How Is the tid2eid in DeepSeek V4 Generated?

Su Jianlin

2026-05-15

Introduction

Students who have trained MoE models know that if the entire MLP portion of the model is replaced with MoE, the first few layers near the Embedding layer often struggle with load balancing. To address this, DeepSeek V3, as well as our own Kimi K2, adopted the strategy of "first_k_dense"—as the name suggests, the first k layers use conventional Dense-type GLU instead of MoE. In DeepSeek V4, this strategy has been changed to "first_k_hash".

Here, "hash" refers to "Hash Routing," proposed in "Hash Layers For Large Sparse Models", which assigns each Token to an Expert using a pre-determined mapping table called `tid2eid` (Token Id to Expert Id). This article explores how this `tid2eid` table is generated.

1 Problem Background

First, it should be noted that for the first few MoE layers, if one applies the Quantile Balancing method proposed by the author in "MoE Wonderland: 6. Optimal Allocation for Balance" and "MoE Wonderland: 7. A Minimalist Solution for Dynamic Activation", the imbalance issue can be significantly alleviated.

The adoption of Hash Routing in V4 can be seen as an attempt in a different direction for the same problem. The idea is that for the first few MoE layers, contextual information is limited, so selecting an Expert based on the input vector may not differ much from selecting based solely on the Token ID, which carries no context. If this is the case, it might be better to simply pre-determine which Expert each Token should activate in the first few layers using the Token ID directly—hence the origin of "`tid2eid` (Token Id to Expert Id)".

So how is the `tid2eid` table generated? Can it be completely random? At first glance, this seems unwise, since the frequency of different Tokens varies; complete randomness could actually worsen imbalance. DeepSeek has not disclosed the details of how this table is generated, so we can only speculate based on our own reasoning.

2 Mathematical Description

Suppose there are m distinct Tokens, and the frequency of the i-th Token is p_i. Without loss of generality, assume they are already sorted in decreasing order, i.e., $p_i \geq p_{i+1}$. Let $x_{i,j} \in \{0,1\}$ indicate whether Token i should activate Expert j (0 means no activation, 1 means activation), with a total of n Experts and each Token selecting k Experts. Then we are effectively solving the system:

$$x_{i,j} \in \{0,1\}, \qquad \sum_{j=1}^{n} x_{i,j} = k, \qquad \sum_{i=1}^{m} p_i x_{i,j} \approx \frac{k}{n} \tag{1}$$

Note that we used $\approx$ (approximately equal) in the last equation because changing it to $=$ might make the system unsolvable in strict terms. Thus, we retain $\approx$, meaning both sides should be as close as possible. Alternatively, we can express this as an optimization problem:

$$\min_{x_{i,j} \in \{0,1\}} \sum_{j=1}^{n} \left(\sum_{i=1}^{m} p_i x_{i,j} - \frac{k}{n} \right)^2 \qquad \text{s.t.} \qquad \sum_{j=1}^{n} x_{i,j} = k \tag{2}$$

A simple greedy algorithm can provide a reasonable solution to this problem:

> Process Tokens one by one in decreasing order of frequency. For each Token, first count the current load on each Expert, then select the k least-loaded Experts as the ones to activate for this Token, and update the load distribution accordingly.

3 Reference Implementation

Although the greedy algorithm is based on a greedy approach in principle, in most cases it yields a near-optimal solution. A reference implementation is as follows:

```python
import numpy as np

# Simulated distribution
m, n, k = 80000, 128, 4
p = 1 / (10 + np.arange(m))
p /= p.sum()

# Greedy processing
x, f = np.zeros((m, k), dtype='int32'), np.zeros(n)
for i in range(m):
    j = f.argsort()[:k]  # Select the k least-loaded Experts
    f[j] += p[i] / k  # Update load distribution
    x[i] = j  # Record into tid2eid

# Evaluate balance
max_vio, min_vio = f.max() * n - 1, f.min() * n - 1
```

Note that this code contains no randomness and is in principle deterministic. But what if we want different `tid2eid` mappings for the first few layers? One could introduce some randomness, for example, by replacing `range(m)` with `np.random.permutation(m)`, i.e., processing Tokens in a random order rather than by frequency. This still produces usable solutions, though with slightly worse balance. We can run this multiple times and pick the solutions with the lowest `max_vio`.

4 Extreme Cases

Now consider an extreme case: $p_1 \gg k/n$, i.e., the frequency of one Token is far above the balanced level. In this situation, no matter how `tid2eid` is arranged, load balance cannot be achieved. In other words, making decisions based only on a single Token ID cannot achieve balance under such conditions. How should we handle this?

The answer is simple: adopt a different Hash method that depends on more input information. Previously, we made decisions based only on the current Token ID. In reality, each Token exists within a sequence and thus has context. If the earlier approach is called "1-gram to expert", we can now consider including the previous Token(s) to form "2-gram to expert", "3-gram to expert", etc.

Take the 2-gram (a, b) as an example. A simple Hash method would be: choose a prime $q > m$, compute $(a \cdot q + b) \mod n$ as the activated Expert. Repeat this with k different primes to get k Experts. By increasing the number of grams, the number of possible input combinations increases significantly. When mapped to a finite number n of Experts, each Expert is very likely to be "filled". Therefore, as long as the Hash function is not particularly poor, the result will be nearly balanced.

Thus, the advantage is that we no longer need to consider frequency or pre-store a `tid2eid` table. The drawback is slightly more complex Hash computation and the possibility that a Token might select the same Expert multiple times—but these are not serious issues.

5 Summary

This article briefly reviews the basic idea behind Hash Routing in DeepSeek V4 and focuses on discussing the construction principle of its `tid2eid` mapping table.