MonoidReduce
or
Commutative Monoids and Memory Efficient Layers in Neural Networks.

Amaru Cuba Gyllensten

May 2025

Abstract

Recent advances in memory-efficient neural network layers, such as FlashAttention, appear as specialized engineering solutions but share a common mathematical structure. We show that many of these kernels can be understood as folds over commutative monoids, a perspective that unifies MapReduce-style computation with modern deep learning optimizations. Building on this, we introduce the Local Gradient Theorem, which provides a sufficient condition under which gradients of monoidal folds can be computed locally from the final output and individual inputs, enabling efficient backward passes without storing all intermediates. We demonstrate that attention, cross entropy, and two-layer MLPs all admit such monoid structures, recovering known memory-efficient kernels and extending the framework to new settings. This algebraic perspective offers a principled foundation for systematically designing memory- and cache-efficient layers, rather than discovering them in an ad-hoc manner.

1 Introduction

The dramatic growth of neural networks, particularly large language models (LLMs), is constantly pushing the boundaries of available computational resources. A significant bottleneck in both training and deploying these models is their substantial memory footprint, especially on accelerator hardware like GPUs with limited high-bandwidth memory. Operations such as self-attention, where a naive implementation yields quadratic memory complexity with regards to sequence length, pose a particular challenge, severely limiting the context window, batch size, and overall model scale.

To address these critical memory constraints, the community has developed a range of innovative memory-efficient kernels. A seminal work in this area is FlashAttention (Dao et al., 2022). FlashAttention introduces an I/O-aware

exact attention algorithm that employs tiling to significantly reduce the number of memory reads and writes between different levels of GPU memory, effectively transforming the memory complexity from quadratic to linear in sequence length. This not only facilitated the use of much longer context windows but also yielded substantial training speedups.

Other work, such as Hsu et al. (2024) provides a suite of optimized Triton kernels specifically designed for efficient LLM training. These kernels leverage techniques such as aggressive operator fusion, in-place gradient computation, and input chunking to dramatically boost throughput and reduce memory usage across various fundamental neural network primitives, including RMSNorm, RoPE, SwiGLU, and CrossEntropy. Similarly, the work Cut your losses (Wijmans et al., 2024) introduces Cut Cross-Entropy (CCE), an approach to mitigate the memory demands of the classification head in models with large vocabularies. CCE achieves a negligible memory footprint for loss computation by reformulating the arithmetic and fusing operations, enabling scaling to arbitrarily large vocabularies without sacrificing training speed or convergence.

While these individual advancements offer immense practical benefits, they often appear as specialized, ad-hoc solutions tailored to particular layers or operations. A closer examination, however, reveals a common underlying mathematical structure that unifies their memory-saving strategies. This paper argues that many of these advanced memory-efficient kernels can be systematically reinterpreted as highly optimized folds over commutative monoids, where computations are reduced to combining partial results in a memory-efficient and inherently parallelizable manner. By formalizing this common pattern under a unified framework based on commutative monoids, we aim to provide a principled approach not only for understanding existing optimizations but also for systematically designing novel memory-efficient layers. Furthermore, we introduce the Local Gradient Theorem, specifying a sufficient condition for efficient backward passes of commutative monoidal folds, which enables local gradient computation and thereby further reduces memory and communication overhead during training.

This paper employs liberal use of type and lambda notation borrowed from functional programming. A short primer of this notation and the mathematical conventions used can be found in appendix A.

1.1 What are commutative monoids?

A monoid S is a tuple $(T, \odot, \mathbf{id})$, where T is a type, $\odot : T \to T \to T$ a binary operation on T, and $\mathbf{id}$ a term of type T. The binary operation must be associative, i.e. $a \odot (b \odot c) = (a \odot b) \odot c$, and $\mathbf{id}$ must be an identity element, i.e. $\mathbf{id} \odot a = a = a \odot \mathbf{id}$. Commutative monoids are monoids in which the binary operation is also commutative, that is, $a \odot b = b \odot a$.

There are many instances of commutative monoids in modern deep learning:

Sum Summation over real numbers:

$$\begin{aligned} T &= \mathbb{R} \\ x \odot y &= x + y \\ \mathbf{id} &= 0 \end{aligned}$$

WSum Weighted sums of vectors.

$$\begin{aligned} T &= \{v : \mathbb{R}^D, w : \mathbb{R}_{\geq 0}\} \\ x \odot y &= z \\ \text{where } z_w &= x_w + y_w \\ z_v &= \begin{cases} x_v \frac{x_w}{z_w} + y_v \frac{y_w}{z_w}, & z_w > 0 \\ 0, & \text{otherwise} \end{cases} \\ \mathbf{id} &= \{v : 0, w : 0\} \end{aligned}$$

LogWSum Weighted sums of vectors with weights in logspace[1]:

$$\begin{aligned} T &= \{v : \mathbb{R}^D, w : \mathbb{R}\} \\ x \odot y &= z \\ \text{where } z_w &= \ln(\exp(x_w) + \exp(y_w)) \\ z_v &= \begin{cases} x_v \exp(x_w - z_w) + y_v \exp(y_w - z_w), & z_w \neq -\infty \\ 0, & \text{otherwise} \end{cases} \\ \mathbf{id} &= \{v : 0, w : -\infty\} \end{aligned}$$

Given a commutative monoid S, we can derive a function $fold$ that takes as input a sequence of Ts and returns the product over all elements in the sequence: $fold : [T] \to T = X \mapsto \bigodot_i X_i$.

2 Properties of Commutative Monoids in the context of Machine Learning

Commutative monoidal folds offer a couple of benefits in the context of large-scale machine learning:

- There is no need to materialize all values in memory during the computation of a fold (forward pass), enabling better usage of GPU-memory.
- Partial results can be computed and combined in arbitrary order, reducing the need for synchronization and enabling a myriad of parallelization schemes.

[1] This monoid applies to attention: $\text{softmax}(a)V = x_0 \odot x_1 \odot \cdots$, where $x_i = \{v = V_i, w = a_i\}$

- *If* the derivative $\frac{d\ x \odot y}{d\ x}$ can be expressed as a function of $x \odot y$ and x, then the same holds for $\frac{d\ fold(X)}{d\ X_i}$, allowing local computation of gradients based only on the final result, thus reducing the need for communication and synchronization during the backward pass.

Theorem 2.1 (Local gradient theorem)**.** *Given a commutative monoid* $(T, \odot, \mathbf{id})$ *whose derivative* $\frac{d\ x \odot y}{d\ x}$ *is a function (D) of* $x \odot y$ *and* x*:*

$$\frac{d\ x \odot y}{d\ x} = D(x \odot y, x) \tag{1}$$

We have that for any sequence $X : [T]$*, the following holds:*

$$\frac{d\ fold(X)}{d\ X_i} = D(fold(X), X_i)$$

Proof. Let $P = fold(X)$, for any partition of X into X^L, X^R, with corresponding partial products $P_L = fold(X^L)$ and $P_R = fold(X^R)$.

$$\begin{aligned} P = P_L \odot P_R = P_R \odot P_L \qquad & \text{(By commutativity and associativity)} \\ \frac{d\ P}{d\ P_L} &= \frac{d\ P_L \odot P_R}{d\ P_L} \\ &= D(P_L \odot P_R, P_L) \qquad \text{(By the condition from eq. 1)} \\ &= D(P, P_L) \end{aligned}$$

This extends trivially to $P_L = X_i$, i.e. $X^L = [X_i]$.

□

Corollary 2.1.1. *Let* $P = fold(X)$*. If we satisfy eq. 1, then for any partition of* X *into* $X^1, X^2, \ldots, X^n$*, with corresponding partial products* $P_i = fold(X^i)$*, we have that*

$$\frac{d\ P}{d\ P_i} = D(P, P_i)$$

this extends trivially to $P_i = X_i$*, i.e.* $X^i = [X_i]$*.*

Corollary 2.1.2. *Consider the scenario where* $X_i = f(a, B_i)$ *and let* $P = fold(X)$*. If we satisfy eq. 1, then for any partition of* B *into* $B^1, B^2, \ldots B^n$ *with corresponding partitions of* X *into* X^1*,* $X^2, \ldots, X^n$ *and partial products* $P_i = fold(X^i)$*, we have that:*

$$\frac{d\ P}{d\ a} = \sum_i \frac{d\ P_i}{d\ a} \cdot D(P, P_i)$$

$$\frac{d\ P}{d\ B^i} = \frac{d\ P_i}{d\ B^i} \cdot D(P, P_i)$$

this extends trivially to $P_i = X_i$*, i.e.* $X^i = [X_i], B^i = [B_i]$*.*

Corollary 2.1.3. *Consider the scenario where $X_{ij} = f(A_i, B_j)$ and let P be the vector of final products ($P_i = fold(X_i) = \bigodot_j X_{ij}$). If we satisfy eq. 1, then we have that:*

$$\frac{d\ P}{d\ A_i} = \sum_j \frac{d\ X_{ij}}{d\ A_i} \cdot D(P_i, X_{ij}) \cdot \frac{d\ P}{d\ P_i}$$
$$\frac{d\ P}{d\ B_j} = \sum_i \frac{d\ X_{ij}}{d\ B_j} \cdot D(P_i, X_{ij}) \cdot \frac{d\ P}{d\ P_i}$$

And more generally that for any partition of A into $A^1, A^2, \ldots, A^m$ and B into $B^1, B^2, \ldots, B^n$, with corresponding partitions of X into $X^{11}, X^{12}, \ldots, X^{mn}$ and partial product vectors P^{ij}, where P^i denotes the vector of products $P^i_k = \bigodot_j P^{ij}_k$:

$$\frac{d\ P}{d\ A^i} = \sum_j \frac{d\ P^{ij}}{d\ A^i} \cdot D(P^i, P^{ij}) \cdot \frac{d\ P}{d\ P^i}$$
$$\frac{d\ P}{d\ B^j} = \sum_i \frac{d\ P^{ij}}{d\ B^j} \cdot D(P^i, P^{ij}) \cdot \frac{d\ P}{d\ P^i}$$

Note: $\frac{d\ P}{d\ P_i}$ and $\frac{d\ P}{d\ P^i}$ are simple diagonal projections.

The implication of this is that gradients can be computed based only on local recomputation (and the relevant final products), which in turn facilitates highly parallel and cache-efficient kernels for backward passes.

During the forward pass, we can parallelize over slices of A and iterate over B-values, aggregating the product in local memory, and/or parallelize over slices of B-values and perform a parallel fold of the partial results.

During the backward pass, we can parallelize over tied slices of P and A, and B, compute partial gradients based on the local gradient theorem, and add them to the appropriate slices of the gradients of A and B (necessitating either atomic adds to global memory, or more refined aggregation schemes for the gradients for A and B).

Corollaries 2.1.2 and 2.1.3 also provide a general method for computing the gradient for A during a fold over B, limited by how efficiently the Jacobian sum can be represented. In the case where T is essentially a scalar[2] this is manageable as the resulting Jacobian has the same shape as A, but for more complex monoids the cost of keeping the Jacobian in memory might be prohibitively expensive. When possible, such an approach simplifies the backwards pass further, enabling us to apply the Jacobian to $\frac{d\ L}{d\ P}$ to get the gradients for A, while simultaneously compute the B-gradient by parallelizing over slices of B, iterating over A-values, and aggregating the B-slice gradient in local memory (and/or parallelize over slices of A as well and perform a parallel fold).

[2]For example, where $a \odot b = \text{logaddexp(a, b)}$ i.e. $fold(X) = \text{logsumexp}(X)$

2.1 Example: Attention

The benefit of not materializing all partial values is most apparent in the case where the fold is applied to the results of matrix multiplications, one such instance is attention:

$$
\begin{aligned}
Q &: \mathbb{R}^{M\times F}\\
K &: \mathbb{R}^{N\times F}\\
V &: \mathbb{R}^{N\times D}\\
\text{attention}(Q,K,V) &= \text{softmax}(QK^\intercal)V\\
\text{attention}(Q,K,V)_i &= Y_{iv}\\
&\quad \text{where}\\
&\quad Y_i = \bigodot_j H_{ij}\\
&\quad H_{ij} = \{z = Q_i K_j^\intercal,\ v = V_j\}\\
&\quad a \odot b = \begin{Bmatrix} z = & \ln(e^{a_z} + e^{b_z}) \\ v = & a_v e^{a_z - z} + b_v e^{b_z - z} \end{Bmatrix}
\end{aligned}
$$

Here, we can think of H as an M by N matrix of T-values, with each such value consisting of an attention weight and value. Combining two such values a and b results in a new total weight corresponding to the sum of their respective non log-space weights, and a new value corresponding to the weighted average of the two values. Since the result is a fold over the N-dimension, the full matrix does not have to be materialized. We can compute appropriate chunks of the matrix, fold the chunk, and add to a running total, only realizing the M normalizing factors, and the $M \times D$-sized weighted sum. Flash attention Dao et al. (2022) can be seen as an instance of this general approach.

We also have that condition 1 holds:

$$
\begin{aligned}
c &= a \odot b\\
\frac{d\ c}{d\ a} &= g \mapsto \begin{pmatrix} z: & (g_z + \langle g_v, a_v - c_v\rangle)\exp(a_z - c_z) \\ v: & g_v \exp(a_z - c_z) \end{pmatrix}\\
&= D(c,a)
\end{aligned}
$$

Which due to corollary 2.1.3 implies that the gradient w.r.t. Q, K and V can be calculated based on the final product Y and local (recomputed) H-values. For pseudocode of such an implementation, see B.

In table 1 we give further examples of this approach applied to cross entropy against class indices, cross entropy between two distributions, and two-layer MLPs, all of which satisfy the condition for the local gradient theorem (see Appendix D).

Input	Map	Reduce
$Q : \mathbb{R}^{M \times F}$ $K : \mathbb{R}^{N \times F}$ $V : \mathbb{R}^{N \times D}$ $\downarrow$ $Y : \mathbb{R}^{M \times D}$	$T : \{z : \mathbb{R}, v : \mathbb{R}^D\}$ $H : T^{M \times N}$ $H_{ij} = \begin{Bmatrix} z : & Q_i K_j^\intercal \\ v : & V_j \end{Bmatrix}$	$A : T^M$ $A_i = \bigodot_j H_{ij}$ $a \odot b = \begin{Bmatrix} z : & \ln(e^{a_z} + e^{b_z}) \\ v : & a_v e^{a_z - z} + b_v e^{b_z - z} \end{Bmatrix}$ $Y_i = A_{iv}$
Attention: $Y = \text{softmax}(QK^\intercal)V$		
$P : \mathbb{R}^{M \times D}$ $C : \mathbb{R}^{N \times D}$ $T : N^M$ $\downarrow$ $Y : \mathbb{R}^M$	$T : \{p : \mathbb{R}, n : \mathbb{R}\}$ $H : T^{M \times N}$ $H_{ij} = \begin{Bmatrix} p : & P_i C_j^\intercal \\ n : & T_i = j \ ? \ p \ : \ 0 \end{Bmatrix}$	$A : T^M$ $A_i = \bigodot_j H_{ij}$ $a \odot b = \begin{Bmatrix} p : & \ln(e^{a_p} + e^{b_p}) \\ n : & a_n + b_n \end{Bmatrix}$ $Y_i = A_{ip} - A_{in}$
Cross Entropy: $Y = \text{cross-entropy}(\text{logits} = PC^\intercal, \text{targets} = T)$		
$P^s : \mathbb{R}^{M \times D}$ $C^s : \mathbb{R}^{N \times D}$ $P^t : \mathbb{R}^{M \times E}$ $C^t : \mathbb{R}^{N \times E}$ $\downarrow$ $Y : \mathbb{R}^M$	$T : \{p : \mathbb{R}, q : \mathbb{R}, n : \mathbb{R}\}$ $H : T^{M \times N}$ $H_{ij} = \begin{Bmatrix} q : & P_i^s C_j^{s\intercal} \\ p : & P_i^t C_j^{t\intercal} \\ n : & q \end{Bmatrix}$	$A : T^M$ $A_i = \bigodot_j H_{ij}$ $a \odot b = \begin{Bmatrix} q : & \ln(e^{a_q} + e^{b_q}) \\ p : & \ln(e^{a_p} + e^{b_p}) \\ n : & a_n e^{a_p - p} + b_n e^{b_p - p} \end{Bmatrix}$ $Y_i = A_{iq} - A_{in}$
Cross Entropy: $Y = \text{cross-entropy}(\text{logits} = P^s C^{s\intercal}, \text{targets} = \text{softmax}(P^t C^{t\intercal}))$		
$X : \mathbb{R}^{B \times M}$ $P : \mathbb{R}^{K \times M}$ $Q : \mathbb{R}^{K \times N}$ $\downarrow$ $Y : \mathbb{R}^{B \times N}$	$T : \{v : \mathbb{R}^N\}$ $H : T^{B \times K}$ $H_{ij} = \{v : \sigma(X_i P_j^\intercal) Q_j\}$	$A : T^B$ $A_i = \bigodot_j X_{ij}$ $a \odot b = a_v + b_v$ $Y_i = A_{iv}$
MLP: $Y = \sigma(XP)Q$		

Table 1: Examples of Monoids T, map functions from inputs to intermediate H-values, reduction operations, and out projections that realizes different operations used in Artificial Neural Networks.

2.2 Batching and Compute efficiency

The pseudocode in listing 1 works elementwise, this is not really appropriate for efficient computation. A more GPU-friendly approach would be to compute the intermediate values and the gradients over appropriately sized slices of the H-matrix while simultaneously folding it. Thankfully, corollary 2.1.3 deals with this exact situation. In appendix C we show a general proof of concept implementation of the MonoidReduce operation, given user-provided functions `proj_fold`, `proj_fold_bwd`, `binary_reduce`.

3 Discussion

In general, activation recomputation reduces memory requirement at the cost of the extra compute needed for recomputation. This obviously holds for MonoidReduce as well. If the amount of memory saved is inconsequential or dwarfed by the compute cost of recomputation, there is little benefit of the approach. One notable example of this is the MLP-example: $Y = \sigma(XP^\intercal)Q$, $X : \mathbb{R}^{B\times M}, P : \mathbb{R}^{K\times M}, Q : \mathbb{R}^{K\times N}$, if $K \gg M$, the approach is viable, whereas if $K \approx M$, there is little to no benefit.

The hope is that with a general and *simple* framework that enables memory and cache-efficient layers, we might be able to iterate and innovate faster by making memory efficient layers by design, rather than by accident.

4 Future work

One interesting avenue for MonoidReduce is MLPs: Let X, P, Q be matrices of shape $B \times D$, $K \times D$, and $K \times D$, respectively, where B and K are very large, but D is small. Naive implementation would result in space complexity $BD + 2KD + BK$, with BK being the dominant factor. MonoidReduce can remove the BK-term, at a cost of extra FLOPs during recomputation. The total FLOPs (forward and backward) for the naive implemntation is $12BKD$, whereas MonoidReduce increases this to $14BKD$. i.e. $\approx 17\%$ increase in FLOPs. If we let $K = B = 16384$, and $D = 128$, this would result in a memory requirement of $\approx 2\%$ of the naive implementation, making MLPs for large batches and large hidden dimension, but small input and output dimension, much more feasible.

A Notation

Types & Terms $a : A$ denotes that the term a has type A. For example, $a : \mathbb{R}$ denotes that the term a is a real number.

Functions $A \to B$ denotes the type of functions from A to B. Lambda functions are written out as $f = x \mapsto x^2 + 3$, meaning $f(x) = x^2 + 3$.

$\to$ is interpreted in a right-associative manner, meaning $A \to B \to C = A \to (B \to C)$, which is isomorphic to $A \times B \to C$.

Product Types $A \times B$ denotes the product type of A and B. For brevity, terms of product types are occasionally written out as tuples. If $a : A$ and $b : B$, we can write $(a, b) : A \times B$, and vice versa. For a term $x : A \times B$, x_0 is the term corresponding to the left-hand type (A), and x_1 corresponds to the right hand-type. We also used named product types, where $\{a : A, b : B\}$ is the type of pairs of A and B indexed by the names a and b, i.e. for $x : \{a : A, b : B\}$ x_a is the A-part of the product, and x_b is the B-part of the product.

By $\frac{d\ A}{d\ B}$ we mean the linear function from A-typed gradients to B-typed gradients, i.e. the jacobian (if expressed as a matrix), or the vector jacobian product (if expressed as a function). $\cdot$ denotes function composition, e.g. the chain rule takes the form $\frac{d\ (f \cdot g)(x)}{d\ x} = \frac{d\ g(x)}{d\ x} \cdot \frac{d\ f(g(x))}{d\ g(x)}$.

B Elementwise MonoidReduce Attention

Listing 1: Attention example using elementwise MonoidReduce.

```
def D(c, a):
    return lambda g: {
        z: (a.z - c.z).exp() * (g.z + g.v @ (a.v - c.v))
        v: (a.z - c.z).exp() * g.v
    }

def forward(Q, K, V):
    """
    Q: M x F matrix
    K: N x F matrix
    V: N x D matrix
    returns:
    A: M-vector of T-values
    Parallelizes over M, iterates over N.
    """
    A = full(id) # M-vector of identity T-values

    for i in M (in parallel):
        q = Q[i]  # F-vector (query)
        # a: T-value aggregator
        # identity initialized
        a = id
        for j in N:
            k = K[j]
            v = V[j]
            # h: local T-value
            h = {z: q @ k, v: v}
            # add local T-value to a
            a = a o h
        A[i] = a
    return A

def backward(A, Q, K, V, gA):
    """
    A: M-vector of T-values
```

```
    gA: output (A) gradient
    returns: gQ, gK, gV
    Parallelizes over M and N.
    """
    gQ, gK, gV = [zeros_like(p) for p in [Q, K, V]]
    for i,j in (M x N) (in parallel):
        a = A[i]
        ga = gA[i]
        q = Q[i]
        k = K[j]
        v = V[j]
        h = {z: q @ k, v: v}
        gh = D(a, h)(ga)
        gQ[i].atomic_add(gh.z * k)
        gK[j].atomic_add(gh.z * q)
        gV[j].atomic_add(gh.v)
    return gQ, gK, gV
```

C Batched MonoidReduce

Using the terminology from corollary 2.1.3, $proj\_fold(A,B)_i = \bigodot_j f(A_i, B_j)$, and $proj\_fold\_bwd(A,B,P,gP)$ the backwards pass w.r.t. the final product, i.e. $(\frac{d\ P^{ij}}{d\ A^i \times B^j} \cdot D(P^i, P^{ij}))(g)$. binary reduce is the $\odot$ operator. Helper functions aslice and bslice are used to slices the P, A, and B-values into chunks.

Listing 2: Generalized, Batched, MonoidReduce example, implemented as torch function.

```
class MonoidReduce(torch.autograd.Function):
    @staticmethod
    def forward(A, B):
        P = id # T-vector of identity values

        A_slices = aslice(A)
        P_slices = aslice(P)
        B_slices = bslice(B)

        for a, p_out in zip(A_slices, P_slices) (in parallel):
            p = id # T-vector-slice of identity values.
            for b in B_slices:
                p = binary_reduce(p, proj_fold(a, b))
            p_out.store(p)
        return P

    @staticmethod
    @once_differentiable
    def backward(ctx, gP):
        gA = 0 # zero-initialized A-shaped gradients
        gB = 0 # zero-initialized B-shaped gradients

        A_slices = aslice(A)
        gA_slices = aslice(gA)
        P_slices = aslice(P)
```

```
gP_slices = aslice(gP)

B_slices = bslice(B)
gB_slices = aslice(gB)

for p, gp, a, ga, b, gb in product(
        zip(P_slices, gP_slices, A_slices, gA_slices),
        zip(B_slices, gB_slices)
        ): (in parallel)
    ga_part, gb_part = proj_fold_bwd(a, b, p, gP)
    ga.atomic_add(ga_part)
    gb.atomic_add(gb_part)
return gA, gB
```

D Gradients for CrossEntropy and MLP

For the examples in table 1 we also have that condition 1 holds, and by extension that theorem 2.1 and all its corollaries hold:

Monoid	Gradient
$T : \{p : \mathbb{R}, n : \mathbb{R}\}$ $a \odot b = \begin{Bmatrix} p: & \ln(e^{a_p} + e^{b_p}) \\ n: & a_n + b_n \end{Bmatrix}$ $= c$	$\frac{d\ c}{d\ a} = g \mapsto \begin{Bmatrix} p: & g_p e^{a_p - c_p} \\ n: & g_n \end{Bmatrix}$
CrossEntropy	
$T : \{q : \mathbb{R}, p : \mathbb{R}, n : \mathbb{R}\}$ $a \odot b = \begin{Bmatrix} q: & \ln(e^{a_q} + e^{b_q}) \\ p: & \ln(e^{a_p} + e^{b_p}) \\ n: & a_n e^{a_p - p} + b_n e^{b_p - p} \end{Bmatrix}$ $= c$	$\frac{d\ c}{d\ a} = g \mapsto \begin{Bmatrix} q: & g_q e^{a_q - c_q} \\ p: & (g_p + g_n(a_n - c_n)) e^{a_p - c_p} \\ n: & g_n e^{a_p - c_p} \end{Bmatrix}$
CrossEntropy between two distributions	
$T : \{v : \mathbb{R}^D\}$ $a \odot b = \{v : a_v + b_v\}$ $= c$	$\frac{d\ c}{d\ a} = g \mapsto g$
MLP	

References

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Re, C. (2022). Flashattention: Fast and memory-efficient exact attention with IO-awareness. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K., editors, *Advances in Neural Information Processing Systems*.

Hsu, P.-L., Dai, Y., Kothapalli, V., Song, Q., Tang, S., Zhu, S., Shimizu, S., Sahni, S., Ning, H., and Chen, Y. (2024). Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*.

Wijmans, E., Huval, B., Hertzberg, A., Koltun, V., and Krähenbühl, P. (2024). Cut your losses in large-vocabulary language models. *arXiv preprint arXiv:2411.09009*.