Motivating Multivariable Calculus with Machine Learning

By Collin Drake

The Discrete Default & The Criticality of Commutativity

For discrete sequences, integration is merely a summation, e.g., $\int \mathcal{D} = \sum \mathcal{D}$ for the discrete sequence $\mathcal{D}$. For all of machine learning's myriad practical cases, its sequences are discrete. Such sequences are most frequently represented as vectors, matrices, or tensors, all of which are flattened and represented in memory as vectors.

The most notable and familiar examples of commutative operations are sums and products. Within machine learning, there also exist operators which are not strictly mathematically equal to another operator, although they do yield nearly identical results. Examples of such operators would be maximums and minimums, e.g., $\max \neq \sum$, but $\Omega_{\max} \approx \Omega_{\Sigma}$ where Ω denotes the model. Integrals, specifically multivariate ones, aren't commutative whereas summations are. This distinction becomes crucial when putting theory into practice. Take the trapezoidal rule for approximating an integral $\int_a^b f(x)\,dx$ for example, which is given by:

$$\begin{aligned} \lozenge_n \int_a^b f(x)\;dx \;&= \frac{\Delta x}{2}\left[f(x_0) + \left(2\sum_{i=1}^{n-1} f(x_i)\right) + f(x_n)\right] \\ &= \frac{\Delta x}{2}\left[f(x_0) + 2f(x_1)\ \ldots\ 2f(x_{n-1}) + f(x_n)\right]. \end{aligned}$$

Since integrals aren't commutative, they cannot be utilized within the loss functions of regressive models like transformers. However, to those of you who've noticed that the above "integral" is, in fact, commutative: bravo. If the above were multivariate, its commutativity would no longer hold. Since that is almost always the case due to machine learning's data being of a high dimensionality, we assume that integrals are always non-commutative. For example, a simple regressive model for forecasting temperature, wind speed, humidity, air pressure, UV, and precipitation across 15 minutes would have a dimensionality of $6 \cdot 15 = 90$, e.g., $\Omega \in \mathbb{R}^{(1,6,15)}$
.

Consider Mean Squared Error (MSE) instead, which is given by:

$$\mathrm{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2.$$

Much like an integral, it measures the distance, or rather the area, between the target sequence $\mathbf{y}$ and the predicted sequence $\hat{\mathbf{y}}$. Unlike an integral, however, it is commutative, and that is easy to both observe and prove. Then, we have Max Squared Error (MaxSE), which is given by:

$$\mathrm{MaxSE}(\mathbf{y}, \hat{\mathbf{y}}) = \max_{i=1}^{n} \left((\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \right),$$

it is equally obvious that it is, indeed, commutative, but more difficult to prove since $\max$ is defined algorithmically instead of mathematically. Such overlap frequently arises within machine learning, and it is therefore not always productive to reconcile the theoretical and the practical.

The criticality of commutativity can, again, be proven in utilizing Wasserstein distance within a loss function, as transformers' optimizers simply cannot descend the gradient of a latent space built by non-commutative operations. Wasserstein distance is given by:

$$W(P, Q) = \inf_{\gamma \in \Gamma(P,Q)} \iint c(x, y)\, d\gamma(x, y),$$

where P and Q are probability distributions, $c(\cdot)$ the defined cost function (often euclidean), and $\gamma(\cdot)$ the joint distribution of P and Q as a function of the dimensions of Γ. As a refresher, Wasserstein distance is defined as the minimal cost of transforming one probability distribution into another, where cost is measured as the amount of probability mass moved multiplied by the distance it is moved.

Although, not all algorithms require such commutativity. Convolution, a cornerstone technique within computer vision, is by definition non-commutative; excepting graph convolution, that is, which merely consists of summations. Such requirements must, therefore, be evaluated and considered on a case-by-case basis. On that note, graph convolution nicely segues us to our next topic, seeing as it is the argmax of a graph convolution.

Embedding Selection Algorithm

The following is a selection algorithm for embedding vectors based on their importance within their parent sequence. The importance is derived by first summating the attention heads, second summating the rows of the resultant attention matrix, and third taking the argmax of the resultant vector. This can be computed as:

$$\underset{j \in \{1...m\}}{\operatorname{argmax}} \sum_{i=1}^{n} \left(\mathbf{A} \sum_{\mathbf{H} \in \mathcal{H}} \mathbf{H} \right)_{ij}.$$

This approach addresses the limitation of transformers in maintaining state across prompts. To supplement this lack of context, a weak Retrieval-Augmented Generation (RAG) mechanism is introduced. This mechanism selects the k most "important" indices and their corresponding tokens, where importance is determined by the summation and argmax of attention scores.

We define the adjacency matrix $\mathbf{A}$ as:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \Bigg| \quad \mathbf{A}_{ij} = 0 \text{ if } i = j$$

The aggregated attention matrix $\mathbf{B}$ is computed as:

$$\mathbf{B} = \mathbf{A} \sum_{\mathbf{H} \in \mathcal{H}} \mathbf{H},$$

where $\mathbf{B}$ denotes the summation of output matrices across the attention heads, multiplied by the adjacency matrix $\mathbf{A}$. We multiply our sum by the adjacency matrix so that the attention scores from tokens to themselves are not considered. An example of the resultant matrix $\mathbf{B}$ is:

$$\mathbf{B} = \begin{bmatrix} 0 & 8.41 & 3.29 \\ 9.2 & 0 & 1.6 \\ 5.1 & 3.8 & 0 \end{bmatrix}.$$

Next, the vector $\mathbf{b}$ is obtained by summing across rows:

$$\mathbf{b} = \left[\sum_{i=1}^{n} \mathbf{B}_{ij} \right] \forall j \in \{1 \ldots m\}.$$

For the example matrix $\mathbf{B}$, we get:

$$\mathbf{b} = [14.3 \quad 12.21 \quad 4.89].$$

The top-k indices of $\mathbf{b}$ are selected using the argmax function:

$$\underset{j \in \{1 \ldots m\}}{\text{argmax}} \sum_{i=1}^{n} \left(\mathbf{A} \sum_{\mathbf{H} \in \mathcal{H}} \mathbf{H} \right)_{ij}.$$

For $k = 1$, the index $i = 0$ would be selected, corresponding to the embedding $\mathbf{x}_0$ of $\mathbf{x}$.

Weak RAG

This is the truly difficult portion, as building a vector database that allows for both fast mutation and lookup is a challenge indeed. As its stands, Hierarchical Navigable Small Worlds (HNSW) and Locality Sensitive Hashing (LSH) would be the preeminent algorithms for such. As it stands HSNW posses the accuracy to be useful in such a way, but its speed is entirely reliant upon the stability of its underlying graph. Adding and removing vectors in quick succession would therefore be highly inefficient. We turn our gaze, then, to LSH, a highly efficient, highly compressed vector lookup and storage paradigm. LSH is better candidate because it does not rely upon any sort of graph structure to achieve excellent search times.