
EECS 182	Deep Neural Networks	Homework 8
Fall 2025	Anant Sahai and Gireeja Ranade	

This homework is due on October 31, at 10:59PM.

1. SSM Convolution Kernel

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

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

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

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

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

Solution: From the SSM recursion $x_{k+1} = Ax_k + Bu_k$, we get

$$x_1 = Bu_0, \quad x_2 = ABu_0 + Bu_1, \quad x_3 = A^2Bu_0 + ABu_1 + Bu_2, \dots$$

Hence,

$$y_0 = Du_0, \tag{1}$$

$$y_1 = Du_1 + CBu_0, \tag{2}$$

$$y_2 = Du_2 + CBu_1 + CABu_0, \tag{3}$$

$$y_3 = Du_3 + CBu_2 + CABu_1 + CA^2Bu_0, \tag{4}$$

$$\vdots \tag{5}$$

This summation matches a discrete convolution $\{y\} = K * \{u\}$ with kernel K given by

$$K = (D, CB, CAB, CA^2B, \dots).$$

(b) Concrete Examples.

- i. *Scalar Case:* Let $n = 1$, and set $A = \alpha$, $B = \beta$, $C = \gamma$, $D = \delta$. Use $\alpha = 0.8$, $\beta = 1$, $\gamma = 1.5$ and compute the kernel up to $L = 4$.

ii. *2D Case:* Let $A \in \mathbb{R}^{2 \times 2}$ be, for instance,

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

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

Solution:

(i) *Scalar example:* For $\alpha = 0.8, \beta = 1, \gamma = 1.5, \delta = 0$, the kernel reads

$$K = (0, 1.5, 1.5 \cdot 0.8, 1.5 \cdot 0.8^2, 1.5 \cdot 0.8^3).$$

(ii) *2D example:*

$$CB = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1,$$

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

$$CA^2B = 0.51,$$

The kernel of length $L = 4$ is $(0.0 \ 1.0 \ 0.7 \ 0.51)$.

Intuitively, the output of a SSM can be thought of as the response function of a sequence of discretized input signals and the output response decays over time. This mirrors the decaying kernel values because they correspond to how much the output value at the current step depends on the input values in the past (the older input values have lower influence on the current output value).

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

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

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

- (d) **Efficient Kernel Computation.** Given A, B, C , how can we compute the kernel, K , efficiently? What are some strategies to parallelize kernel computation? You may assume $L = 2^N$ for some N for simplicity.

Solution: To fully make use of parallel computational power of GPU, we want to compute $X^L = (I, A, \dots, A^{L-1})$ fast. We can apply divide-and-conquer idea on this:

$$X^L = (X^{L/2}, A^{L/2} X^{L/2})$$

$$A^L = A^{L/2} A^{L/2}$$

For each $L = 2^l$ with $l \in \{1, 2, \dots, N\}$, we can compute X^L through this recurrent formula so that we can compute the whole kernel with a maximum computation depth of $\mathcal{O}(\log n \log L)$ where $\mathcal{O}(\log n)$ is the maximum depth of computation for matrix multiplication.

- (e) **Adding structure to A .** Suppose A is a diagonal matrix. How can we leverage this structure to compute the kernel K more efficiently?

Solution: If A is diagonal, we no longer need to compute matrix multiplication, so we could pre-compute all (I, A, \dots, A^{L-1}) in $O(\log L)$ time (instead of $O(\log n \log L)$). In practice, the matrix multiplication is often much more expensive than the theoretical limit $O(\log n)$, so removing the need to do matrix multiplication for computing the kernel speeds up things a lot more.

- (f) **Diagonal-plus-low-rank (DPLR) structure** Now if A has the following form:

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

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

Solution:

$$A^L = (I_n + pp^\top)^L = I_n + Lpp^\top + \binom{L}{2}(pp^\top)^2 + \dots + (pp^\top)^L$$

We observe that $(pp^\top)^l = p(p^\top p)^{l-1}p^\top = pp^\top v^{l-1}$ where $v = \|p\|_2^2$. Simplifying the equation above yields

$$A^L = I_n + pp^\top \sum_{l=1}^L \left[\binom{L}{l} v^{l-1} \right]$$

It is easy to see that computing $\sum_{l=1}^L \left[\binom{L}{l} v^{l-1} \right]$ can be done in $\log(L)$ using divide-and-conquer with no matrix multiplication involved.

2. Coding SSM Forward

You showed that SSMs forward pass can be implemented using recurrent and convolution approaches in the previous problem. In this problem, we will implement the two approaches and compare their efficiency. Particularly, you are given the following update equation:

$$\mathbf{h}_{t+1} = (W \cdot \mathbf{h}_t + U \cdot \mathbf{x}_t + \mathbf{b})$$

You are given all the input \mathbf{x}_t for $t = 1, 2, \dots, T$ and the initial hidden state $\mathbf{h}_0 = 0$. You need to implement the forward pass for the SSM computing all the hidden states \mathbf{h}_t for $t = 1, 2, \dots, T$.

We will complete the notebooks [q_coding_ssm_forward_cpu.ipynb](#) and [q_coding_ssm_forward_gpu.ipynb](#) on the CPU and GPU respectively.

CPU Implementation:

- (a) Implement the forward pass by unrolling the SSM in time in a recurrent manner.

Solution: Check the notebook for the implementation.

- (b) Implement the forward pass by using a convolution-based approach. You will create the kernel K and use `nn.Conv1d` to apply the kernel to the input x . Hint: You can create the kernel using a divide-and-conquer method.

Solution: Check the notebook for the implementation.

- (c) Compare the runtime of the two approaches and explain your findings.

Solution: The total number of computations for the recurrent approach is $O(T \cdot H^2)$, while for the convolution based approach it is $O(H^3T)$. The convolution based approach is slower since it performs more computations than the recurrent approach and we cannot do parallelization on the CPU.

GPU Implementation:

- (d) Now implement the forward pass on the GPU. You can copy your existing implementation to the GPU notebook.

Solution: Check the notebook for the implementation.

- (e) Compare the runtime of the two approaches and explain your findings.

Solution: We can parallelize the convolution based approach on the GPU, making it faster than the recurrent approach for small H . However, as H increases, the recurrent approach starts to perform similarly or better due to the overhead of H^3 computations required when constructing the kernel.

- (f) To optimize the convolution-based approach, we will constrain W to be diagonal and leverage depthwise convolutions. Implement the new forward pass under this constraint and now compare the runtimes.

Solution: Check the notebook for the implementation. We observe that the depthwise convolution approach is faster than the recurrent approach for all H values.

3. Self-Supervised Linear Purification

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

$$\mathcal{L}_1(W; \mathbf{X}) = \|\mathbf{X} - W\mathbf{X}\|_F^2 \quad (6)$$

While the trivial solution $W = \mathbf{I}$ can minimize the reconstruction loss (6), we will now see how weight-decay (or equivalently in this case, ridge-style regularization) can help us achieve non-trivial purification.

$$\mathcal{L}_2(W; \mathbf{X}, \lambda) = \underbrace{\|\mathbf{X} - W\mathbf{X}\|_F^2}_{\text{Reconstruction Loss}} + \lambda \underbrace{\|W\|_F^2}_{\text{Regularization Loss}} \quad (7)$$

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

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

$$W^{(\alpha)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad W^{(\beta)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad (8)$$

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

$$\mathbf{X} = \begin{bmatrix} -2.17 & 1.98 & 2.41 & -2.03 \\ 0.02 & -0.01 & 0.01 & -0.02 \end{bmatrix} \quad (9)$$

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

Encoder	Reconstruction Loss	Regularization Loss
α	_____	_____
β	0.001	_____

Solution:

Encoder	Reconstruction Loss	Regularization Loss
α	0	2
β	0.001	1

- ii. For what values of the regularization parameter λ is the identity matrix $W^{(\alpha)}$ get higher loss \mathcal{L}_2 in (7), as compared to $W^{(\beta)}$?

Solution: Plugging the numbers from the table into Eqn. 7, the β encoder is a better choice for

$$\lambda > 1 \times 10^{-3}.$$

This suggests that despite the encoder being square, regularization induces a bottleneck in the encoder, preventing it from learning an identity mapping.

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

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

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

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

$$\widehat{W} = \left[\begin{array}{c} \quad \\ \quad \\ \quad \end{array} \right] \cdot \left[\begin{array}{ccc} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & \\ & \frac{\sigma_2^2}{\sigma_2^2 + \lambda} & \\ & & \ddots \\ & & & \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \end{array} \right] \cdot \left[\begin{array}{c} \quad \\ \quad \\ \quad \end{array} \right] \quad (10)$$

Solution:

$$\hat{W} = \left[\begin{array}{c} \quad \\ \mathbf{U} \\ \quad \end{array} \right] \cdot \left[\begin{array}{ccc} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & & \\ & \frac{\sigma_2^2}{\sigma_2^2 + \lambda} & \\ & & \ddots \\ & & & \frac{\sigma_m^2}{\sigma_m^2 + \lambda} \end{array} \right] \cdot \left[\begin{array}{c} \quad \\ \mathbf{U}^\top \\ \quad \end{array} \right]$$

- ii. **Derive the above expression.**

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

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

Solution: Sum of m completely decoupled ridge-regression approach:

Since the squared Frobenius norm is just the sum of squares of all the elements in the matrix, we can decouple them into sum of m l2 norms. Specifically:

$$\mathcal{L}_2(W; \mathbf{X}, \lambda) = \|\mathbf{X} - W\mathbf{X}\|_F^2 + \lambda\|W\|_F^2 = \sum_{i=1}^m \|\mathbf{X}_i - W_i\mathbf{X}\|_2^2 + \lambda\|W_i\|_2^2$$

where $\mathbf{X}_i \in \mathbb{R}^{1 \times n}$ and $W_i \in \mathbb{R}^{1 \times m}$ are rows. We can also write it as:

$$\mathcal{L}_2(W; \mathbf{X}, \lambda) = \sum_{i=1}^m \|\mathbf{X}_i^\top - \mathbf{X}^\top W_i^\top\|_2^2 + \lambda\|W_i^\top\|_2^2$$

which resembles the classical ridge regression optimization problem $\|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2$ which can be minimized with $\hat{\beta} = (X^\top X + \lambda I)^{-1}X^\top y$.

Through pattern matching $\mathbf{X}_i^\top = y, \mathbf{X}^\top = X, W_i^\top = \beta$, we can arrive with the result:

$$\hat{W}_i^\top = (\mathbf{X}\mathbf{X}^\top + \lambda I)^{-1}\mathbf{X}\mathbf{X}_i^\top$$

$$\hat{W} = \mathbf{X}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda I)^{-1}$$

Now substitute $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ to the above result, we have:

$$\begin{aligned} \hat{W} &= \mathbf{U}\Sigma\mathbf{V}^\top(\mathbf{U}\Sigma\mathbf{V}^\top)^\top(\mathbf{U}\Sigma\mathbf{V}^\top(\mathbf{U}\Sigma\mathbf{V}^\top)^\top + \lambda I)^{-1} \\ \hat{W} &= \mathbf{U}\Sigma\Sigma^\top\mathbf{U}^\top(\mathbf{U}\Sigma\Sigma^\top\mathbf{U}^\top + \lambda I)^{-1} \\ \hat{W} &= \mathbf{U}\Sigma\Sigma^\top\mathbf{U}^\top\mathbf{U}(\Sigma\Sigma^\top + \lambda I)^{-1}\mathbf{U}^\top \\ \hat{W} &= \mathbf{U}\Sigma\Sigma^\top(\Sigma\Sigma^\top + \lambda I)^{-1}\mathbf{U}^\top \end{aligned}$$

Solution: Vector calculus approach:

Alternatively we can solve this problem by taking the gradient of the loss function with respect to the weight matrix and finding the optimal point by setting it to zero.

$$\nabla \mathcal{L}_2(W; \mathbf{X}, \lambda) = \nabla(\|\mathbf{X} - W\mathbf{X}\|_F^2 + \lambda\|W\|_F^2) = 0$$

Using property of the frobenius norm $\|A\|_F^2 = \text{Tr}(A^T A)$ we can rewrite the objective function:

$$\|\mathbf{X} - W\mathbf{X}\|_F^2 + \lambda\|W\|_F^2 = \text{Tr}((\mathbf{X} - W\mathbf{X})(\mathbf{X} - W\mathbf{X})^T) + \lambda \text{Tr}(WW^T)$$

Next, expand out the terms in the trace and compute the gradient with respect to W .

$$\nabla_W \left[\text{Tr}(\mathbf{X}\mathbf{X}^T) - \text{Tr}(\mathbf{X}\mathbf{X}^T W) - \text{Tr}(W\mathbf{X}\mathbf{X}^T) + \text{Tr}(W\mathbf{X}\mathbf{X}^T W) + \lambda \text{Tr}(WW^T) \right]$$

$$= -2\mathbf{X}\mathbf{X}^T + 2W\mathbf{X}\mathbf{X}^T + 2\lambda W = 0$$

Solving for W we get

$$W = \mathbf{X}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda I)^{-1}$$

The problem can then be completed by substituting in $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ as in the alternative solution.

Solution: More elegant solution using the invariance (geometric) properties of trace and Frobenius norm.

The Frobenius norm is invariant under orthogonal transformation.

Proof: The Frobenius norm is equal to $\|A\|_F^2 = \text{tr}(AA^T)$, and since the trace has the cyclic property, we have $\|A\|_F^2 = \text{tr}(AA^T) = \text{tr}(U^TUAU^T) = \text{tr}(UAA^TU^T) = \|UA\|_F^2$ for any orthogonal matrix U . Similarly $\|A\|_F^2 = \|AV^T\|_F^2$ for any orthogonal matrix V .

Next, we expand

$$\begin{aligned} L &= \|X - WX\|_F^2 + \lambda\|W\|_F^2 \\ &= \|U\Sigma V^T - WU\Sigma V^T\|_F^2 + \lambda\|W\|_F^2 \\ &= \|\mathcal{U}(\Sigma - U^T W U \Sigma)\mathcal{V}^T\|_F^2 + \lambda\|W\|_F^2 \\ &= \|\Sigma - U^T W U \Sigma\|_F^2 + \lambda\|\mathcal{U}^T W U \mathcal{V}^T\|_F^2 \\ &= \|\Sigma - \tilde{W}\Sigma\|_F^2 + \lambda\|\tilde{W}\|_F^2 \end{aligned}$$

where we defined $\tilde{W} := U^T W U$, or alternatively,

$$W = U\tilde{W}U^T$$

Now since Σ is already diagonalized, minimizing $\|\Sigma - \tilde{W}\Sigma\|_F^2 + \lambda\|\tilde{W}\|_F^2$ is simple, since we can directly expand it:

$$\begin{aligned} L &= \sum_i (\sigma_i - \sigma_i \tilde{W}_{ii})^2 + \sum_{i \neq j} (\sigma_i \tilde{W}_{ij})^2 + \lambda \sum_i (\tilde{W}_{ii})^2 + \lambda \sum_{i \neq j} (\tilde{W}_{ij})^2 \\ &= \sum_i [(\sigma_i - \sigma_i \tilde{W}_{ii})^2 + \lambda(\tilde{W}_{ii})^2] + \sum_{i \neq j} (\sigma_i^2 + \lambda)\tilde{W}_{ij}^2 \end{aligned}$$

As we see, the loss splits into many quadratic losses, each one depending on a single entry of \tilde{W} .

Minimizing each, we find $\tilde{W} = \text{diag}\left(\frac{\sigma_i^2}{\sigma_i^2 + \lambda}\right)$.

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

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

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

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

Solution:

For the first condition, we have

$$\frac{\sigma_i^2}{\sigma_i^2 + \lambda} \geq 0.8 \quad \forall i \in \{1, 2, 3\}$$

Since σ_i are decreasing, we get

$$\frac{\sigma_3^2}{\sigma_3^2 + \lambda} \geq 0.8 \implies \frac{4^2}{4^2 + \lambda} \geq 0.8 \implies \lambda \leq 4$$

Similarly, solving for the second condition gives:

$$\frac{\sigma_4^2}{\sigma_4^2 + \lambda} \leq 0.6 \implies \frac{1}{1 + \lambda} \leq 0.5 \implies \lambda \geq 1$$

Thus,

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

4. Ridge-Attention

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

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

Solution: This question is a warmup. By multiplying m with n , we get that $mn = \sum_{i=1}^n x_i$, which means that $\sum_{i=1}^{n+1} x_i = mn + x_{n+1}$ and so $m' = \frac{mn+x_{n+1}}{n+1}$. In other words, we can update a running average with a new point in constant time. We don’t have to start from scratch.

- (b) Let us now shift to thinking about traditional ridge-regression with n training pairs (\mathbf{x}_i, y_i) where \mathbf{x}_i are d -dimensional vectors and y_i are scalars. Let the matrix $A = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix}$ and vector $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ so that we can find the familiar closed-form solution

$$\mathbf{w}_* = (A^\top A + \lambda I)^{-1} A^\top \mathbf{y} \tag{11}$$

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

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

i.e. Complete:

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

Solution: Linear regression and ridge-regression are just fancy ways to average. It is all about being able to see this clearly with math.

Using the outer-product form of matrix multiplication, since the columns of A^\top are \mathbf{x}_i and the rows of A are \mathbf{x}_i^\top , we know that $A^\top A = \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^\top$. From this we conclude that $(A^\top A + \lambda I) = \lambda I + \sum_{i=1}^n \mathbf{x}_i \mathbf{x}_i^\top$.

This will be useful later because it tells us that adding-in one more data point in ridge regression is tantamount to performing a rank-1 update on the matrix we need to invert.

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

Solution: Since the columns of A^\top are \mathbf{x}_i and matrix-vector multiplication consists of taking a linear combination of the columns of the matrix, we know that $A^\top \mathbf{y} = \sum_{i=1}^n \mathbf{x}_i y_i$.

This will be useful later because it tells us that adding one more data point in scalar-output ridge regression is tantamount to adding one more vector in the second part of (11).

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

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

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

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

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

Solution: The answer is $O(nd^2)$.

The key to understanding this is that the computation of non-causal ridge self-attention can be broken up into two phases. In the first phase, we compute a $d \times d$ matrix W_* that represents the learned linear-transformation from all the n data points: $(\mathbf{k}_i, \mathbf{v}_i)$ in the attention layer. In the second phase, we simply compute $W_*^\top \mathbf{q}_i$ for each of the n queries. The second phase consists of n matrix-vector products and so clearly takes $O(nd^2)$ time. But what about the first phase?

For the first phase, we need to compute:

$$W_* = (\lambda I + A^\top A)^{-1} A^\top B \quad (12)$$

where $A = \begin{bmatrix} \mathbf{k}_1^\top \\ \mathbf{k}_2^\top \\ \vdots \\ \mathbf{k}_n^\top \end{bmatrix}$ and $B = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_n^\top \end{bmatrix}$ since we are computing (11) exactly d times for the d different entries in the value vectors, amortizing the load of computing all but the last part.

The first matrix $(\lambda I + A^\top A)$ can be computed in time $O(nd^2)$ and inverted in time $O(d^3)$. The second matrix $A^\top B = \sum_{i=1}^n \mathbf{k}_i \mathbf{v}_i^\top$ can be computed in time $O(nd^2)$. The matrix-matrix multiplication can

also be performed in time $O(d^3)$. But since $n > d$ by assumption, each of the $O(d^3)$ times is also $O(nd^2)$ and this means that the total computation is just $O(nd^2)$ as well since there are a constant number of $O(nd^2)$ steps to perform.

In fact, you can make the computation of W_* effectively be $O(nd^2)$ steps when $n < d$ too. The matrix inversion can be computed starting with $(\lambda I)^{-1}$, then iterating with the Sherman–Morrison Formula (see a later part). Each step takes $O(d^2)$ time, and there are n steps, taking $O(nd^2)$ time in total.

But all this is not required for this part of the problem.

- (d) Assume that a ridge self-attention layer is used in a Transformer architecture and there is a downstream loss. **For which of these will backprop successfully pass gradients if we use ridge self-attention?**
- The ridge λ viewed as a learnable parameter for the self-attention layer.
 - The keys
 - The values
 - The queries

Solution: Gradients pass through to all of these terms easily because all the steps involved are fully differentiable. As far as the queries are concerned, ridge self-attention is just a linear operation of multiplying by W_* . While the keys, values, and ridge λ do factor into the computation of W_* in a nonlinear way, it is a differentiable function.

- (e) Now step back. There is a nice trick (called the Sherman–Morrison Formula) by which one can update the inverse of an invertible matrix to which you make a rank-1 update. Let M be an invertible square $d \times d$ matrix and let \mathbf{u}, \mathbf{v} be two d -dimensional vectors. Then:

$$(M + \mathbf{u}\mathbf{v}^\top)^{-1} = M^{-1} - \frac{1}{1 + \mathbf{v}^\top M^{-1} \mathbf{u}} (M^{-1}\mathbf{u})(\mathbf{v}^\top M^{-1}) \quad (13)$$

Assume that a $d \times d$ matrix times either a d -dimensional row or column vector costs d^2 operations, and so does the evaluation of a dyad $\mathbf{u}\mathbf{v}^\top$. Assume that computing a Euclidean inner-product costs d operations. **Assuming that you already had M^{-1} in hand, what is the computational cost of one application of (13)?**

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

Solution: The answer is $O(d^2)$. This is because $M^{-1}\mathbf{u}$ can be computed in $O(d^2)$ time as a matrix-vector product. And so can $\mathbf{v}^\top M^{-1}$ by the same logic. Computing the dyad $(M^{-1}\mathbf{u})(\mathbf{v}^\top M^{-1})$ in (13) is therefore $O(d^2)$ in total. $\mathbf{v}^\top M^{-1}\mathbf{u}$ is just a Euclidean-inner product that we can compute in $O(d)$ time, and then dividing each entry of a vector by a scalar also costs $O(d)$ time. The total is thus $O(d^2)$ computation to do an inverse of a rank-1 update to a matrix.

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

Describe explicitly how you would compute causal ridge-self-attention in a computationally efficient manner. Leverage your decomposition of the ridge-regression formula in part (b) of this problem together with the Sherman–Morrison formula from (13) to avoid having to do $O(n^2)$ computations while still calculating causal ridge-self-attention outputs correctly for all n positions in the context.

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

Solution: Since the GPT-3 paper awakened the community to the power of in-context learning that such models seem to exhibit, understanding where this capability is coming from was one of the key pieces of the intellectual agenda. The fact that softmax-attention itself is close to soft-nearest-neighbor style online learning was one very important key clue. This overall problem should help you see that conceptually, one has alternatives. What this part does is show you why recurrences can also exhibit this kind of online learning capability.

Simple recurrences also intrinsically make the computation be $O(n)$ as far as the total context-length is concerned because you just make one forward pass through the data, doing the same amount of computation for every single position. By contrast, traditional softmax-attention has to do a linearly increasing amount of computation for every position since the pool of potential neighbor (key, value) pairs is growing and we need to compute the score of the position-specific query with respect to each of them. That is what makes traditional softmax-attention take $O(n^2)$ computation with context-length n . If we were to treat each position t independently and start from scratch, causal ridge-self-attention would also end up taking $O(n^2)$ computation. But we can do better by reusing work via recurrences.

The key here is to do things step by step. For causal ridge attention, we need to understand the $W_*(t)$ so that the output of attention is $W_*^\top(t)\mathbf{q}_t$. We need to be able to compute this. If we have the $d \times d$ matrix $W_*(t)$, then computing $W_*^\top(t)\mathbf{q}_t$ just costs $O(d^2)$ operations.

Adapting (12) for causal ridge-self-attention merely requires redefining the the relevant A and B ma-

trices in that expression. Let $A_t = \begin{bmatrix} \mathbf{k}_1^\top \\ \mathbf{k}_2^\top \\ \vdots \\ \mathbf{k}_t^\top \end{bmatrix}$ and $B_t = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_t^\top \end{bmatrix}$ so that we just have the keys and values up to this point in time being used. Now, the counterpart to (12) is:

$$W_*(t) = (\underbrace{\lambda I + A_t^\top A_t}_{M_t})^{-1} \underbrace{A_t^\top B_t}_{J_t} \quad (14)$$

There are two ways that one can proceed to get an algorithm. First (and more ambitiously), we can try to get an efficiently computable recursion for $W_*(t)$ itself:

$$W_*(t) = (M_t)^{-1} J_t \quad (15)$$

$$= (\lambda I + \sum_{i=1}^t \mathbf{k}_i \mathbf{k}_i^\top)^{-1} (\sum_{i=1}^t \mathbf{k}_i \mathbf{v}_i^\top) \quad (16)$$

$$= (M_{t-1} + \mathbf{k}_t \mathbf{k}_t^\top)^{-1} (J_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top) \quad (17)$$

$$= (M_{t-1}^{-1} - \underbrace{\frac{1}{1 + \mathbf{k}_t^\top M_{t-1}^{-1} \mathbf{k}_t} (M_{t-1}^{-1} \mathbf{k}_t))(\mathbf{k}_t^\top M_{t-1}^{-1})}_{U_t}) (J_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top) \quad (18)$$

$$= M_{t-1}^{-1} J_{t-1} - U_t J_{t-1} + (M_{t-1}^{-1} \mathbf{k}_t) \mathbf{v}_t^\top - (U_t \mathbf{k}_t) \mathbf{v}_t^\top \quad (19)$$

where we can consider $M_0 = \lambda I$ and hence $M_0^{-1} = \lambda^{-1} I$ and $J_0 = \mathbf{0}$. Notice that to compute (17) recursively, we just need to invert a rank-1 update to a matrix that we have already inverted, as well as do a rank-1 update to a matrix that we have already computed.

The expression (19) immediately reveals what we need to track recursively across time: M_t^{-1} and J_t

for sure, and the product $W_*(t) = M_t^{-1}J_t$ as well since it can also be reused.

To be explicit: we have the following $O(d^2)$ computation-cost recursions for updating M_t^{-1} and J_t :

$$M_t^{-1} = M_{t-1}^{-1} - \left(\frac{1}{1 + \mathbf{k}_t^\top M_{t-1}^{-1} \mathbf{k}_t} (M_{t-1}^{-1} \mathbf{k}_t) (\mathbf{k}_t^\top M_{t-1}^{-1}) \right), \quad (20)$$

$$J_t = J_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top. \quad (21)$$

Going back to justify why the entire recursion for $W_*(t)$ given by (19) can also be done in $O(d^2)$ computation: Computing the first term $M_{t-1}^{-1}J_{t-1}$ is trivial since we already have it as $W_*(t-1)$. Doing the sums of all these terms in (19) together just costs us $O(d^2)$. Now, let us look at the individual other terms in (19).

Here, it is helpful to observe that U_t itself is actually a dyad inside:

$$U_t = \underbrace{\left(\frac{1}{1 + \mathbf{k}_t^\top M_{t-1}^{-1} \mathbf{k}_t} (M_{t-1}^{-1} \mathbf{k}_t) \right)}_{\mathbf{u}_a} \underbrace{(\mathbf{k}_t^\top M_{t-1}^{-1})}_{\mathbf{u}_b^\top} \quad (22)$$

and so the term $U_t J_{t-1} = \mathbf{u}_a (\mathbf{u}_b^\top J_{t-1})$ that can be computed as a vector-matrix multiplication followed by a dyad computation. So this is $O(d^2)$ given computing $\mathbf{u}_a, \mathbf{u}_b^\top$ themselves, which we already know is $O(d^2)$ as well. So all this is $O(d^2)$.

By similar reasoning, the term $(U_t \mathbf{k}_t) \mathbf{v}_t^\top = \mathbf{u}_a ((\mathbf{u}_b^\top \mathbf{k}_t) \mathbf{v}_t^\top)$ is also $O(d^2)$ as an inner-product ($O(d)$) followed by scaling a vector ($O(d)$) followed by a dyad calculation $O(d^2)$.

The final term to understand is $(M_{t-1}^{-1} \mathbf{k}_t) \mathbf{v}_t^\top$ which is clearly just a matrix-vector multiply followed by a dyad calculation — also $O(d^2)$.

This means that the entire operation of causal ridge-self-attention can be done in $O(nd^2)$ — the same order as non-causal ridge-self-attention.

The above answer is in great detail and gives us something that, strictly speaking, we do not really need: namely all the $W_*(t)$ for each time t as explicit $d \times d$ matrices. The other approach is to simply leverage the two $O(d^2)$ computation recursions (20) for updating M_t^{-1} — remembering to start at $M_0^{-1} = \lambda^{-1} I$ — and (21) for updating J_t — remembering to start at $J_0 = \mathbf{0}$. Then the causal ridge-self-attention answer for the query \mathbf{q}_t can be computed as $W_*^\top(t) \mathbf{q}_t = J_t^\top ((M_t^{-1})^\top \mathbf{q}_t)$ as a matrix-vector multiplication followed by another matrix-vector multiplication. This is $O(d^2)$ computation.

Either of these paths is sufficient for full credit on this problem, and we expected people to take the easier second path.

Notice what you have learned here: there are nonlinear¹ recursions with $O(d^2)$ dimensional state that permit one to exactly implement an attention-style mechanism that clearly does online learning from the given context. This is important since one of the ongoing and exciting efforts in the deep learning community is trying to figure out how to construct efficiently-computable recursions that can allow models to efficiently scale to very large² contexts.

¹The required nonlinearities in this case are analog multiplication and division. But it is known that we can approximate these with other nonlinearities combined with linear operations.

²We did not talk about it here in this course, but efficiently scaling to very large contexts is important for engineering settings where we want to use such generative models in conjunction with databases, etc. If we can count on the sophisticated generative model to figure out how to synthesize and distill relevant information from a lot of context, then we can use less sophisticated approaches to get approximately relevant stuff into the context. The larger the context we can productively use, the easier it is to make sure that the relevant information is there.

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

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

Solution: For a given query, we have $W_*^\top \mathbf{q}$ as our output. Looking at (12), this is

$$W_*^\top \mathbf{q} = ((\lambda I + A^\top A)^{-1} A^\top B)^\top \mathbf{q} \quad (23)$$

$$= B^\top (A(\lambda I + A^\top A)^{-1}) \mathbf{q} \quad (24)$$

Since $B^\top = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_n]$, we can see that the column given by expression $(A(\lambda I + A^\top A)^{-1}) \mathbf{q}$ gives us the counterpart of the attention weights. By itself, this is not very interpretable although it does answer the question. To make this slightly more interpretable, we can look at the i -th row of this vector and notice by the properties of matrix multiplication that it is $\mathbf{k}_i^\top (\lambda I + A^\top A)^{-1} \mathbf{q} = \langle \mathbf{q}, (\lambda I + A^\top A)^{-1} \mathbf{k}_i \rangle$. In other words, the i -th attention weight is the inner product between the query and a context-adjusted i -th key.

You could also do this calculation via kernel ridge regression which has the same predictions as regular ridge-regression, but uses a different formula which factors through an $n \times n$ matrix instead of $d \times d$. Recall that in place of $(A^\top A + \lambda I_{d \times d})^{-1} A^\top$, we use $A^\top (\lambda I_{n \times n} + AA^\top)^{-1}$ in kernel-ridge regression. Following the approach above, you would therefore get that the attention weights are: $(\lambda I_{n \times n} + AA^\top)^{-1} A \mathbf{q}$. Notice that $A \mathbf{q}$ is a column which consists of the same n inner-products between the n keys and the query as would be computed in standard soft-max attention. The difference is that in ridge-attention, instead of normalizing these inner-products into attention-weights by using a softmax, we have a “normalizing” matrix $(\lambda I_{n \times n} + AA^\top)^{-1}$ that only depends on the pairwise inner-products of all the keys with each other.

This problem hopefully serves to simultaneously de-mystify attention weights while also giving you leverage in thinking more clearly about what is going on.

5. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!

We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

- (a) **What sources (if any) did you use as you worked through the homework?**
- (b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)
- (c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Naman Jain.
- Qiyang Li.
- Dhruv Shah.
- Anant Sahai.