i.$$

Equivalently, if we prefer column form,

$$w(q) = KPq \in \mathbb{R}^n,$$

and $\hat{y}(q) = V^\top w(q)$.

So to **visualize ridge-attention weights** for a given query:

1. Compute $P = (K^\top K + \lambda I)^{-1}$.
2. Compute $z = Pq$.
3. Compute $w(q) = Kz \in \mathbb{R}^n$.
4. The i-th entry $w_i(q)$ is the (possibly negative, not necessarily summing to 1) weight on value v_i.

These $w_i(q)$ play the role analogous to the softmax attention weights.

Sources

okay for question 1c-onward, you did not consider the matrix multiplication time, can you factor that in and give me new answers for 1c, 1d, 1e, 1f

Thought for 3m 9s

Ask anything

Thinking

okay for question 1c-onward, you did not consider the matrix multiplication time, can you factor that in and give me new answers for 1c, 1d, 1e, 1f

Thought for 3m 9s

Good catch — you're right that I'd implicitly treated matrix multiplies as "unit cost." Let's redo **1(c)–1(f)** and explicitly include the cost of matrix/matrix–vector multiplications.

I'll use:

- n = state dimension (so $A \in \mathbb{R}^{n\times n}$)
- d = input dimension ($u_k \in \mathbb{R}^d$)
- p = output dimension ($y_k \in \mathbb{R}^p$)
- Dense costs (standard):
  - matvec $M \in \mathbb{R}^{a\times b}, x \in \mathbb{R}^b$: $\Theta(ab)$
  - matmat $M_1 \in \mathbb{R}^{a\times b}, M_2 \in \mathbb{R}^{b\times c}$: $\Theta(abc)$

1(c) Efficient computation with convolutions (with matrix cost)

We assume the kernel $K = \{K_\ell\}_{\ell=0}^{L}$ is already known, with
$K_\ell \in \mathbb{R}^{p\times d}$, and we have input $u_k \in \mathbb{R}^d$ of length $L+1$.
Output:

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

Work and depth with convolutions

For each output time step k:

- We must compute up to $L+1$ **matvecs** $K_\ell u_{k-\ell}$, each costing $\Theta(pd)$.
- Then sum the resulting $L+1$ vectors in $\mathbb{R}^p$.

Total work (if you literally do the sum as written):

- Number of matvecs: $\sim (L+1)^2$ across all $k \in \{0, \dots, L\}$.
- Work: $\Theta\big(L^2 \cdot pd\big)$.

In practice you'd implement this as a convolution (FFT or highly optimized conv operator) with work closer to $\Theta(L\,pd\log L)$, but the question is about parallelism / critical path, so:

Critical path length (with matmul cost):

- Suppose one matvec $K_\ell u_j$ has latency $T_{\text{mv}} = \Theta(pd)$ if done "straight-line."
  With enough cores:
  - All $K_\ell u_{k-\ell}$ across all (k, ℓ) can be done *in parallel* → contributes ~T_{mv} to depth.
  - For a fixed k, summing $L+1$ p-dimensional vectors via a tree reduction costs $\Theta(\log L)$ additions per component, so depth $\Theta(\log L)$.

So the **minimum critical path length** for the convolution implementation is roughly

$$\boxed{\text{depth}_{\text{conv}} \ \sim\ T_{\text{mv}} + \Theta(\log L) \ =\ \Theta(pd) + \Theta(\log L)}.$$

If you further parallelize within the matvec (over p and d), the T_{mv} term itself shrinks toward $\Theta(\log p + \log d)$, so you get something like $\Theta(\log p + \log d + \log L)$.

Naive direct computation from the recursion

Unrolled recursion:

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

Per time step k:

- $Ax_k \in \mathbb{R}^n$: $\Theta(n^2)$ (dominant term).
- $Bu_k \in \mathbb{R}^n$: $\Theta(nd)$.
- $Cx_k \in \mathbb{R}^p$: $\Theta(pn)$.
- $Du_k \in \mathbb{R}^p$: $\Theta(pd)$.

Total per-step work: $\Theta(n^2 + nd + pn + pd)$.
If $d, p = O(n)$, this is $\Theta(n^2)$ per step, so

- **Total work**: $\Theta(Ln^2)$.
- **Critical path length**: each state depends on the previous, so you *must* do at least L sequential state

Total per-step work: $\Theta(n^2 + nd + pn + pd)$.
If $d, p = O(n)$, this is $\Theta(n^2)$ per step, so

- **Total work**: $\Theta(Ln^2)$.
- **Critical path length**: each state depends on the previous, so you *must* do at least L sequential state updates. Even if each matvec is fully parallelized internally, you still have

  $$\boxed{\text{depth}_{\text{recur}} \ \sim \ L \cdot T_{\text{step}}}$$

  where T_{step} is the latency of one state update (essentially one $n \times n$ matvec). If you parallelize inside the matvec, this is $\Theta(L \log n)$.

Summary of (c):

- Recurrent: work $\Theta(Ln^2)$, depth $\Theta(L \cdot T_{\text{step}})$.
- Convolution (with known kernel): work roughly $\Theta(L^2pd)$ in the naive implementation, but depth only $\Theta(T_{\text{mv}} + \log L)$.
  So with enough parallel compute, the convolutional implementation has dramatically shorter critical path (factor $\approx L/\log L$) at the cost of more total arithmetic.

1(d) Efficient kernel computation (with matrix cost)

We need to compute

$$K_0 = D, \qquad K_\ell = CA^{\ell-1}B, \quad \ell = 1, \dots, L.$$

Here:

- $A \in \mathbb{R}^{n \times n}$
- $B \in \mathbb{R}^{n \times d}$
- $C \in \mathbb{R}^{p \times n}$.

Naive sequential approach

One simple unrolled approach:

- Let $M_1 = B$, and define

  $$M_{\ell+1} = AM_\ell \quad (\ell = 1, \dots, L-1).$$

  Then $M_\ell = A^{\ell-1}B$.
- Then $K_\ell = CM_\ell$.

Costs:

- Each $M_{\ell+1} = AM_\ell$: matmat $n \times n$ by $n \times d$: $\Theta(n^2d)$.
- Do this for $\ell = 1, \dots, L-1$ → work $\Theta(Ln^2d)$.
- Each $K_\ell = CM_\ell$: matmat $p \times n$ by $n \times d$: $\Theta(pnd)$, so total $\Theta(Lpnd)$ (usually not the bottleneck if p small/≈d).

So naive **work**: $\Theta(Ln^2d)$.
Depth (sequential): $\Theta(L)$ matmats, each of latency $T_{\text{mm}} = \Theta(n^2d)$ (or $\Theta(n^3)$ if you treat it as $n \times n$ × $n \times n$), so:

$$\boxed{\text{depth}_{\text{naive}} \sim L \cdot T_{\text{mm}}}.$$

Parallel / divide-and-conquer strategy (for $L = 2^N$)

Goal: reduce **depth** while keeping similar or slightly larger total work.

Two key ideas:

1. **Matrix-power precomputation via repeated squaring**
   Compute:

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

   in $\Theta(N) = \Theta(\log L)$ dense $n \times n$ matmats.
   Work: $\Theta(n^3 \log L)$.
   Depth: $\Theta(\log L) \cdot T_{n \times n}$.
2. **Divide-and-conquer over time with parallelism**
   You can structure the computation of all $M_\ell = A^{\ell-1}B$ as a segment tree or parallel prefix:
   - At the top level, split the time indices $\{1, \dots, L\}$ into first half and second half.
   - Use the precomputed power $A^{L/2}$ to "jump" from the end of the first half to the start of the second half.
   - Recurse within each half; at each recursion you multiply by appropriate power of A.

This gives:

- **Work** still on the order of $\Theta(Ln^2d)$ matmats A plus $\Theta(n^3 \log L)$ for power precomputation.
- **Depth**: each level of recursion uses one additional multiplication by a precomputed power of A, so

Ask anything
Thinking

This gives:

- **Work** still on the order of $\Theta(Ln^2 d)$ matmats AM plus $\Theta(n^3 \log L)$ for power precomputation.
- **Depth**: each level of recursion uses one additional multiplication by a precomputed power of A, so

$$\boxed{\mathrm{depth}_{\mathrm{kernel}} \sim \log L \cdot T_{\mathrm{mm}}}$$

  vs $\sim L \cdot T_{\mathrm{mm}}$ for the naive sequential power chain.

So factoring matrix-multiplication time:

- Naive kernel generation: $\Theta(Ln^2 d)$ work, depth $\Theta(LT_{\mathrm{mm}})$.
- Divide-and-conquer / prefix-style: similar work (plus $\Theta(n^3 \log L)$ overhead), but depth only $\Theta(\log L \cdot T_{\mathrm{mm}})$.

1(e) Diagonal A (with matrix cost)

Now suppose

$$A = \mathrm{diag}(a_1, \dots, a_n).$$

Then:

- $A^{\ell-1} = \mathrm{diag}(a_1^{\ell-1}, \dots, a_n^{\ell-1})$; powering is **elementwise**, no heavy matrix-matrix operations.

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

1. Compute $A^{\ell-1}B$. Because $A^{\ell-1}$ is diagonal, this is just scaling each **row** of B by a scalar:
   - Cost per ℓ: $\Theta(nd)$ (rowwise scaling).
   - For all ℓ: $\Theta(Lnd)$.
2. Then multiply by C: for each ℓ, $K_\ell = C(A^{\ell-1}B)$ is a matmat $p \times n$ by $n \times d$: $\Theta(pnd)$, so $\Theta(Lpnd)$ overall.

You can improve further:

- Precompute the "base" rank-1 contributions:

$$M_r := C_{:,r} B_{r,:} \in \mathbb{R}^{p \times d}, \quad r = 1, \dots, n,$$

  so:

$$K_\ell = \sum_{r=1}^{n} a_r^{\ell-1} M_r.$$

- Then:
  - Precompute M_r once: $\Theta(pnd)$.
  - Maintain the sequences $a_r^{\ell-1}$ by recurrence $a_r^\ell = a_r^{\ell-1} a_r$: $\Theta(Ln)$.
  - For each ℓ, form K_ℓ via a weighted sum of M_r: $\Theta(npd)$ if done straightforwardly.

The main point: **multiplying by A is now cheap** — we never do an $n \times n$ dense matmat, only scaling operations.

Compared to dense A:

- Dense: each AM is $\Theta(n^2 d)$.
- Diagonal: each AM is $\Theta(nd)$.

So:

$$\boxed{\text{Kernel work with diagonal } A \sim \Theta(Lnd + Lpnd) \quad \text{vs} \quad \Theta(Ln^2 d) \text{ for dense } A.}$$

Depth-wise, you can still apply the same divide-and-conquer tricks as in (d), but the per-level cost is now $\Theta(nd)$ instead of $\Theta(n^2 d)$.

1(f) DPLR: $A = I_n + pp^\top$ (with matrix cost)

Now suppose

$$A = I_n + pp^\top, \quad p \in \mathbb{R}^n.$$

For any vector $x \in \mathbb{R}^n$,

$$Ax = x + p(p^\top x).$$

This can be computed as:

1. $s = p^\top x$: inner product, cost $\Theta(n)$.
2. ps: scale p by scalar s, cost $\Theta(n)$.
3. Add to x: cost $\Theta(n)$.

So one **vector** multiplication by A costs $\Theta(n)$ instead of $\Theta(n^2)$.

For a **matrix** $M \in \mathbb{R}^{n \times d}$ (e.g., multiple columns or ↓ iple starting vectors):

Ask anything

+ Thinking

ChatGPT can make mistakes. Check important info.

So one **vector** multiplication by A costs $\Theta(n)$ instead of $\Theta(n^2)$.

For a **matrix** $M \in \mathbb{R}^{n\times d}$ (e.g., multiple columns or multiple starting vectors):

- Treat each column as a vector:
  - For each column m_j, $Am_j = m_j + p(p^\top m_j)$.
  - Inner product per column: $\Theta(n)$.
  - Scaling and addition: $\Theta(n)$.
- Total cost: $\Theta(nd)$ to compute AM, vs $\Theta(n^2 d)$ for dense A.

So for kernel generation:

- As in the naive scheme from (d), you can still compute

$$M_1 = B, \quad M_{\ell+1} = AM_\ell, \quad K_\ell = CM_\ell.$$

- But now each step $M_{\ell+1} = AM_\ell$ is $\Theta(nd)$, not $\Theta(n^2 d)$.

Hence:

$$\boxed{\text{Work with DPLR } A \sim \Theta(Lnd + Lpnd) \quad \text{vs} \quad \Theta(Ln^2 d) \text{ for dense } A.}$$

You can *also* apply the divide-and-conquer / prefix tricks:

- Use AM as the "combine" operator in a parallel scan over time.
- Each "combine" call costs $\Theta(nd)$.
- Total work still $\Theta(Lnd)$ but depth becomes $\Theta(\log L \cdot nd)$ if you treat each combine as a unit, or $\Theta(\log L \cdot (\log n + \log d))$ if you parallelize inside the combines as well.

So the low-rank structure mainly **reduces the per-step cost** from $\Theta(n^2 d)$ to $\Theta(nd)$; combining that with the same parallelization strategies as in (d) gives you both lower work and good depth.

If you want, we can plug in some concrete n, d, p, L (e.g., typical SSM numbers) and compare actual asymptotic regimes where recurrent vs convolution vs structured A changes which implementation is preferable.

hw08_solution.pdf
PDF

can you look at this solution and let me know if your answer for 4f and 4g matches

Thought for 23s

Short answer: yes, they match in substance. Let me line them up so you can see the correspondence.
hw08_solution

4(f): Causal ridge self-attention

Official solution notation:

- Keys/values up to time t:
  A_t = stacked keys, B_t = stacked values
- Define
  $M_t = \lambda I + A_t^\top A_t = \lambda I + \sum_{i=1}^t k_i k_i^\top$
  $J_t = A_t^\top B_t = \sum_{i=1}^t k_i v_i^\top$
- Ridge weights at time t:
  $W^*(t) = M_t^{-1} J_t$
- They then use **Sherman–Morrison** to update M_t^{-1} from M_{t-1}^{-1} in $O(d^2)$, and update J_t from J_{t-1} via a rank-1 update $J_t = J_{t-1} + k_t v_t^\top$.
- Output for query q_t: either multiply $W^*(t)$ by q_t, or use the "minimal" variant
  $y_t = W^*(t)^\top q_t = J_t^\top (M_t^{-1})^\top q_t$, all in $O(d^2)$ per time step.

What I wrote:

- I used almost the same objects, just different letters:
  - $S_t = \lambda I + \sum_{i=1}^t k_i k_i^\top$ (this is their M_t)
  - $P_t = S_t^{-1}$ (their M_t^{-1})
  - $B_t = \sum_{i=1}^t k_i v_i^\top$ (their J_t)
- Recursion:
  - Update P_t from P_{t-1} using Sherman–Morrison on the rank-1 update $k_t k_t^\top \to O(d^2)$.
  - Update $B_t = B_{t-1} + k_t v_t^\top$.
  - Weight matrix $W_t = P_t B_t$, and output $y_t = W_t^\top q_t$; total per-step $O(d^2)$.

Ask anything

Thinking

- Recursion:
  - Update P_t from P_{t-1} using Sherman–Morrison on the rank-1 update $k_t k_t^\top \to O(d^2)$.
  - Update $B_t = B_{t-1} + k_t v_t^\top$.
  - Weight matrix $W_t = P_t B_t$, and output $y_t = W_t^\top q_t$; total per-step $O(d^2)$.

So conceptually it's *exactly* the same algorithm:

- Track the inverse covariance M_t^{-1} and the "cross term" J_t,
- Update them in $O(d^2)$ per step via rank-1 updates,
- Compute each causal attention output in $O(d^2)$,
- Total cost $O(nd^2)$.

The official solution goes further and writes the recursion for $W^*(t)$ explicitly and then shows the "minimal" variant that only stores M_t^{-1} and J_t; I described the same two ideas but in a slightly more high-level way. There's no contradiction.

✅ **Verdict for 4(f):** Same method and same complexity, just different notation and detail level.

4(g): Attention weights for ridge-attention

Official solution:

They start from

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

so for query q,

$$W^{*\top} q = \left((\lambda I + A^\top A)^{-1} A^\top B\right)^\top q = B^\top \left(A(\lambda I + A^\top A)^{-1}\right) q.$$

They interpret:

- Let $w(q) := A(\lambda I + A^\top A)^{-1} q \in \mathbb{R}^n$.
- Since $B^\top = [v_1\ v_2 \dots v_n]$, the output is

  $$\sum_{i=1}^{n} w_i(q)\, v_i,$$

  so the **ridge-attention weights** are the entries of

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

They then give the kernel-ridge version:

$$\text{weights} = (\lambda I_n + AA^\top)^{-1} A q.$$

What I wrote:

- I denoted the keys matrix as $K \in \mathbb{R}^{n\times d}$ (their A).
- I defined

  $$P = (K^\top K + \lambda I)^{-1},$$

  and

  $$w(q) = KPq \in \mathbb{R}^n.$$

  Then

  $$\hat{y}(q) = \sum_{i=1}^{n} w_i(q)\, v_i.$$

If you substitute $K = A$, $P = (\lambda I + A^\top A)^{-1}$, you get exactly

$$w(q) = A(\lambda I + A^\top A)^{-1} q,$$

which is the same expression they identify as the attention weights.

I didn't explicitly write the kernel-ridge alternative $(\lambda I_n + AA^\top)^{-1} Aq$, but that's just the dual formulation of the same mapping.

✅ **Verdict for 4(g):** Yes, my answer is the same: the weights are the n-vector

$$w(q) = A(\lambda I + A^\top A)^{-1} q,$$

whose entries tell you how each value vector is weighted in the final output.

If you'd like, I can rewrite my 4(f) / 4(g) in exactly their notation (with A_t, B_t, M_t, J_t) so you can just drop them straight into your notes.

Sources

Ask anything

Thinking

4(g): Attention weights for ridge-attention

Official solution:

They start from

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

so for query q,

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

They interpret:

- Let $w(q) := A(\lambda I + A^\top A)^{-1} q \in \mathbb{R}^n$.
- Since $B^\top = [v_1\ v_2 \dots v_n]$, the output is

  $$\sum_{i=1}^{n} w_i(q)\, v_i,$$

  so the **ridge-attention weights** are the entries of

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

They then give the kernel-ridge version:

$$\text{weights} = (\lambda I_n + AA^\top)^{-1} Aq.$$

What I wrote:

- I denoted the keys matrix as $K \in \mathbb{R}^{n \times d}$ (their A).
- I defined

  $$P = (K^\top K + \lambda I)^{-1},$$

  and

  $$w(q) = KPq \in \mathbb{R}^n.$$

  Then

  $$\hat{y}(q) = \sum_{i=1}^{n} w_i(q)\, v_i.$$

If you substitute $K = A$, $P = (\lambda I + A^\top A)^{-1}$, you get exactly

$$w(q) = A(\lambda I + A^\top A)^{-1} q,$$

which is the same expression they identify as the attention weights.

I didn't explicitly write the kernel-ridge alternative $(\lambda I_n + AA^\top)^{-1} Aq$, but that's just the dual formulation of the same mapping.

✅ **Verdict for 4(g):** Yes, my answer is the same: the weights are the n-vector

$$w(q) = A(\lambda I + A^\top A)^{-1} q,$$

whose entries tell you how each value vector is weighted in the final output.

If you'd like, I can rewrite my 4(f) / 4(g) in exactly their notation (with A_t, B_t, M_t, J_t) so you can just drop them straight into your notes.

Sources

gpt can you give me a pdf for the log of this chat please

Thought for 17s

Here you go — I made a simple PDF transcript of this homework chat:

Download the chat log PDF

Ask anything

+ Thinking

ChatGPT can make mistakes. Check important info.