Efficient Inversion Method for "Diagonal + Low-Rank" Triangular Matrices

Jianlin Su

July 1, 2025

From the article "A Brief History of Linear Attention: From Imitation, Innovation to Feedback", we can observe that DeltaNet and subsequent linear Attention models are basically associated with the inverse matrix $(\boldsymbol{I} + \boldsymbol{K}\boldsymbol{K}^\top \odot \boldsymbol{M}^-)^{-1}$. This article specifically explores the calculation of the inverse of such triangular matrices characterized by a "diagonal + low-rank" structure.

1 Basic Results

We define the problem generally as follows:

> Given matrices $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{n\times d}$ and a diagonal matrix $\boldsymbol{\Lambda} \in \mathbb{R}^{n\times n}$, satisfying $n \gg d$, define
> $$\boldsymbol{T} = \boldsymbol{\Lambda} + \boldsymbol{Q}\boldsymbol{K}^\top \odot \boldsymbol{M}^- \tag{1}$$
> where $\boldsymbol{M}^- = \boldsymbol{M} - \boldsymbol{I}$, and the matrix $\boldsymbol{M}$ is defined as
> $$M_{i,j} = \begin{cases} 1, & i \geq j \\ 0, & i < j \end{cases} \tag{2}$$
> The goal is to find the inverse matrix $\boldsymbol{T}^{-1}$ and prove that its complexity is $\mathcal{O}(n^2)$.

First, if there were no lower triangular constraint $\odot\boldsymbol{M}^-$, the problem could be directly solved by the Woodbury matrix identity:

$$(\boldsymbol{\Lambda} + \boldsymbol{Q}\boldsymbol{K}^\top)^{-1} = \boldsymbol{\Lambda}^{-1} - \boldsymbol{\Lambda}^{-1}\boldsymbol{Q}(\boldsymbol{I} + \boldsymbol{K}^\top\boldsymbol{\Lambda}^{-1}\boldsymbol{Q})^{-1}\boldsymbol{K}^\top\boldsymbol{\Lambda}^{-1} \tag{3}$$

It is easy to verify that the computational complexity of the right-hand side is $\mathcal{O}(n^2)$. However, after adding $\odot\boldsymbol{M}^-$, $\boldsymbol{T}$ itself no longer possesses the "diagonal + low-rank" structure, so it cannot be solved directly by this identity. Given the lower triangular characteristic, a basic approach is recursion, as we have the block matrix identity:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{C} & \boldsymbol{B} \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{A}^{-1} & \boldsymbol{0} \\ -\boldsymbol{B}^{-1}\boldsymbol{C}\boldsymbol{A}^{-1} & \boldsymbol{B}^{-1} \end{bmatrix} \tag{4}$$

This allows us to transform $\boldsymbol{T}^{-1}$ into a recursive form (convention: in the absence of parentheses, slicing has the highest precedence):

$$\boldsymbol{T}^{-1}_{[:l+1,:l+1]} = \begin{bmatrix} \boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{0} \\ -\boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\boldsymbol{T}_{[l:l+1,:l]}\boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{T}^{-1}_{[l:l+1,l:l+1]} \end{bmatrix} \tag{5}$$

The main calculation here is $\boldsymbol{T}_{[l:l+1,:l]}\boldsymbol{T}^{-1}_{[:l,:l]}$, which is a multiplication of a $1 \times l$ and an $l \times l$ matrix. The complexity is $\mathcal{O}(l^2)$, meaning the complexity of each iteration grows quadratically, resulting in a total complexity of $\mathcal{O}(n^3)$.

2 Low-Rank Structure

Of course, this is because we haven't yet utilized the low-rank structure of $\boldsymbol{T}$ (before the $\odot \boldsymbol{M}^-$ operation). By utilizing it, we get $\boldsymbol{T}_{[l:l+1,:l]} = \boldsymbol{Q}_{[l:l+1]}\boldsymbol{K}_{[:l]}^{\top}$. Substituting this into the above equation yields:

$$\boldsymbol{T}^{-1}_{[:l+1,:l+1]} = \begin{bmatrix} \boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{0} \\ -\boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\boldsymbol{Q}_{[l:l+1]}\boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{T}^{-1}_{[l:l+1,l:l+1]} \end{bmatrix} \tag{6}$$

Note that $\boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]} \in \mathbb{R}^{d\times l}$. If we can use this as a recursive variable, the complexity of each iteration will only be $\mathcal{O}(l)$, and the total complexity can be successfully reduced to $\mathcal{O}(n^2)$. Following this logic, we have:

$$\begin{aligned} \boldsymbol{K}^{\top}_{[:l+1]}\boldsymbol{T}^{-1}_{[:l+1,:l+1]} &= \begin{bmatrix} \boldsymbol{K}^{\top}_{[:l]} & \boldsymbol{K}^{\top}_{[l:l+1]} \end{bmatrix} \begin{bmatrix} \boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{0} \\ -\boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\boldsymbol{Q}_{[l:l+1]}\boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{T}^{-1}_{[l:l+1,l:l+1]} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{0} \end{bmatrix} + \boldsymbol{K}^{\top}_{[l:l+1]} \underbrace{\begin{bmatrix} -\boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\boldsymbol{Q}_{[l:l+1]}\boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{T}^{-1}_{[l:l+1,l:l+1]} \end{bmatrix}}_{\text{which is actually } (\boldsymbol{T}^{-1})_{[l:l+1,:l+1]}} \end{aligned} \tag{7}$$

As we can see, this recursive process does not involve $\mathcal{O}(l^2)$ operations. Therefore, the approach is feasible; we only need to introduce a new variable to cache $\boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]}$. If we replace $l+1$ with $l+c$, we can obtain the recursion in a chunked format.

The test code is as follows:

```python
import numpy as np

n, d, c = 1000, 100, 200
Q = np.random.randn(n, d) / d**0.5
K = np.random.randn(n, d) / d**0.5
T = np.tril(Q @ K.T, -1) + np.eye(n)

Y, Z = np.zeros((n, n)), np.zeros((d, n))
for l in range(0, n, c):
    Y[l:l + c, l:l + c] = np.linalg.inv(T[l:l + c, l:l + c])
    Y[l:l + c, :l] = - Y[l:l + c, l:l + c] @ Q[l:l + c] @ Z[:, :l]
    Z[:, :l + c] += K[l:l + c].T @ Y[l:l + c, :l + c]

print(np.allclose(Y @ T, np.eye(n)))
```

3 Multiplication Calculation

Based on the same idea, we can also prove:

For any matrix $\boldsymbol{V} \in \mathbb{R}^{n\times d}$, calculating $\boldsymbol{T}^{-1}\boldsymbol{V}$ only requires $\mathcal{O}(n)$ complexity.

The proof only requires a slight modification of the previous process. First, we have:

$$\begin{aligned}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[:l+1]} &= \boldsymbol{T}^{-1}_{[:l+1,:l+1]}\boldsymbol{V}_{[:l+1]} \\ &= \begin{bmatrix}\boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{0} \\ -\boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\boldsymbol{Q}_{[l:l+1]}\boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]} & \boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\end{bmatrix}\begin{bmatrix}\boldsymbol{V}_{[:l]} \\ \boldsymbol{V}_{[l:l+1]}\end{bmatrix} \\ &= \begin{bmatrix}\boldsymbol{T}^{-1}_{[:l,:l]}\boldsymbol{V}_{[:l]} \\ -\boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\boldsymbol{Q}_{[l:l+1]}\boldsymbol{K}^{\top}_{[:l]}\boldsymbol{T}^{-1}_{[:l,:l]}\boldsymbol{V}_{[:l]} + \boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}\boldsymbol{V}_{[l:l+1]}\end{bmatrix} \\ &= \begin{bmatrix}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[:l]} \\ \boldsymbol{T}^{-1}_{[l:l+1,l:l+1]}(\boldsymbol{V}_{[l:l+1]} - \boldsymbol{Q}_{[l:l+1]}\boldsymbol{K}^{\top}_{[:l]}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[:l]})\end{bmatrix}\end{aligned} \tag{8}$$

Then:

$$\begin{aligned}\boldsymbol{K}^{\top}_{[:l+1]}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[:l+1]} &= \begin{bmatrix}\boldsymbol{K}^{\top}_{[:l]} & \boldsymbol{K}^{\top}_{[l:l+1]}\end{bmatrix}\begin{bmatrix}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[:l]} \\ (\boldsymbol{T}^{-1}\boldsymbol{V})_{[l:l+1]}\end{bmatrix} \\ &= \boldsymbol{K}^{\top}_{[:l]}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[:l]} + \boldsymbol{K}^{\top}_{[l:l+1]}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[l:l+1]}\end{aligned} \tag{9}$$

Therefore, by only caching $\boldsymbol{K}^{\top}_{[:l]}(\boldsymbol{T}^{-1}\boldsymbol{V})_{[:l]} \in \mathbb{R}^{d\times d}$, the computational complexity of each step becomes independent of l, so the total complexity is $\mathcal{O}(n)$. Similarly, replacing $l+1$ with $l+c$ yields the chunked format.

The test code is as follows:

```python
import numpy as np

n, d, c = 1000, 100, 200
Q = np.random.randn(n, d) / d**0.5
K = np.random.randn(n, d) / d**0.5
V = np.random.randn(n, d) / d**0.5
T = np.tril(Q @ K.T, -1) + np.eye(n)

Y, Z = np.zeros((n, d)), np.zeros((d, d))
for l in range(0, n, c):
    X = np.linalg.inv(T[l:l + c, l:l + c])
    Y[l:l + c] = X @ (V[l:l + c] - Q[l:l + c] @ Z)
    Z += K[l:l + c].T @ Y[l:l + c]

print(np.allclose(T @ Y, V))
```

4 Summary

This article discussed the inversion problem of triangular matrices with "diagonal + low-rank" characteristics. Such matrices commonly appear in modern linear Attention models.