Implementation of Projective Pruning

Algorithm Overview. Our approach determines the unit to prune by minimizing the projection distance, formulated as:

$$\arg\min_i \|\boldsymbol{W}_{-i}^T \left(\boldsymbol{W}_{-i}\boldsymbol{W}_{-i}^T + \lambda\mathbb{I}\right)^{-1} \boldsymbol{W}_{-i}\boldsymbol{w}_i - \boldsymbol{w}_i\|_2. \tag{1}$$

The unpruned weights are updated using the projection of the pruned row

$$\boldsymbol{q}_i = \left(\boldsymbol{W}_{-i}\boldsymbol{W}_{-i}^T + \lambda\mathbb{I}\right)^{-1} \boldsymbol{W}_{-i}\boldsymbol{w}_i, \tag{2}$$

with the following redistribution scheme:

$$\forall j \neq i: \quad \boldsymbol{w}_j \leftarrow \boldsymbol{w}_j \cdot (1 + \alpha\boldsymbol{q}_j), \tag{3}$$
$$b_j \leftarrow b_j \cdot (1 + \beta\boldsymbol{q}_j), \tag{4}$$
$$\boldsymbol{v}_j \leftarrow \boldsymbol{v}_j \cdot (1 + \gamma\boldsymbol{q}_j). \tag{5}$$

At each step, we compute the projection distance for each row, prune the one with the smallest distance, and update the remaining weights. This is repeated until the target sparsity is reached.

Lazy Updates with Coefficient Tracking

The redistribution is linear, therefore instead of recalculating projections at each pruning step, we can lazily update the weights indirectly by tracking their coefficients.

Coefficient Tracking. We maintain sequences of update vectors $(\boldsymbol{u}_0, \ldots, \boldsymbol{u}_r)$, initialized as $\boldsymbol{u}_0 = (1, \ldots, 1) \in \mathbb{R}^{n-t}$, where $n - t$ is the number of remaining neurons. The update rule is:

$$\boldsymbol{u}_{t+1} \leftarrow \boldsymbol{u}_t \odot \left((1 + \phi) \cdot \boldsymbol{q}_i^{(t)}\right). \tag{6}$$

Here, $\odot$ denotes element-wise multiplication, $\boldsymbol{q}_i^{(t)}$ is the projection vector for the pruned row at step t, and ϕ is a placeholder for α, β, or γ. We maintain separate sequences $\boldsymbol{u}^{(\alpha)}$, $\boldsymbol{u}^{(\beta)}$, and $\boldsymbol{u}^{(\gamma)}$ for each coefficient.

Coefficient Adjustment. To ensure the validity of projection coefficients for the next pruning step, we adjust the remaining rows' coefficients:

$$\forall j \neq i: \quad \boldsymbol{q}_j^{(t+1)} \leftarrow \boldsymbol{q}_j^{(t)} \odot \frac{1}{(1 + \alpha) \cdot \boldsymbol{q}_i^{(t)}} \tag{7}$$

Applying Coefficient. After all pruning steps, we update the weights using the final coefficient vectors:

$$\forall j \neq i: \quad \boldsymbol{w}_j' \leftarrow \boldsymbol{w}_j \cdot (\boldsymbol{u}_r^{(\alpha)})_j, \tag{8}$$
$$\boldsymbol{b}_j' \leftarrow \boldsymbol{b}_j \cdot (\boldsymbol{u}_r^{(\beta)})_j, \tag{9}$$
$$\boldsymbol{v}_j' \leftarrow \boldsymbol{v}_j \cdot (\boldsymbol{u}_r^{(\gamma)})_j, \tag{10}$$

and keep only the unpruned rows.

Masking Weights

To avoid costly memory reallocation everytime a row is removed, we use a binary mask to track pruned and unpruned rows in $\boldsymbol{W}$. Since the coefficients don't require contiguous storage, this allows for in-place coefficient updates and efficient access using views.

Fast Matrix Inverses

Even with the lazy updates, we still need to compute all the matrix inverses for the initial pruning step.

Instead of computing $\mathbf{G}_{-ii}^{-1} := (\boldsymbol{W}_{-i}\boldsymbol{W}_{-i}^T + \lambda\mathbb{I}_{n-1})^{-1}$ directly for each of n rows, we compute $\mathbf{G}^{-1} := (\boldsymbol{W}\boldsymbol{W}^T + \lambda\mathbb{I}_n)^{-1}$ once and derive the other inverses.

Derivation[1] of $\mathbf{G}_{-ii}^{-1}$

$\mathbf{G}_{-ii}$ is a rank-1 update of $\mathbf{G}$: $\mathbf{G}_{-ii} = (\mathbf{G}_{\text{row}\neq i,\text{col}\neq i})$. To move the i-th element to the last row and column, we permute it and rename its partitions:

$$\mathbb{P}_{i,n}\mathbf{G}\mathbb{P}_{i,n} = \begin{pmatrix} \mathbf{G}_{-ii} & \mathbf{G}_i^T \\ \mathbf{G}_i & \mathbf{G}_{ii} \end{pmatrix} = \begin{pmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^T & \mathbf{D} \end{pmatrix}, \tag{11}$$

where $\mathbf{A} \in \mathbb{R}^{n-1,n-1}, \mathbf{C} \in \mathbb{R}^{n-1}, \mathbf{D} \in \mathbb{R}$. Similarly, we permute $\mathbf{G}^{-1}$:

$$\mathbb{P}_{i,n}\mathbf{G}^{-1}\mathbb{P}_{i,n} = \begin{pmatrix} \mathbf{a} & \mathbf{c} \\ \mathbf{c}^T & \mathbf{d} \end{pmatrix}, \tag{12}$$

where $\mathbf{a} \in \mathbb{R}^{n-1,n-1}, \mathbf{c} \in \mathbb{R}^{n-1}, \mathbf{d} \in \mathbb{R}$.

Using the identity $\mathbb{I}_n = (\mathbb{P}_{i,n}\mathbf{G}\mathbb{P}_{i,n})\left(\mathbb{P}_{i,n}\mathbf{G}^{-1}\mathbb{P}_{i,n}\right)$ and (11), (12), we get:

$$\begin{pmatrix} \mathbb{I}_{n-1} & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{Aa} + \mathbf{Cc}^T & \mathbf{Ac} + \mathbf{Cd} \\ \mathbf{C}^T\mathbf{a} + \mathbf{Dc}^T & \mathbf{C}^T\mathbf{c} + \mathbf{Dd} \end{pmatrix} \tag{13}$$

From the upper blocks $\mathbf{Ac} + \mathbf{Cd} = 0$ and $\mathbf{Aa} + \mathbf{Cc}^T = \mathbb{I}_{n-1}$:

$$\mathbf{Cc}^T = \mathbf{C}(\mathbf{dd}^{-1})\mathbf{c}^T = (\mathbf{Cd})\mathbf{d}^{-1}\mathbf{c}^T = (-\mathbf{Ac})\mathbf{d}^{-1}\mathbf{c}^T \tag{14}$$

$$\mathbf{Aa} + \mathbf{Cc}^T = \mathbf{Aa} - \mathbf{Acd}^{-1}\mathbf{c}^T = \mathbf{A}(\mathbf{a} - \mathbf{cd}^{-1}\mathbf{c}^T) = \mathbb{I}_{n-1} \tag{15}$$

where (14) is valid because $\mathbf{d} \in \mathbb{R}, \mathbf{d} \geq \lambda^2$. Since $\mathbf{G}_{-ii}^{-1} = \mathbf{A}^{-1}$, we derive:

$$\mathbf{G}_{-ii}^{-1} = \mathbf{a} - \frac{1}{\mathbf{d}}\mathbf{cc}^T, \tag{16}$$

which allows to efficiently compute $\mathbf{G}_{-ii}^{-1}$ from $\mathbf{G}^{-1}$.

[1]whuber. Cross Validated, 2020. `https://stats.stackexchange.com/q/450186`