Commutative Semigroups and Efficient Neural Networks

Amaru Cuba Gyllensten

February 2023

1 Introduction

1.1 Notation

This paper employs liberal use of type and lambda notation borrowed from functional programming. A short primer of this notation is given below.

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.

Sum Types $A + B$ denotes the sum type of A and B. This is a discriminative sum, meaning that a term $x : A + B$ *knows* whether it is a left hand term A, or a right hand term B. A term of a sum type is written out $\amalg_i x$, where i indexes over the types in the sum. For example, terms of a sum type $A+B$ can either take the form $\amalg_0 x$, where $x : A$, or $\amalg_1 x$ where $x : B$.

Unit Type If used as a type, 1 denotes the unit type, i.e. the type that is inhabited by one term. This term is written out as $()$.

Integer Types If integers are used types, e.g. N, they denote the type that is inhabited by N terms. The type 2, for example, contains two terms, more commonly known as the Boolean type.

Exponent Types A type A^N where N is an integer denotes the type of N-tuples of A. In other words, $\mathbb{R}^2$ denotes the type of pairs of reals, or points in a plane. More generally, a term $v : A^B$ can be thought of as an array of As indexed by B (or equivalently, $B \to A$).

1.2 What are commutative semigroups?

A semigroup S is a tuple $(T, \circ)$, where T is the type, and $\circ : T \to T \to T$ is a binary operation. The binary operation must be associative, i.e. $a \circ (b \circ c) = (a \circ b) \circ c$.

Sum Summation over real numbers form a Semigroup:

$$\begin{aligned} T &= \mathbb{R} \\ x \circ y &= x + y \end{aligned}$$

Cat Concatenation over lists form a Semigroup:

$$\begin{aligned} T &= [a] \\ x \circ y &= x +\!\!+\, y \end{aligned}$$

Compose Composition over endo-functions form a Semigroup:

$$\begin{aligned} T &= A \to A \\ f \circ g &= x \mapsto f(g(x)) \end{aligned}$$

MMul Multiplication over matrices form a Semigroup.

$$\begin{aligned} T &= \mathbb{R}^{N \times N} \\ \mathbf{A} \circ \mathbf{B} &= \mathbf{AB} \end{aligned}$$

WSum Weighted sums of vectors in vector space $\mathcal{V}$ over field F form a Semigroup.

$$\begin{aligned} T &= \{v : \mathcal{V}, w : F\}) \\ x \circ y &= z \\ \text{where } z_w &= x_w + y_w \\ z_v &= x_v \frac{x_w}{z_w} + y_v \frac{y_w}{z_w} \end{aligned}$$

Max Given a type P with a total order, maximum (and minimum) forms a Semigroup.

$$\begin{aligned} T &= P \\ x \circ y &= \begin{cases} x & \text{if } x > y \\ y & \text{otherwise} \end{cases} \end{aligned}$$

Given a semigroup S, we can derive a function $fold$ that takes as input a nonempty sequence of Ts and returns the product over all elements in the sequence: $fold : [T] \to T$. The fact that $\circ$ is associative gives us the freedom to compute the products in arbitrary order as long as the order of elements remains the same: i.e. $fold([a,b,c,d]) = a \circ ((b \circ c) \circ d) = (a \circ b) \circ (c \circ d) = ...$

Commutative semigroups are semigroups where $\circ$ is commutative as well as associative, i.e. $a \circ b = b \circ a$. Out of the examples above, **Sum**, **WSum**, and **Max** are commutative semigroups. This give even more freedom with regards to the order of operations, as we no longer require the elements to be in their original order: $fold([a,b,c,d]) = a \circ ((b \circ c) \circ d) = (d \circ a) \circ (b \circ c) = ...$

For both of these cases, $fold$ing can be done without keeping the whole sequence in memory, and is highly parallelizable and distributable. This is far from a novel insight, and has been applied in for example MapReduce Lin (2013); Dean and Ghemawat (2008), but similar methods are just recently being applied to machine learning.

1.3 Commutative Semigroups and Machine Learning

The benefit of commutative semigroups in the context of machine learning stems from the fact that a fold over a sequence need not realize the whole sequence in memory, and that *if* a derivative $\frac{d\ x \circ y}{d\ x}$ can be expressed as a function of $x \circ y$ and x, then the same statement holds for a fold over a sequence, with the derivative of the fold w.r.t. each element being given by that same function. In essence, this implies that the sequence being folder over need not be realized during training, but can be accumulated in an online fashion and computed with arbitrary execution order, and that the backward pass can be highly parallellized as it only depends on the result and each individual input. This has recently been used in the context of self-attention Dao et al. (2022); Rabe and Staats (2021).

Theorem 1. *Given a commutative semigroup $(T, \circ)$ whose derivative $\frac{d\ x \circ y}{d\ x}$ is a function (D) of $x \circ y$ and x, i.e:*

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

We have that for any sequence $X : [T]$, the derivative of the fold over X and any element X_i is the same function D of $fold(X)$ and X_i:

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

Proof. Given a sequence X, we let $X_{\neg i}$ be the sequence X with the ith element removed.

$$\begin{aligned}
fold(X) &= X_i \circ fold(X_{\neg i}) && \text{By commutativity and associativity} \\
\frac{d\ fold(X)}{d\ X_i} &= \frac{d\ X_i \circ fold(X_{\neg i})}{d\ X_i} \\
&= D(X_i \circ fold(X_{\neg i}), X_i) && \text{By the precondition (eq. 1)} \\
&= D(fold(X), X_i)
\end{aligned}$$

□

1.4 Attention

To illustrate the usefulness of this in the case of self-attention, consider the case for a single attention vector $q : \mathbb{R}^V$ and value vectors $\mathbf{V} : \mathbb{R}^{V \times D}$. The output of self-attention for this particular query is $\text{softmax}(q) \cdot \mathbf{V} : \mathbb{R}^D$. We can reformulate this as a fold over a weighted sum where the weights are logarithmic, which is a commutative semigroup:

$$\begin{aligned}
T &= \{v : \mathbb{R}^D, w : \mathbb{R}\} \\
x \circ y &= z \\
&\quad \text{where} \\
&\quad z_w = \ln(\exp(x_w) + \exp(y_w)) \\
&\quad z_v = x_v \exp(x_w - z_w) + y_v \exp(y_w - z_w)
\end{aligned}$$

If we let $X_i = (\mathbf{V}_i, q_i)$, we have that $fold(X)_v = (X_0 \circ X_1 \circ ... \circ X_{D-1})_v = \text{softmax}(q) \cdot \mathbf{V}$. We also have that the derivative of $x \circ y$ w.r.t. x is a function of $x \circ y$ and x:

$$z = x \circ y \tag{2}$$

$$\frac{d\ z}{d\ x} = g \mapsto \begin{pmatrix} v : & g_v \exp(x_w - z_w) \\ w : & (g_w + g_v^\top (x_v - z_v)) \exp(x_w - z_w) \end{pmatrix} \tag{3}$$

Which by theorem 1 extends to $fold(X)$.

This readily extends to the full case with several attention vectors given by $\mathbf{Q} \cdot \mathbf{K}^\top$, for $\mathbf{Q} : \mathbb{R}^{Q \times D}, \mathbf{K} : \mathbb{R}^{V \times D}$:

$$\begin{aligned}
T &= \{v : \mathbb{R}^{Q \times D}, w : \mathbb{R}^Q\} \\
x \circ y &= z \\
&\quad \text{where} \\
&\quad z_{iw} = \ln(\exp(x_{iw}) + \exp(y_{iw})) \\
&\quad z_{iv} = x_{iv} \exp(x_{iw} - z_{iw}) + y_{iv} \exp(y_{iw} - z_{iw}) \\
\text{and } & X_j = (\mathbf{1V}_j, \mathbf{Q} \cdot \mathbf{K}_j)
\end{aligned}$$

$$
\begin{aligned}
z &= fold(X) \\
\frac{d\ z_v}{d\ \mathbf{V}_i} &= g_v \mapsto \mathbf{1}^\top g_v \exp(q_i - z_w) \\
\frac{d\ z_v}{d\ \mathbf{Q}_i} &= g_v \mapsto (g_w + g_v^\top(\mathbf{V}_i - z_v)) \exp(q_i - z_w) \\
\frac{d\ z_v}{d\ \mathbf{K}_i} &= g_v \mapsto (g_w + g_v^\top(\mathbf{V}_i - z_v)) \exp(q_i - z_w)
\end{aligned}
$$

This brings a couple of benefits:

- There is no need to keep all of q or $\mathbf{V}$ in memory during the forward pass as the result can be calculated in an online fashion[1].
- We are free to choose execution order, which gives considerably freedom in terms of compute-memory optimization.
- The method is highly parallelizable and memory efficient w.r.t. the backward pass, since the gradient for X_i only depends on the result $fold(X)$ and the individual value X_i.
- It is often possible to implement $fold$ in a more efficient, batched, manner. For example

  $$
  \begin{aligned}
  z &= fold(X) \\
  z_w &= \ln \sum_i (\exp(X_{iw})) \\
  z_v &= \sum_i \exp(X_{iw} - z_w) X_{iv}
  \end{aligned}
  $$

  This makes it possible use n-ary tree reduction strategies, rather than just binary tree reduction.

We believe that this extends to multiple avenues that are of interest in Machine Learning:

- A similar method can be applied to Cross-Entropy loss, which enables larger batches in contrastive learning.
- It applies to other aggregations, such as max-pooling, et.c.

[1]In self-attention, it should be noted that V, being reused across several queries, probably should be retained in memory for efficiency reasons.

$$\begin{tikzcd} F\,X \arrow[r, "F_p"] \arrow[d, "f"] & F\,T \arrow[d, "fold"] \\ Y & T \arrow[l, "q"] \end{tikzcd}$$

Figure 1: The general framework for efficient folding: For a structure of Xs (FX), map each X to T using some function $p : X \to T$, fold over the structure of Ts, and map the resulting T to Y using some function $q : T \to Y$, where $f = q \cdot fold \cdot F_p$.

- A variant of this (but not differentiable) also applies to sampling from a logit distribution:

$$\begin{aligned} T &= \mathcal{D}(A) \times \mathbb{R} \\ x \circ y &= z \\ &\quad \text{where} \\ &\quad z_w = \ln(\exp(x_w) + \exp(y_w)) \\ &\quad z_v = \begin{cases} x_v & \text{with probability } \exp(x_w - z_w) \\ y_v & \text{with probability } \exp(y_w - z_w) \end{cases} \end{aligned}$$

  where $\mathcal{D}(A)$ denotes the type of *samplers* of A, where samplers are considered equivalent if they sample according to the same probability distribution.

1.5 Cross-Entropy loss

The same method can be applied to cross-entropy loss. Given the following semigroup:

$$\begin{aligned} T &= \{w : \mathbb{R}, p : \mathbb{R}, n : \mathbb{R}\} \\ x \circ y &= z \\ &\quad \text{where} \\ &\quad z_w = \ln(\exp(x_w) + \exp(y_w)) \\ &\quad z_p = x_p \frac{z_w}{x_w} + y_p \frac{z_w}{y_w} \\ &\quad z_n = x_n + y_n \end{aligned}$$

We have the following:

$$
\begin{aligned}
q &: \mathbb{R}^C \\
p &: \mathbb{R}^C \\
X_i &= (q_i, p_i q_i, p_i q_i) \\
z &= fold(X) \\
z_p - z_n &= \mathrm{ce}(q, p)
\end{aligned}
$$

Where $\mathrm{ce}(q, p)$ is the cross entropy between a true distribution p and logits q over the set of classes C.

We also satisfy eq. 1 for theorem 1:

$$
\frac{d\ z}{d\ x} = g \mapsto \begin{pmatrix} w: & g_w \exp(x_w - z_w) + g_p \left(\frac{z_p}{z_w} \exp(x_w - z_w) - z_w \frac{x_p}{x_w^2} \right) \\ p: & g_p \frac{z_w}{x_w} \\ n: & g_n \end{pmatrix}
$$

$$
\frac{d\ z_p - z_n}{d\ x} = g \mapsto \begin{pmatrix} w: & g \left(\frac{z_p}{z_w} \exp(x_w - z_w) - z_w \frac{x_p}{x_w^2} \right) \\ p: & g \frac{z_w}{x_w} \\ n: & -g \end{pmatrix}
$$

In the case when the distribution p is one-hot, this simplifies to the following:

$$
\begin{aligned}
T &= \{w : \mathbb{R}, n : \mathbb{R}\} \\
x \circ y &= z \\
&\quad \text{where} \\
&\quad z_w = \ln(\exp(x_w) + \exp(y_w)) \\
&\quad z_n = x_n + y_n
\end{aligned}
$$

$$
\begin{aligned}
q &: \mathbb{R}^C \\
j &: C \\
X_i &= (q_i, q_i \delta_{ij}) \\
z &= fold(X) \\
z_w - z_n &= \mathrm{ce}(q, j)
\end{aligned}
$$

$$
\begin{aligned}
\frac{d\ z}{d\ x} &= g \mapsto \begin{pmatrix} w: & g_w \exp(x_w - z_w) \\ n: & g_n \end{pmatrix} \\
\frac{d\ z_w - z_n}{d\ q_i} &= g \mapsto g(\exp(q_i - z_w) - \delta_{ij})
\end{aligned}
$$

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*.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Lin, J. (2013). Monoidify! monoids as a design principle for efficient mapreduce algorithms. *arXiv preprint arXiv:1304.7544*.

Rabe, M. N. and Staats, C. (2021). Self-attention does not need o(n^2) memory. *CoRR*, abs/2112.05682.