Anti-derivative setup

December 19, 2021

1 GPU Uniform UMAP

We currently perform Uniform UMAP on the cpu. However, it is naturally extendable to the gpu environment. Specifically, it is performing large element-wise operations and applying gradient descent to minimize error. Thus, we can employ automatic differentiation engines to quickly perform the search.

1.1 Gradient

Recall our loss function:

$$\mathcal{C} = \sum_{j \neq i} p_{ij} \log \left(\frac{p_{ij}}{q_{ij}} \right)$$

where $p_{ij} = \exp\left(||x_i - x_j||^2/\sigma_i\right)$ is the nearest-neighbor likelihoods in the high-dimensional space and $q_{ij} = (1 + ||y_i - y_j||^2)^{-1}/Z$ is the nearest-neighbor likelihoods in low-dimensional space. We define the normalization term Z by $Z = \sum_{k \neq l}(1 + ||y_k - y_l||^2)^{-1}$

Then our derivative with respect to the low-dimensional point y_i can be represented as

$$\frac{\partial C}{\partial y_i} = \underbrace{4 \sum_{j \neq i} p_{ij} q_{ij} Z (y_i - y_j)}_{F_{attr}} - \underbrace{4 \sum_{j \neq i} q_{ij}^2 Z (y_i - y_j)}_{F_{rep}}$$

Given the above setup, we begin gradient descent with a list $\mathcal{E}$ of nearest neighbors in high-dimensional space, where each $\mathcal{E}_{ij}$ represents an edge in the high-dimensional k-nearest neighbors graph. We then collect gradients by calculating the $|\mathcal{E}|$ attractive forces – one for along each nearest neighbor edge. We similarly calculate the $|\mathcal{E}|$ repulsive forces between one of the points from the attractive force and a random point y_k. To obtain UMAP, we collect the attractive forces on both y_i and y_j but the repulsive force only on y_i. To obtain tSNE, we only collect the attractive and repulsive forces to y_i and ignore y_j. Once we've summed all of the corresponding forces, we apply them uniformly with momentum gradient descent across Y.

This approach combines the principles of UMAP and tSNE to use UMAP's fast optimization method with tSNE's normalization and gradient schema. Thus, we minimize the number of computations by only calculating the relevant attractive forces and countering them by an equal amount of repulsive forces.

1.2 GPU implementation

The above gradient descent method is difficult to implement with auto-differentiation libraries such as Tensorflow and Pytorch. Indeed, if we calculate the loss on the nearest neighbors and random points, its gradient will contain a superfluous repulsive force for the nearest neighbors and a correspondingly superfluous attractive force for the random pairs of points. Although these are theoretically valid, they upset the balance obtained by the previously described optimization method and create convergence issues.

To remedy this, we seek the following loss functions through antiderivatives:

$$\mathcal{C}_{attr}(y_i) = \int F_{attr} dy_i = \int p_{ij} q_{ij} Z(y_i - y_j) dy_i$$
$$\mathcal{C}_{rep}(y_i) = \int F_{rep} dy_i = \int q_{ij}^2 Z(y_i - y_j) dy_i$$

In this way, the gradient of $\mathcal{C}_{attr}$ with respect to the nearest neighbor edges would give us only the attractive force between the nearest neighbors as in the original Uniform UMAP implementation. The same logic applies for the repulsive forces across random points.