Unstructured Pruning Methods

Nikita Breskanu
nbreskanu73@gmail.com

Abstract

As Large Language Models scale, compression methods have become essential for reducing memory footprints and computational costs. Among these, unstructured pruning seeks to mask a fraction of model weights to achieve sparsity without specific structural constraints. This document provides a personal mathematical deep-dive into these techniques. I first cover one-shot methods including Optimal Brain Surgeon, SparseGPT, and Wanda, followed by training-based approaches such as Movement Pruning and oBERT. To my knowledge, this is a unique synthesis that provides both rigorous mathematical derivations and explicit connections between these disparate frameworks. Finally, a comparative study using BERT on the CoNLL-2003 benchmark analyzes the trade-offs between pruning latency, sample efficiency, and model quality.

1 Optimal Brain Surgeon (OBS)

1.1 Per-row update

OBS tries to solve the following optimization task:

$$L := \mathbb{E}_{x\sim\mathbb{X}}\|(W - M \odot W')x\|^2 \approx \|(W - M \odot W'))X\|^2 \to \min_{W',M}$$

where M is binary mask, and W' are new weights.

However, this task is not solvable directly (unfortunately). To be more exact, given M, optimization by W' can be done analytically. But the opposite isn't true.

Let's try to solve simpler task:

$$\|(W - M \odot W')X\|^2 \to \min_{W'}$$

We can separate by rows:

$$\sum_i \|(w_i - m_i \odot w_i')^T X\|^2 \to \min$$

And consider each row independently (index i is omited)

$$\|(w - m \odot w')^T X\|^2 \to \min$$

We can rewrite the task:

$$\|(w-m\odot w')^T X\|^2 = \|X^T(w-m\odot w')\|^2 = \langle X^T(w-m\odot w'), X^T(w-m\odot w')\rangle = \langle H(w-m\odot w'), w-m\odot w'\rangle$$

where $H := XX^T$

So, we get:

$$\langle H(w - m \odot w'), w - m \odot w'\rangle \to \min_{w',\|m\|_0=k} \tag{1}$$

Let's divide the row depending on mask as:

$w = (w_P, w_R)$ and $w' = (w'_P, w'_R)$, where w_P are weights with pruned indices (where mask m have zeroes), and w_R are remaining weights. By design, $w'_P = 0$.

Also, let's split the matrix H accordingly:

$$H = \begin{bmatrix} H_{PP} & H_{PR} \\ H_{RP} & H_{RR} \end{bmatrix}$$

Now, we can rewrite the objective (solve by $\Delta w := w' - w$):

$$L = \langle \begin{bmatrix} H_{PP} & H_{PR} \\ H_{RP} & H_{RR} \end{bmatrix} \begin{bmatrix} \Delta w_P \\ \Delta w_R \end{bmatrix}, \begin{bmatrix} \Delta w_P \\ \Delta w_R \end{bmatrix} \rangle = \langle \begin{bmatrix} H_{PP}\Delta w_P + H_{PR}\Delta w_R \\ H_{RP}\Delta w_P + H_{RR}\Delta w_R \end{bmatrix}, \begin{bmatrix} \Delta w_P \\ \Delta w_R \end{bmatrix} \rangle$$

$$L = \langle H_{PR}\Delta w_R, \Delta w_P \rangle + \langle H_{RP}\Delta w_P, \Delta w_R \rangle + \langle H_{RR}\Delta w_R, \Delta w_R \rangle + \text{const}$$

Now, we can differentiate by Δw_R ($H_{PR}^T = H_{RP}$ and $H_{RR}^T = H_{RR}$):

$$\nabla L_{\Delta w_R} = 2H_{RR}\Delta w_R + 2H_{RP}\Delta w_P = 0 \implies \Delta w_R = -H_{RR}^{-1}H_{RP}\Delta w_P$$

Now, substituting back in $\Delta w_P = -w_P$, we obtain:

$$w'_R = w_R + H_{RR}^{-1}H_{RP}w_P$$

Although this is already good, but inverting subblock H_{RR} is computationally complex. So, we can go further.

Let's remember Gaussian Elimination identity [11] (I wrote only the most important part):

$$H^{-1} = \begin{bmatrix} H_{PP} & H_{PR} \\ H_{RP} & H_{RR} \end{bmatrix}^{-1} = \begin{bmatrix} (H_{PP} - H_{PR}H_{RR}^{-1}H_{RP})^{-1} & \cdot \\ -H_{RR}^{-1}H_{RP}(H_{PP} - H_{PR}H_{RR}^{-1}H_{RP})^{-1} & \cdot \end{bmatrix} = \begin{bmatrix} H_{P/R}^{-1} & \cdot \\ -H_{RR}^{-1}H_{RP}H_{P/R}^{-1} & \cdot \end{bmatrix}$$

where $H_{P/R} = H_{PP} - H_{PR}H_{RR}^{-1}H_{RP}$ is the Schur complement.

This way, we can actually see that:

$$H_{RR}^{-1}H_{RP} = H_{RR}^{-1}H_{RP}H_{P/R}^{-1}H_{P/R} = -(H^{-1})_{RP}((H^{-1})_{PP})^{-}1$$

By substituting that to the solution, we get the final form of OBS update:

$$\boxed{w'_R = w_R - (H^{-1})_{RP})((H^{-1})_{PP})^{-1}w_p} \tag{2}$$

In this formula, we need to invert only the H itlself, and PP subblock of it.

If we have only a single pruned column $P = \{j\}$, then the update simplifies nicely:

$$w'_R = w_R - (H^{-1})_{Rj}\frac{w_j}{(H^{-1})_{jj}}$$

This can be extended to the whole column, because $0 = w'_j = w_j - (H^{-1})_{jj}\frac{w_j}{(H^{-1})_{jj}}$:

$$w' = w - (H^{-1})e_j\frac{w_j}{(H^{-1})_{jj}} \tag{3}$$

where e_j is the basis vector of column j. That is the exact update for pruning a single column from OBS paper [4].

1.2 Approximate selection of mask M

Okay, now comes the hard part: optimizing w.r.t. mask M.

$$L = \langle H_{PP}\Delta w_P, \Delta w_P\rangle + \langle H_{PR}\Delta w_R, \Delta w_P\rangle + \langle H_{RP}\Delta w_P, \Delta w_R\rangle + \langle H_{RR}\Delta w_R, \Delta w_R\rangle$$

Let's substitute our previously updated solution (intermediate result) in the loss:

$$\Delta w'_R = -H_{RR}^{-1}H_{RP}\Delta w_P$$

$$\begin{aligned} L = \langle H_{PP}\Delta w_P, \Delta w_P\rangle + \langle H_{PR}(-H_{RR}^{-1}H_{RP}\Delta w_P), \Delta w_P\rangle + \langle H_{RP}\Delta w_P, -H_{RR}^{-1}H_{RP}\Delta w_P\rangle + \\ +\langle H_{RR}H_{RR}^{-1}H_{RP}\Delta w_P, H_{RR}^{-1}H_{RP}\Delta w_P\rangle = \\ = \langle H_{PP}\Delta w_P, \Delta w_P\rangle - \langle H_{PR}H_{RR}^{-1}H_{RP}\Delta w_P, \Delta w_P\rangle = \\ = \langle (H_{PP} - H_{PR}H_{RR}^{-1}H_{RP})\Delta w_P, \Delta w_P\rangle \end{aligned}$$

We know that $\Delta w_P = -w_P$, so we get:

$$L = \langle (H_{PP} - H_{PR}H_{RR}^{-1}H_{RP})w_P, w_P\rangle \to \min_{|P|=k}$$

Again we use the Schur matrix identity $(H^{-1})_{PP} = H_{P/R}^{-1} = (H_{PP} - H_{PR}H_{RR}^{-1}H_{RP})^{-1}$ and we get:

$$L = \langle ((H^{-1})_{PP})^{-1} \times w_P, w_P\rangle \to \min_{|P|=k} \tag{4}$$

This is the simplest form of the final object. The task is to select the k columns $P = \{j_1, j_2, \ldots, j_k\}$, such that the loss is minimized.

Unfortunately, this is an NP-hard problem. But, authors of OBS method try to find approximate solution.

Let's remember that $H = XX^T$ is the estimation of the feature covariance matrix. We can make an assumption that all features are independent (or at least close to being independent), then the matrix becomes diagonal. Under that heavy assumption, the loss simplifies drastically:

$$L = \sum_j ((H^{-1})_{jj})^{-1} \times w_j^2 = \sum_j \frac{w_j^2}{H_{jj}^{-1}}$$

Now, the optimization problem becomes very easy. The solution is simply selecting the k columns with the lowest ratio:

$$\boxed{P = \text{TopK}_{\text{smallest}}\left(\frac{w_j^2}{H_{jj}^{-1}}\right)} \tag{5}$$

2 SparseGPT

2.1 Incremental update

One of the problems with OBS update is that it has to be done for each row sequentially, and for each row different matrix H_{PP}^{-1} needs to be inverted. That is computationally heavy for large matrices, as in heavy-parameterized LLMs like Llama-7B.

SparseGPT uses very clever math to apply incremental updates across all rows jointly, and approximates exact OBS updates.

Lemma 1. *Let P be a fixed set of pruned indices, and $S_0 = \emptyset, S_1 = \{j_1\}, S_2 = \{j_1, j_2\}, \ldots, S_k = P$ be the sequence of P subsets. Consider the following incremental update procedure:*

$$w_{R_0}^{(S_0)} = w$$

$$w_{R_n}^{(S_{n+1})} = w_{R_n}^{(S_n \cup \{j_{n+1}\})} = w_{R_n}^{(S_n)} - \frac{w_{j_{n+1}}^{(S_n)}}{(H_{/S_n}^{-1})_{j_{n+1}j_{n+1}}} (H_{/S_n}^{-1})_{:,j_{n+1}} \tag{6}$$

Where S_n is the previous subset of P pruned indices, and j_{n+1} is the newly appended pruned index, $R_n := P \setminus S_n$; Here $H_{/S_n}^{-1}$ is the Schur complement for matrix H^{-1}:

$$H_{/S_n}^{-1} := (H^{-1})_{R_n/S_n} = (H^{-1})_{R_nR_n} - (H^{-1})_{R_nS_n}((H^{1})_{S_nS_n})^{-1}(H^{-1})_{S_nR_n} \in \mathbb{R}^{R_n \times R_n}$$

$$H_{/S_0}^{-1} = H^{-1}$$

Then, for each $S_n \subset P$, $w^{(S_n)}$ is the exact solution to the OBS minimization task, which means:

$$w_{R_n}^{(S_n)} = w_{R_n} - (H^{-1})_{R_nS_n})((H^{-1})_{S_nS_n})^{-1} w_{S_n}; \quad w_{S_n}^{(S_n)} = 0$$

Proof. Let's perform induction on n.

Basis step:

Trivially, for $n = 1$ and a single j_1 the incremental update 6 gives:

$$w_{R_0}^{(S_1)} = w_{R_0}^{(S_0)} - \frac{w_{j_1}^{(S_0)}}{(H_{/S_0}^{-1})_{j_1j_1}} (H_{/S_0}^{-1})_{:,j_1}$$

Here, we can simplify, because $w_{R_0}^{(S_0)} = w$ and Schur complement is H^{-1}:

$$w^{S_1} = w - \frac{w_{j_1}}{(H^{-1})_{j_1j_1}} (H^{-1})_{:,j_1}$$

Thus, we get exactly (Eq. 3) update that is a solution for single index pruning.

Inductive step:

Here we will prove inductive step for S_{n+1} when inductive hypothesis is true for S_n. For clarity, let's slightly abuse notation and write: $j := j_{n+1}, S := S_n, R := R_n, M := R_{n+1} = R_n \setminus \{j\}$. Let's also denote: $A := H^{-1}, D := (A_{SS})^{-1}$.

Suppose the lemma already holds for $w^{(S_n)}$:

$$w_R^S = w_R - A_{RS} D w_S; \quad w_S^{(S)} = 0$$

We have the incremental update:

$$w_R^{(S \cup \{j\})} = w_R^S - \frac{w_j^S}{(H_{/S}^{-1})_{jj}} (H_{/S}^{-1})_{:,j}$$

We need to prove:

$$w_M^{S \cup \{j\}} = w_M - A_{M(S \cup \{j\})}(A_{(S \cup \{j\})(S \cup \{j\})})^{-1} w_{(S \cup \{j\})}; \quad w_{S \cup \{j\}}^{(S \cup \{j\})} = 0$$

First, let's notice that we update only the remaining indices, and obviously, $w_j^{(S \cup \{j\})} = 0$. Thus, all the pruned weights are zero, so we need to prove only the left side.

We know that:

$$H_{/S}^{-1} = A_{RR} - A_{RS} D A_{SR}$$

Let's substitute that into the incremental update:

$$w_R^{(S \cup \{j\})} = w_R^S - \frac{w_j^S}{A_{jj} - A_{jS} D A_{Sj}} (A_{Rj} - A_{RS} D A_{Sj})$$

Then, substitute w_R^S and w_j^S from the induction hypothethis:

$$w_R^{(S\cup\{j\})} = w_R - A_{RS}Dw_S - \frac{w_j - A_{jS}Dw_S}{A_{jj} - A_{jS}DA_{Sj}}(A_{Rj} - A_{RS}DA_{Sj})$$

For clarity, let's denote denominator as $\alpha := A_{jj} - A_{jS}DA_{Sj}$:

$$w_R^{(S\cup\{j\})} = w_R - A_{RS}Dw_S - \frac{w_j}{\alpha}A_{Rj} + \frac{w_j}{\alpha}A_{RS}DA_{Sj} + \frac{1}{\alpha}A_{Rj}A_{jS}Dw_S - \frac{1}{\alpha}A_{RS}DA_{Sj}A_{jS}Dw_S$$

Let's factor out by A_{RS} and A_{Rj}:

$$w_R^{(S\cup\{j\})} = w_R - A_{Rj}\left(\frac{w_j}{\alpha} - \frac{1}{\alpha}A_{jS}Dw_S\right) - A_{RS}\left(Dw_s - \frac{1}{\alpha}DA_{Sj}(w_j) + \frac{1}{\alpha}DA_{Sj}A_{jS}Dw_S\right)$$

Now we can introduce matrix notation:

$$w_R^{(S\cup\{j\})} = w_R - \begin{bmatrix}A_{Rj} & A_{RS}\end{bmatrix}\begin{bmatrix}\frac{w_j}{\alpha} - \frac{A_{jS}Dw_S}{\alpha} \\ -\frac{DAS_jw_j}{\alpha} + (D + \frac{1}{\alpha}DA_{Sj}A_{jS}D)w_s\end{bmatrix}$$

Next, we introduce another matrix which is multiplied by $\begin{bmatrix}w_j \\ w_S\end{bmatrix}$:

$$w_R^{(S\cup\{j\})} = w_R - \begin{bmatrix}A_{Rj} & A_{RS}\end{bmatrix}\begin{bmatrix}\frac{1}{\alpha} & -\frac{1}{\alpha}A_{jS}D \\ -\frac{1}{\alpha}DAS_j & D + \frac{1}{\alpha}DA_{Sj}A_{jS}D\end{bmatrix}\begin{bmatrix}w_j \\ w_S\end{bmatrix}$$

Now, if we remember that $\alpha = A_{jj} - A_{jS}DA_{Sj} = (A_{(S\cup\{j\})(S\cup\{j\})})_{j/S}$, we get the exact Gaussian elimination formula for $A^{-1}_{(S\cup\{j\})(S\cup\{j\})}$:

$$w_R^{(S\cup\{j\})} = w_R - A_{R(S\cup\{j\})}A^{-1}_{(S\cup\{j\})(S\cup\{j\})}w_{S\cup\{j\}}$$

So, we have proved that the statement holds for $R = M \cup \{j\}$ indices of w, so obviously, it also holds for a subset $M \subset R$ indices. Thus, we prove the lemma.

□

2.2 Cholesky decomposition and simplifications

Now we get closer to SparseGPT algorithm [2]. Let's consider Cholesky decomposition of matrix $A := H^{-1}$:

$$A = LL^T$$

Now we can try to express $H^{-1}_{/S}$ in terms of L:

$$H^{-1}_{/S} = A_{RR} - A_{RS}A^{-1}_{SS}A_{SR} = L_{R,:}L^T_{:,R} - L_{R,:}L^T_{:,S}(L_{S,:}L^T_{:,S})^{-1}L_{S,:}L^T_{:,R} = \\ = L_{R,:}\left[I - L^T_{:,S}(L_{S,:}L^T_{:,S})^{-1}L_{S,:}\right]L^T_{:,R}$$

Here:

$$L = \begin{bmatrix}L_{SS} & L_{SR} \\ L_{RS} & L_{RR}\end{bmatrix}$$

Notice that $L_{SR} \neq 0$, because there are unpruned indices below the largest pruned index S.

Crucial simplification: At the moment of appending new index j, the impact of previous indices $i < j$ is ommited, i.e. $L_{SR} \approx 0$:

$$L \approx \begin{bmatrix}L_{SS} & 0 \\ L_{RS} & L_{RR}\end{bmatrix}$$

This approximation turns into equality *only when pruned indices all come before unpruned ones.* However, at the first iterations, almost all indices are remaining, and are positioned after the pruned ones, and the error is little. As the iterations continue, the error grows.

Using that simplification, we can significantly simplify $H_{/S}^{-1}$:

$$L_{S,:}L_{:,S}^T = \begin{bmatrix} L_{SS} & L_{SR} \end{bmatrix} \begin{bmatrix} L_{SS}^T \\ (L_{SR})^T \end{bmatrix} \approx L_{SS}L_{SS}^T$$

Then, the term under bracket becomes:

$$I - L_{:,S}^T(L_{S,:}L_{:,S}^T)^{-1}L_{S,:} \approx I - \begin{bmatrix} L_{SS}^T \\ (L_{SR})^T \end{bmatrix} L_{SS}^{-T}L_{SS}^{-1} \begin{bmatrix} L_{SS} & L_{SR} \end{bmatrix} \approx I - \begin{bmatrix} L_{SS}^T \\ 0 \end{bmatrix} L_{SS}^{-T}L_{SS}^{-1} \begin{bmatrix} L_{SS} & 0 \end{bmatrix} =$$
$$= I - \begin{bmatrix} I_S & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & I_R \end{bmatrix}$$

And then:

$$H_{/S}^{-1} \approx \begin{bmatrix} L_{RS} & L_{RR} \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & I_R \end{bmatrix} \begin{bmatrix} (L_{RS})^T \\ L_{RR}^T \end{bmatrix} = L_{RR}L_{RR}^T$$

Now, we can compute $(H_{/S}^{-1})_{jj}$:

$$(H_{/S}^{-1})_{jj} = (L_{RR}L_{RR}^T)_{jj} = \langle L_{jR}, L_{Rj}^T \rangle = \sum_{i<j} L_{ji}^2 + L_{jj}^2$$

Here summation is only on $i < j$ because L_{RR} is lower diagonal.

Next, on an incremental update, we suppose that we take as j the first index of remaining R columns. Generally, this is not true, but here the impact of unpruned $\{i < j : i \in R\}$ is neglected.

$$(H_{/S}^{-1})_{jj} = \sum_{i<j} L_{ji}^2 + L_{jj}^2 \approx L_{jj}^2$$

Then, let's compute $(H_{/S}^{-1})_{:,j}$:

$$(H_{/S}^{-1})_{:,j} = L_{RR}L_{Rj}^T = \begin{bmatrix} L_{1R}, \dots, L_{j-1,R}, L_{jR}, \dots \end{bmatrix} \begin{bmatrix} L_{j1} \\ \dots \\ L_{j,j-1} \\ L_{jj} \\ 0 \\ \dots \end{bmatrix} = \sum_{i<j} L_{ji}L_{iR} + L_{jj}L_{jR}$$

Here again, the impact of $i < j$ is neglected, so we get:

$$(H_{/S}^{-1})_{:,j} = \sum_{i<j} L_{ji}L_{iR} + L_{jj}L_{jR} \approx L_{jj}L_{jR}$$

Now, having the approximations, we can substitute them into the incremental update (Eq. 6):

$$w_{R_n}^{(S_n \cup \{j_{n+1}\})} = w_{R_n}^{(S_n)} - \frac{w_{j_{n+1}}^{(S_n)}}{(H_{/S_n}^{-1})_{j_{n+1}j_{n+1}}} (H_{/S_n}^{-1})_{:,j_{n+1}} \approx w_{R_n}^{(S_n)} - \frac{w_{j_{n+1}}^{(S_n)}}{L_{j_{n+1}j_{n+1}}^2} L_{j_{n+1}j_{n+1}} L_{j_{n+1}R_n}$$

Finally, after reducing a fraction:

$$\boxed{w_{R_n}^{(S_n \cup \{j_{n+1}\})} = w_{R_n}^{(S_n)} - \frac{w_{j_{n+1}}^{(S_n)}}{L_{j_{n+1}j_{n+1}}} L_{j_{n+1}R_n}}$$

And in SparseGPT, the previous unpruned weights are neglected, so the update is performed on indices: $\hat{R_n} := \{i \in R_n : i \geq j_{n+1}\}$. Trivially weight on pruned index $i = j_{n+1}$ is updated to zero, so in the algorithm only $i > j_{n+1}$ indices are updated.

Noticeably, this update formula is not attached to previously pruned indices, and can be performed sequentially on $\{1, \dots, m\}$, while at each i checking whether $i \in P$, and if so, perform the update on the remaining weights. And most importantly, not it can be performed jointly across all rows w_l of matrix W, without the need of row-wise for loop.

I still don't fully understand why block-separation is done, which assumes H^{-1} is a block-diagonal matrix, and why mask is selected as

$$P \leftarrow \text{TopK}_{\text{smallest}} \left(\frac{w_j^2}{L_{jj}^2} \right)$$

instead of the one used in OBS:

$$P \leftarrow \text{TopK}_{\text{smallest}} \left(\frac{w_j^2}{(H^{-1})_{jj}} \right)$$

We can write:

$$(H^{-1})_{jj} = A_{jj} = \sum_{i<j} L_{ji}^2 + L_{jj}^2$$

and again, the impact of previous indices is neglected. It's probably for consistency and to avoid interaction with previous weights, however I can't justify exactly why.

3 Wanda

In Wanda [8], the pruning is simplified even further: the remaining weights no longer get updated:

$$L = \|(w - m \odot w')^T X\|^2 \overset{w=w'}{=} \|((\mathbb{I} - m) \odot w^T) X\|^2 \to \min_{\|m\|_0 = k}$$

We can rewrite the objective as:

$$L = \langle XX^T((\mathbb{I} - m) \odot w), (\mathbb{I} - m) \odot w \rangle$$

And with the indices:

$$L = \langle H_{PP} w_P, w_P \rangle \to \min_{|P|=k}$$

And here the authors simplify the task even further by considering diagonal H:

$$L = \langle H_{PP} w_P, w_P \rangle = \sum_{i \in P} H_{ii} w_i^2 \to \min_{|P|=k}$$

This task has a solution:

$$P = \text{TopK}_{\text{smallest}}(w_j^2 H_{jj}) = \text{TopK}_{\text{smallest}}(w_j^2 \|X_{j,:}\|^2) = \text{TopK}_{\text{smallest}}(|w_j| \|X_{j,:}\|)$$

This is also easily deducable from OBS mask approximation (Eq. 5), where $(H^{-1})_{jj} \overset{H-\text{diag}}{\approx} (H_{jj})^{-1}$.
As a result, Wanda proposes a very simple algorithm for pruning the weights:

$$\boxed{P = \text{TopK}_{\text{smallest}}(|w_j| \|X_{j,:}\|)}$$

Wanda doesn't require updating the remaining weights, as in SparseGPT or OBS, which makes it much faster, and it achieves comparable to SparseGPT results.

4 Outlier weighted layerwise sparsity (OWL)

While previous approaches pruned all layers uniformly, Outlier weighted layerwise sparsity (OWL) [10] proved that this approach is very suboptimal, and it is possible to achieve much better quality by pruning weights adaptively.

As we know, Wanda prunes according to the following metric:

$$P = \text{TopK}_{\text{smallest}}(|w_j|\|X_{j,:}\|) = \text{TopK}_{\text{smallest}}(A_j)$$

where $A_j := |w_j|\|X_{j,:}\|$.

OWL calculates the following statistic about the distribution $\{A_1, \dots, A_m\}$:

$$\hat{D} := \mathbb{P}(A > M \times \mathbb{E}A) = \frac{1}{m}\sum_j [A_j > M\frac{1}{m}\sum_i A_j]$$

where A is a random variable uniformly drawn from $\{A_1, \dots, A_m\}$, M is a hyperparameter (typically in $[3, 10]$).

This statistic tells how heavy is the tail in a column distribution, i.e. how many "outliers" are present.

Then, for each row and each layer, we define:

$$S_{li} = \alpha(1 - \hat{D}_{li})$$

where α is the proportionality coefficient that is same across all layers and is selected so that the resulting sparsity is same as we require.

Suppose $W_l \in \mathbb{R}^{n_l \times m_l}$. Then:

$$\sum_l \sum_i S_{li} m_l = S \sum_l m_l n_l$$

where S is the required total sparsity of the model.

$$\alpha \sum_l \sum_i (1 - \hat{D}_{li}) m_l = S \sum_l m_l n_l \implies \alpha = \frac{S \sum_l m_l n_l}{\sum_l \sum_i (1 - \hat{D}_{li}) m_l}$$

In that formula, we measure the outlier score $\hat{D}_{li}$ for each row in each layer. Using the fact the all rows within the layer have same size, we can simplify:

$$\sum_i (1 - \hat{D}_{li}) m_l = m_l n_l - m_l \sum_i \hat{D}_{li} = m_l n_l - m_l n_l (\frac{1}{n_l}\sum_i \hat{D}_{li}) = m_l n_l (1 - \frac{1}{n_l}\sum_i \hat{D}_{li})$$

Now, we can define:

$$D_l := \frac{1}{n_l}\sum_i \hat{D}_{li} = \mathbb{P}(A_l > M \times \mathbb{E}A_l)$$

where probability and expectation are now taken across all rows and columns: $A_l \in \{A_{11}, \dots, A_{n_l m_l}\}$. This is the outlier score for the weight W_l. Then, we obtain:

$$\alpha = \frac{S \sum_l m_l n_l}{\sum_l m_l n_l (1 - D_l)}$$

Also, it's easy to derive:

$$S_{li} = \alpha(1 - \hat{D}_{li}) \implies \frac{1}{n_l}\sum_i S_{li} = \alpha \frac{1}{n_l}\sum_i (1 - \hat{D}_{li}) = \alpha(1 - \frac{1}{n_l}\hat{D}_{li}) \implies S_l = \alpha(1 - D_l)$$

where S_l is the sparsity of the weight matrix W_l.

So, finally we get:

$$\boxed{D_l = \frac{1}{m_l n_l}\sum_i \sum_j [A_{lij} > M\bar{A}_l]}$$

where $\bar{A}_l = \frac{1}{n_l m_l} \sum_{ij} A_{lij}$

And per-layer sparsity formula:

$$S_{l'} = S \times \frac{\sum_l m_l n_l (1 - D_{l'})}{\sum_l m_l n_l (1 - D_l)}$$

5 Experiments

5.1 Experimental setup

Experiments were performed on CoNLL-2003 [9] Named Entity Recognition (NER) dataset using 'bert-base-cased' model [1]. First, I factorized the embeddings of BERT (factorization dimension is 64) to make him much lighter. Then, I finetuned it to achieve reasonable quality.

All layers except for (factorized) embeddings and output linear projection were pruned. Calibration data was taken from the train subset of the dataset.

All the described above one-shot pruning methods were used. For SparseGPT, default hyperparameters from the paper (block size 128, mask block 64) were used. For OWL, I tried using the recommended $M \in [3, 10]$ values, however I empirically found that $M = 1$ is the most stable.

5.2 Results

Comparison on (Tab. 1) shows that OBS outperforms SparseGPT, especially when sparsity is high, however it takes significantly more time than SparseGPT. Wanda performs well on low sparsity scenarios, but has terrible quality on 70+% sparsity.

Figure (Fig. 1) shows that for high sparsity, sample size becomes very important. Perhaps, F1 would increase even further for larger sample size like 9192 and above.

Table 1: Unstructured pruning on 4096 samples calibration data. OBS performs the best, however takes much longer time than SparseGPT. Wanda is fast, but performs very poorly on highly sparse regimes.

Sparsity	Method	F1 (Uniform)	F1 (OWL)	Time (s)
0%	Dense	0.872	0.872	-
50%	OBS	0.870	0.868	192.5
50%	SparseGPT	0.869	**0.871**	18.0
50%	Wanda	0.862	0.863	**0.9**
60%	OBS	0.862	0.863	231.0
60%	SparseGPT	**0.866**	**0.866**	18.0
60%	Wanda	0.827	0.826	**0.9**
70%	OBS	0.851	**0.857**	267.9
70%	SparseGPT	0.851	0.856	18.0
70%	Wanda	0.526	0.563	**0.9**
80%	OBS	0.804	**0.816**	300.7
80%	SparseGPT	0.770	0.791	18.0
80%	Wanda	0.033	0.042	**0.9**

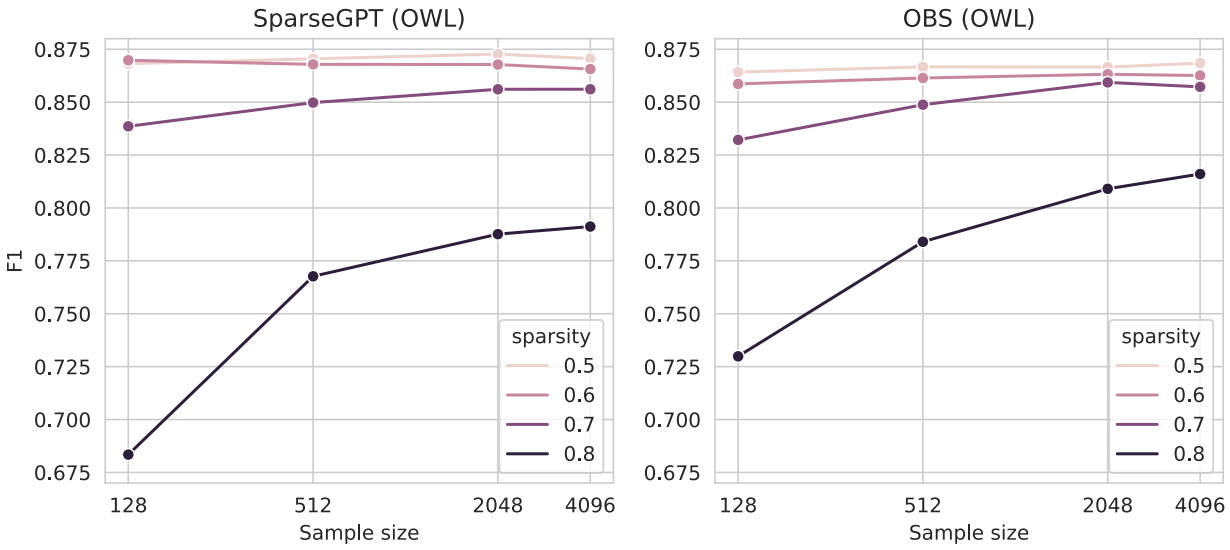

Figure 1: Sample size vs F1 for SparseGPT and OBS. Highly sparse regimes are less sample-efficient and require much more samples.

5.3 Conclusion

From the experiments, I draw several conclusions about the use case of the one-shot pruning methods:

- Wanda: usable when sparsity is relatively low (50% at below). Almost instant pruning time because the remaining weights are not updated.
- OBS: may take a while because it has to update each row of the weight matrices individually. Recommended to use only in high-sparsity scenarios, otherwise SparseGPT/Wanda is preferred due to much lower computational costs.
- SparseGPT: middle ground between Wanda (no update of remaining weights at all) and OBS (exact update of the remaining weights). Great to use everywhere as it doesn't take much time.
- OWL: If you don't tune the M hyperparameter, you are very likely to end up with a quality worse than uniform pruning. For a good M value, OWL generally outperforms uniform pruning. However, for this task, the increase in quality is not as high as I thought based on the results in the original paper.

6 Movement Pruning

Previous approaches (OBS, SparseGPT, Wanda) were one-shot: they require only a small calibration data for pruning. However, if computational resources allow, it is better to finetune the model.

Movement Pruning [7] is a popular supervised pruning technique developed by Hugging Face which uses continuously updated 'movement' scores S for pruning throughout the finetuning process.

6.1 Formulation

Suppose we have a linear layer with weights $W \in \mathbb{R}^{n \times m}$. In Movement Pruning, along with the weights, we store another trainable parameter — the pruning scores $S \in \mathbb{R}^{n \times m}$. During forward pass:

$$M = (M_{ij})_{i=1,j=1}^{n,m}$$

Where in hard movement pruning:

$$M_{ij}^{\text{hard}} := \{(i,j) \notin \underset{i',j'}{TopK_{\text{smallest}}}(S_{i'j'})\}$$

where k is chosen such that the required sparsity is obtained. And in soft movement pruning:

$$M_{ij} := [S_{ij} > \tau]$$

where τ is the fixed for all layers (fixed during a single epoch, however across the epoch sometimes τ scheduling is used) threshold parameter.

Then, the obtained mask is used to prune the weights:

$$y = (W \odot M)x$$

where x is the input of the layer and y is the output.

Noticeably, in both (hard and soft) variants, calculating gradient w.r.t. scores directly is complicated. We can only compute the gradient w.r.t. the mask:

$$y_i = \sum_{j=1}^{M} W_{ij} M_{ij} x_j \implies \frac{\delta L}{\delta M_{ij}} = \frac{\delta L}{\delta y_i}\frac{\delta y_i}{\delta M_{ij}} = \frac{\delta L}{\delta y_i} W_{ij} x_j$$

So, the authors use so called 'straight-through' gradient computation by simply defining:

$$\boxed{\frac{\delta L}{\delta S_{ij}} = \frac{\delta L}{\delta M_{ij}} = \frac{\delta L}{\delta y_i} W_{ij} x_j}$$

Thus, the scores become trainable.

In the hard variant, sparsity is directly controlled via the selection of the $k = mn \times S$ where S is the desired sparsity for the layer. However, in the soft variant with the fixed τ there is no reason for the sparsity to increase. In fact, the model would probably try to increase all scores such that no weights are pruned, thus obtaining the best quality. That is why there is a need for the penalty (regularization) to increase sparsity:

$$L_{\text{mvp}} = L + \lambda_{\text{mvp}} \sum_{l} \sum_{ij} \sigma(S_{ij}^{(l)})$$

where $S^{(l)}$ is the matrix of the scores for linear layer l. This penalty drives the scores towards $-\infty$ which increases the sparsity. It's important to note that there is no direct control over the resulting sparsity. It is only implicitly controlled by the scale of λ_{mvp} hyperparameter: the larger λ_{mvp}, the greater sparsity. This can be potentially seen as a drawback, as achieving a desired level of sparsity may require searching for the best hyperparameter λ_{mvp}. However, in practice, soft Movement Pruning produces better results than the hard one.

6.2 Authors' interpretation

Although we have defined the training (and pruning) procedure, it is still hard to understand why scores trained with straight-through gradient propagation should result in something meaningful. To justify straight-through update, let's notice that:

$$\frac{\delta L}{\delta W_{ij}} = \frac{\delta L}{\delta y_i}\frac{\delta y_i}{\delta W_{ij}} = \frac{\delta L}{\delta y_i} M_{ij} x_j$$

Although the scores S_{ij} will have non-zero gradients even when $M_{ij} = 0$, let's consider the case when $M_{ij} = 1$, because that's when the weight gradient is not necessarily zero.

So, if $M_{ij} = 1$, then:

$$\frac{\delta L}{\delta W_{ij}} = \frac{\delta L}{\delta y_i} x_j \implies \frac{\delta L}{\delta S_{ij}} = \frac{\delta L}{\delta y_i} W_{ij} x_j = W_{ij} \frac{\delta L}{\delta W_{ij}}$$

Let's analyze the sign of the score gradient: if $\frac{\delta L}{\delta S_{ij}} < 0$, that means that the score will increase after the gradient update (because the task is to minimize the loss and the gradient gets subtracted from the value). Negative sign has 2 possible scenarios:

1) $W_{ij} > 0$, $\frac{\delta L}{\delta W_{ij}} < 0$ (positive weight will increase after update)

2) $W_{ij} < 0$, $\frac{\delta L}{\delta W_{ij}} > 0$ (negative weight will decrease after update)

In both scenarios, the weight goes further away from 0 after the update. That's why the score increases, which results in lower chance of pruning the weight.

If $\frac{\delta L}{\delta S_{ij}} > 0$, it's easy to show that the opposite happens: the weight goes towards zero, thus it's meaningful to increase the chance of pruning the weight.

Okay, we figured out that the sign of the gradient is meaningful and tells us about the direction in which the weight is moving (to or away from zero). But what about the absolute value? If the gradient w.r.t. the weight $|\frac{\delta L}{\delta W}|$ is large, the weight will be heavily pushed to or away from zero, so it makes sense to update the score heavily too. I can't provide the meaningful explanation about the reason being the impact of the absolute weight value $|W|$ on the score update. One possible explanation is the following: if the weight value is low: $|W| \approx 0$, the weight sign can change easily. That's why we can't fully trust the direction in which we are going. On the opposite, if $|W| \gg 0$, we are pretty sure about its sign and that it is not going to change. Thus, we can update the score more aggressively.

With the slight abuse of notation, let's focus on the individual weight S_{ij} and denote it as S.

Assuming the standard gradient descent, we can write:

$$S(t+1) = S(t) - \alpha_t \frac{\delta L}{\delta S}(t) = S(t) - \alpha_t W(t) \frac{\delta L}{\delta W}(t)$$

where t is the number of the iteration, α_t is the learning rate.

The scores are usually initialized with zeros, so at the final iteration T we have:

$$\boxed{S(T) = -\sum_{t=0}^{T-1} \alpha_t W(t) \frac{\delta L}{\delta W}(t)} \tag{7}$$

The authors propose to interpret these scores as the average (weighted) likelihood of the weights going away from zero throughout the training. If the weight is generally being pushed away from zero, the score is high, and it's very unlikely that it is pruned.

6.3 My interpretation

Here is my personal and perhaps more concrete interpretation of the final score (Eq. 7). We can observe that:

$$\frac{\delta L}{\delta \log |W|} = \frac{\delta L}{\delta W} \frac{\delta W}{\delta \log |W|} = \frac{\delta L}{\delta W} \left(\frac{\delta \log |W|}{\delta W} \right)^{-1} = \frac{\delta L}{\delta W} \left(\frac{1}{W} \right)^{-1} = \frac{\delta L}{\delta W} W$$

Using this identity, we can rewrite (Eq. 7):

$$S(T) = -\sum_{t=0}^{T-1} \alpha_t W(t) \frac{\delta L}{\delta W}(t) = -\sum_{t=0}^{T-1} \alpha_t \frac{\delta L}{\delta \log |W|}(t)$$

This is basically a gradient descend for $\log |W|$. We can write that gradient descend:

$$\log |W^\star| = \log |W(0)| - \sum_{t=0}^{T-1} \alpha_t \frac{\delta L}{\delta \log |W|}(t)$$

where $W^\star$ is the final weight that would have been obtained by performing gradient descend on $\log|W|$. However, it's important to notice that here we're calculating the gradient in the points W which are updated by descend on W, not on it's logarithm, so the following equality is approximate:

$$S(T) \approx \log|W^\star| - \log|W(0)|$$

This is easily interpretable: if $\log|W^\star| - \log|W(0)|$ is low, that means that decreasing $\log|W|$, and thus decreasing $|W|$ (because logarithm is the monotone function) improves the loss, because the gradient descend is (approximately) on $\log|W|$. Then, it makes sense to decrease it even further and prune it completely.

In other scenario, if $\log|W^\star| - \log|W(0)|$ is not low, decreasing the absolute weight value will improve the loss less (commonly, even increase it), so it's better not to prune them.

However, a curious reader may ask: **"Why not directly compute** $\log|W(T)| - \log|W(0)|$ **and prune by that difference?"**. I think, the main reason is that the method is continuously applying the pruning throughout training, thus **the remaining weights have the time to adapt to the pruning**, and the final scores account for the updated weights.

7 Optimal BERT Surgeon (oBERT)

7.1 Prelude: Hessian approximation with Fisher matrix

In this section I'll shortly derive a very common approximation of the Hessian used in machine learning, namely, with Fisher matrix.

Let's denote a loss on a single sample with $l_w(y, \hat{y})$, and f as the model, such that we get $\hat{y} = f(x)$. Suppose that the loss represents a negative logarithm of likelihood (NLL):

$$l_w(y, f(x)) = -\log p_w(y|x)$$

where p_w is the pdf for the distribution that model weights w generate. This assumption is commonly the case in most optimization tasks. For example, MSE loss is NLL for Normal distribution and Cross Entropy Loss is NLL for Bernoulli.

The training objective is to minimize:

$$L(w) := \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{D}} l_w(y, f(x)) = \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{D}}(-\log p_w(y|x)) \to \min_w$$

where $\mathbb{D}$ is the data distribution.

Data distribution $\mathbb{D}$ implicitly defines an unknown (and possibly very complex) pdf $p_{\mathbb{D}}(y, \hat{y})$. We can add and substract data entropy from the loss:

$$L(w) = \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{D}}(-\log p_w(y|x) + \log p_{\mathbb{D}}(y|x) - \log p_{\mathbb{D}}(y|x)) = \mathrm{KL}(p_{\mathbb{D}}(y|x)\|p_w(y|x)) - \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{D}} \log p_{\mathbb{D}}(y|x)$$

The entropy doesn't depend on w, so we proved that the task is equivalent to minimizing KL divergence between data distribution and model generated distribution. Next, we will use a crucial assumption, that the model is well-fit, meaning:

$$p_w(y|x) \approx p_{\mathbb{D}}(y|x)$$

Next, let's derive the following equation for the log-derivative:

$$\mathbb{E}_w\left(\frac{\delta \log p_w(y|x)}{\delta w}\right) = \int \frac{\delta \log p_w(y|x)}{\delta w} p_w(y|x)dxdy = \int \frac{1}{p_w(y|x)} \frac{\delta p_w(y|x)}{\delta w} p_w(y|x)dxdy =$$
$$= \int \frac{\delta p_w(y|x)}{\delta w} dxdy = \frac{\delta}{\delta w} \int p_w(y|x)dxdy = \frac{\delta}{\delta w} 1 = 0$$

Here we assume that the integral is "good" enough that we can factor out differentiation. So, we showed that expectation of the log-derivative term $\frac{\delta \log p_w(y|x)}{\delta w}$, also sometimes called score-function, is zero.

Let's take derivative of this expectation, which should result in also zero:

$$0 = \frac{\delta}{\delta w}\mathbb{E}_w(\frac{\delta \log p_w(y|x)}{\delta w}) = \int \frac{\delta}{\delta w}(\frac{\delta \log p_w(y|x)}{\delta w} p_w(y|x))dxdy =$$

$$= \int \frac{\delta^2 \log p_w(y|x)}{\delta^2 w} p_w(y|x)dxdy + \int \frac{\delta \log p_w(y|x)}{\delta w}\left(\frac{\delta p_w(y|x)}{\delta w}\right)^T dxdy =$$

$$= \int \frac{\delta^2 \log p_w(y|x)}{\delta^2 w} p_w(y|x)dxdy + \int \frac{\delta \log p_w(y|x)}{\delta w}\left(\frac{\delta \log p_w(y|x)}{\delta w}\right)^T p_w(y|x)dxdy =$$

$$= \mathbb{E}_w\left(\frac{\delta^2 \log p_w(y|x)}{\delta^2 w}\right) + \mathbb{E}_w\left(\frac{\delta \log p_w(y|x)}{\delta w}\left(\frac{\delta \log p_w(y|x)}{\delta w}\right)^T\right)$$

Here, the last expectation is often called a "Fisher" matrix.

Next, let's go back to the loss notations:

$$l_w(y, f(x)) = -\log p_w(y|x) \implies \nabla l_w = -\frac{\delta \log p_w(y|x)}{\delta w}, \nabla^2 l_w = -\frac{\delta^2 \log p_w(y|x)}{\delta w^2}$$

Substituting this into previously obtained equation, we get:

$$\mathbb{E}_w(-\nabla^2 l_w) + \mathbb{E}_w(\nabla l_w \nabla l_w^T) = 0 \implies \mathbb{E}_w(\nabla^2 l_w) = \mathbb{E}_w(\nabla l_w \nabla l_w^T)$$

Finally, using the fact that the model is well-fit and model distribution is approximately data distribution, we get:

$$\nabla^2 L(w) = \nabla^2 \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{D}} l_w(y, f(x)) = \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{D}} \nabla^2 l_w(y, f(x)) \approx \mathop{\mathbb{E}}_{(x,y)\sim w} \nabla^2 l_w(y, f(x)) =$$

$$= \mathop{\mathbb{E}}_{(x,y)\sim w} (\nabla l_w(y, f(x))(\nabla l_w(y, f(x))^T) \approx \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{D}} (\nabla l_w(y, f(x))(\nabla l_w(y, f(x))^T)$$

So, finally, after replacing the expectation with empirical mean, we get an approximation for the Hessian:

$$\boxed{\nabla^2 L(w) \approx \frac{1}{N}\sum_{q=1}^{N} \nabla l(y_q, f(x_q))\nabla l(y_q, f(x_q))^T} \tag{8}$$

7.2 Task formulation

Let's consider the weight row $w \in \mathbb{R}^m$ (index of the row and of the layer are omitted for clarity). Suppose $w' \in \mathbb{R}^m$ is the new updated row, where some values are pruned.

We can write second order Taylor composition for $L(w')$:

$$L(w') \approx L(w) + \langle \nabla L(w), w' - w\rangle + \frac{1}{2}\langle \nabla^2 L(w)(w' - w), w' - w\rangle \tag{9}$$

Here we make an assumption that w is close to the optimum, which is typically the case for weights after training, so $\nabla L(w) \approx 0$. Interestingly, it is possible to obtain update formulas without this assumption, however they are more complex because they include $\nabla L(w)$. Let's define $H := \nabla^2 L(w)$ as the Hessian. Now, our task becomes:

$$L(w') \approx L(w) + \frac{1}{2}\langle H(w' - w), w' - w\rangle \to \min_{\|w'\|_0 = k}$$

where we minimize by the w' with k pruned values. $L(w)$ doesn't depend on w', so we obtain the following optimization task:

$$\langle H(w' - w), w' - w\rangle \to \min_{\|w'\|_0 = k} \tag{10}$$

7.3 Connection to one-shot pruning methods

We can notice that this looks very similar to the task (Task 1), that was used for one-shot pruning methods. The only difference is the Hessian: in one-shot methods $H = XX^T = \sum_q x_q x_q^T$, whereas here $H = \nabla^2 L(w)$.

In order to make some connection between those matrices, let's use well-known approximation for the Hessian with the empirical Fisher matrix, that we have derived earlier (Eq. 8):

$$\nabla^2 L(w) \approx \frac{1}{N} \sum_{q=1}^{N} \nabla l(w, x_q) \nabla l(w, x_q)^T$$

where $L(w, x_q)$ is the loss obtained for the input x_q. Here with the slight abuse of notation I denote x_q as the layer input instead of the model input, and the weights w as the row instead of all model weights, as in (Eq. 8). $l(w, x_q)$ is the loss obtained for weight row w given layer input x_q.

Suppose $y_q := \langle w, x_q \rangle$ is the output for the row w. If W is the weight matrix, then $y_{qi} = \langle w_i, x_q \rangle$ is the ith component of the output. Now we can write:

$$\nabla l(w, x_q) = \frac{\delta l}{\delta w} = \frac{\delta l}{\delta y_q} \frac{\delta y_q}{\delta w} = \frac{\delta l}{\delta y_q} x_q$$

Substituting that in the approximation for the Hessian, we obtain:

$$\nabla^2 L(w) \approx \frac{1}{N} \sum_{q=1}^{N} \left(\frac{\delta l}{\delta y_q} \right)^2 x_q x_q^T$$

Now, all that is left to get to the one-shot matrix is to omit the activation gradient $\frac{\delta l}{\delta y_q}$. One possible reason for omitting the output gradient is if they were statistically independent from the input xx^T. Interestingly, the same assumption about gradients w.r.t outputs and input activations was made in a second order optimizer K-FAC [6].

$$\mathbb{E} \frac{\delta l}{\delta y} (xx^T) \approx \mathbb{E} \frac{\delta l}{\delta y} \cdot \mathbb{E} xx^T$$

Then, for each row, we have a coefficient before the matrix XX^T, which doesn't impact the optimization.

So, finally, under this heavy simplification:

$$\nabla^2 L(w) \approx \frac{1}{N} \left(\frac{\delta l}{\delta y} \right)^2 \sum_{q=1}^{N} x_q x_q^T = \frac{1}{N} \left(\frac{\delta l}{\delta y} \right)^2 XX^T \propto XX^T$$

where $\left(\frac{\delta l}{\delta y} \right)^2$ is the coefficient that is factored out.

Now we can see another interpretation of the minimization task (Task 1) used in one-shot methods, other than just minimizing the difference between new and old outputs. Also, we can use the same solutions (Eq. 2, 5) with H being the Hessian.

7.4 Approximate Hessian computation

Great, we have shown in the previous section, that the loss minimization task has the same structure as (Task 1), and we can use solutions (Eq. 2, 5). However, for that we need to know the Hessian for the initial weights w. In practice, computing (and even storing) Hessian is prohibitively infeasible, so we need to approximate it.

First, the authors of oBERT [5] also use the Fisher approximation (Eq. 8) for the (flattened) weight matrix:

$$\nabla^2 L(W) \approx \frac{1}{N} \sum_{q=1}^{N} \nabla l_q \nabla l_q^T$$

For the OBS solutions, an inverse Hessian is required. So, the authors propose to use Woodbury matrix identity [3]:

$$(A+UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1}+VA^{-1}U)^{-1}VA^{-1} \in \mathbb{R}^{mn\times mn}$$

We can write the Fisher approximation as the continuous update:

$$F_0 = \lambda I, \quad F_q = F_{q-1} + \frac{1}{N}\nabla l_q \nabla l_q^T, \quad \nabla^2 L(W) = F_N$$

where F_0 is initialized with a small positive identity for stability.

Using the Woodbury identity with $A = F_{q-1}, \quad U = \nabla l_q, \quad C = \frac{1}{N}, \quad V = \nabla l_q^T$, we get:

$$F_q^{-1} = \left(F_{q-1} + \nabla l_q \left(\frac{1}{N}\right) \nabla l_q^T\right)^{-1} \overset{\text{Woodbury}}{=} F_{q-1}^{-1} - \frac{F_{q-1}^{-1}\nabla l_q \nabla l_q^T F_{q-1}^{-1}}{N + \langle F_{q-1}^{-1}\nabla l_q, \nabla l_q\rangle}$$

So, we can continuously update F_q^{-1}, starting from $F_0^{-1} = \frac{1}{\lambda}I$.

However, storing such a matrix F_q^{-1} is still infeasible, and performing matrix-vector multiplications on it is also expensive. So, the authors use the following approximation: **At each step F_q^{-1}, and thus the inverse Hessian, are block-diagonal**.

$$F_q^{-1} = \begin{bmatrix} \hat{F}_q^{(1)} & 0 & \dots & 0 \\ 0 & \hat{F}_q^{(2)} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \hat{F}_q^{(S)} \end{bmatrix}$$

where $\hat{F} \in \mathbb{R}^{B\times B}$ is the block of F^{-1}, and $S := \frac{mn}{B}$ is the number of blocks.

$$F_q^{-1}\nabla l_q = \begin{bmatrix} \hat{F}_q^{(1)}\nabla l_q^{(1)} \\ \dots \\ \hat{F}_q^{(S)}\nabla l_q^{(S)} \end{bmatrix}$$

where $\nabla l = \begin{bmatrix} l_q^{(1)} \\ \dots \\ l_q^{(S)} \end{bmatrix}$

Noticeably, in the numerator of the update formula we have $(F_q^{-1}\nabla l_q)(F_q^{-1}\nabla l_q)^T$ which at the end will include off-block-diagonal terms. However, we ignore them as part of the block-diagonal approximation:

$$(F_q^{-1}\nabla l_q)(F_q^{-1}\nabla l_q)^T \approx \begin{bmatrix} (\hat{F}_q^{(1)}\nabla l_q^{(1)})(\hat{F}_q^{(1)}\nabla l_q^{(1)})^T & 0 & \dots & 0 \\ 0 & (\hat{F}_q^{(2)}\nabla l_q^{(2)})(\hat{F}_q^{(2)}\nabla l_q^{(2)})^T & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & (\hat{F}_q^{(S)}\nabla l_q^{(S)})(\hat{F}_q^{(S)}\nabla l_q^{(S)})^T \end{bmatrix}$$

For the denominator of the fraction:

$$\langle F^{-1}\nabla l, \nabla l\rangle = \sum_{b=1}^{S} \langle \hat{F}^{(b)}\nabla l^{(b)}, \nabla l^{(b)}\rangle$$

So, after combining everything together, we have:

$$\hat{F}_q^{(b)} = \hat{F}_{q-1}^{(b)} - \frac{\hat{F}_{q-1}^{(b)}\nabla l_q^{(b)}(\nabla l_q^{(b)})^T \hat{F}_{q-1}^{(b)}}{N + \sum_{b'=1}^{S}\langle \hat{F}_{q-1}^{(b')}\nabla l_q^{(b')}, \nabla l_q^{(b')}\rangle}$$

In oBERT, the denominator is approximated so that each block update doesn't depend on the other blocks, and thus the block update can be treated as the iterative approximation, as if there was only that block as the whole matrix.

$$\sum_{b'=1}^{S} \langle \hat{F}_{q-1}^{(b')} \nabla l_q^{(b')}, \nabla l_q^{(b')} \rangle \approx \langle \hat{F}_{q-1}^{(b)} \nabla l_q^{(b)}, \nabla l_q^{(b)} \rangle$$

This can be a heavy approximation, however it's only a scalar, so perhaps this adjustment doesn't affect the final result a lot.

Finally, we get the following online block update for each block $b = 1, \ldots, S$:

$$\boxed{\hat{F}_q^{(b)} = \hat{F}_{q-1}^{(b)} - \frac{\hat{F}_{q-1}^{(b)} \nabla l_q^{(b)} (\nabla l_q^{(b)})^T \hat{F}_{q-1}^{(b)}}{N + \langle \hat{F}_{q-1}^{(b)} \nabla l_q^{(b)}, \nabla l_q^{(b)} \rangle}}$$

In this case, we don't store the full (inverse) Hessian, but only its S $B \times B$ blocks, thus the memory complexity is $SB^2 = Bmn$. In the oBERT paper, the authors used $B = 50$. However, for large LLMs 50x memory increase is almost always infeasible. This is a big drawback of this method.

7.5 Weight pruning and remaining weight update

After computing the block-diagonal Hessian approximation, we can now perform weight pruning and update for the remaining weights. Noticeably, in this case we can flatten the weights without thinking about the columns, because we already know the (approximate) Hessian for all weights. So, we similarly update the (Task 10):

$$\langle H(w' - w), w' - w \rangle \to \min_{\|w'\|_0 = k}$$

where $w, w' \in \mathbb{R}^{mn}$ are the flattened weight matrices, and $H \in \mathbb{R}^{mn \times mn}$ is the Hessian, inverse of which was computed with the iterative formula from previous section.

Now, we can use (Eq. 2):

$$w'_R = w_R - (H^{-1})_{RP})((H^{-1})_{PP})^{-1} w_P$$

We can show that:

$$w'_P = w_P - (H^{-1})_{PP})((H^{-1})_{PP})^{-1} w_P = w_P - w_P = 0$$

So, it's possible to write the unified equation:

$$w' = w - (H^{-1})_{:,P}((H^{-1})_{PP})^{-1} w_P$$

Now we can use the block-diagonal structure of the Hessian to write this expression for each block:

$$w'_b = w_b - (\hat{F}^{(b)})_{:,P_b}((\hat{F}^{(b)})_{P_b P_b})^{-1} w_{P_b}$$

where b is the index of the block, $w_b \in \mathbb{R}^B$ are the weights corresponding to that block, P_b are the pruned indices within block b, and $\hat{F}^{(b)} := \hat{F}_N^{(b)}$ is the bth block of the block-diagonal inverse Hessian H^{-1}.

However, this equation is still hard to compute as the number of blocks S is large. So, the authors propose a heavy assumption:

$$((\hat{F}^{(b)})_{P_b P_b})^{-1} \approx (\text{diag}(\hat{F}_{ii}^{(b)}, i \in P_b))^{-1} = \text{diag}(\hat{F}_{ii}^{(b)})_{P_b P_b}^{-1}$$

This is, of course, generally false. However, if the block size is relatively low, and using the fact that the Hessian is usually diagonally dominant, this assumption could make some sense.

Under that assumption, we get:

$$w'_b = w_b - (\hat{F}^{(b)})_{:,P_b}((\hat{F}^{(b)})_{P_b P_b})^{-1} w_{P_b} \approx w_b - (\hat{F}^{(b)})_{:,P_b}(\text{diag}(\hat{F}^{(b)})_{P_b P_b})^{-1} w_{P_b} =$$

$$= w_b - \begin{bmatrix} (\hat{F}^{(b)})_{:,P_b} & (\hat{F}^{(b)})_{:,R_b} \end{bmatrix} \begin{bmatrix} (\text{diag}(\hat{F}^{(b)})_{P_b P_b})^{-1} w_{P_b} \\ 0 \end{bmatrix} = w_b - \hat{F}^{(b)}(\text{diag}(\hat{F}^{(b)}))^{-1}(w_b \odot (1 - M_b))$$

where M_b is the mask of the remaining indices inside block b.

Then, using the block-diagonal structure of $\hat{F}$, we can write joint equation for updating all weights:

$$\boxed{w' = w - \hat{F}(\text{diag}(\hat{F}))^{-1}(w \odot (1 - M))}$$

where $M \in \mathbb{R}^{mn}$ is the entire mask of remaining weights. This expression can be efficiently computed for each layer.

And finally, as for the mask selection, oBERT uses the same formula as in OBS (Eq. 5):

$$P = \text{TopK}_{\text{smallest}} \left(\frac{w_j^2}{H_{jj}^{-1}} \right)$$

In the new notation:

$$\boxed{M_i = \left[w_i \in \text{TopK}_{\text{largest}} \left(\frac{w_j^2}{\hat{F}_{jj}} \right) \right]}$$

7.6 Multiple epochs

Let's refer to the (approximate) inverse Hessian computation and to weight pruning as a single **epoch**. The authors empirically observed that it is better to gradually increase the sparsity, rather than do everything in one-shot. This makes total sense: the second order Taylor approximation (Eq. 9) assumes that the new weights w' are in a sense "close" to the original weights w. So, for the lower sparsity, the approximation is more exact, than for the higher sparsity.

That is why, the authors propose to perform pruning in several epochs. Each time increasing the sparsity towards the final desired level. Obviously, if the weights were pruned in the previous epochs, they would have zero magnitude, so they will be also pruned in the next epochs:

$$P^{(1)} \subset P^{(2)} \subset \ldots \subset P^{(T)}$$

where T is the number of epochs and $P^{(t)}$ is the subset of pruned indices on epoch t.

An important note is that throughout the pruning epochs oBERT method **does not** perform the gradient descend w.r.t. to the loss. However, after the pruning epochs, it's desirable to finetune the final remaining weights adapt them even better. During this process, the pruned weights are fixed and don't get updated. So, the algorithm generally looks like:

(optional) T_i epochs of initial finetuning
$\rightarrow$ T epochs of pruning
$\rightarrow$ T_R epochs of finetuning of the remaining weights.

8 Conclusion

This deep-dive illustrates that the evolution of one-shot unstructured pruning is largely an exercise in the efficient approximation of the OBS-style task solution. Second-order training-based methods essentially replace activation covariance with hessian approximation for the same task. Movement pruning stands apart from the rest of the methods as it has no connection with OBS task and instead prunes based on scores which reflect the connection between pushing the weight towards zero and the loss. By synthesizing these approaches into a unified mathematical narrative, I hope this document provides a clearer intuition for the underlying mechanics of modern model compression.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[2] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International conference on machine learning*, pages 10323–10337. PMLR, 2023.

[3] William W Hager. Updating the inverse of a matrix. *SIAM review*, 31(2):221–239, 1989.

[4] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pages 293–299. IEEE, 1993.

[5] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models. *arXiv preprint arXiv:2203.07259*, 2022.

[6] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[7] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in neural information processing systems*, 33:20378–20389, 2020.

[8] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

[9] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003.

[10] Lu Yin, You Wu, Zhenyu Zhang, Cheng-Yu Hsieh, Yaqing Wang, Yiling Jia, Gen Li, Ajay Jaiswal, Mykola Pechenizkiy, Yi Liang, et al. Outlier weighed layerwise sparsity (owl): A missing secret sauce for pruning llms to high sparsity. *arXiv preprint arXiv:2310.05175*, 2023.

[11] Fuzhen Zhang. *The Schur complement and its applications*, volume 4. Springer Science & Business Media, 2006.