Batch Loss Score for Dynamic Data Pruning

Supplementary Material

7. Simplicity of BLS: *One-Line Proxy* and Seamless Integration

A hallmark of BLS is its exceptional ease of integration into existing training workflows, particularly when enhancing dynamic data selection frameworks like InfoBatch [36] or SeTa [54] that were originally designed around per-sample losses ($l_i(t)$). BLS achieves this through a conceptual ***one-line proxy*** mechanism that seamlessly modifies the score generation within such frameworks, coupled with an intrinsically simple update logic. This entirely obviates the need for direct $l_i(t)$ computation, a significant practical advantage as shown in Figure 4 and detailed in Sec. 8.

Algorithm 1 illustrates this streamlined integration, contextualized with a framework like InfoBatch. The core idea is that BLS wraps the existing data handling object (e.g., an `InfoBatchInstance`) and, through its conceptual `.proxy()` method (realized by the BLS class constructor in our implementation), effectively overrides or augments its internal score update mechanism to use the mean batch loss instead of per-sample losses.

The ***one-line proxy*** is demonstrated at Line 2 of Algorithm 1: `DataHandler` $\leftarrow$ `BLS(InfoBatchInst,` α`).proxy()`. Here, `InfoBatchInst` is an initialized instance of a base pruning framework. The `BLS` wrapper effectively modifies this instance (now referred to as `DataHandler`) such that its subsequent `.update()` calls will utilize BLS's scoring logic, which operates on the mean batch loss. This single line conceptually redirects the core of the importance scoring from a per-sample loss basis to a batch loss basis.

The ***three-line injection*** then highlights the minimal conceptual changes to a standard training loop when using this BLS-enhanced data handler:

1. **Line 3 (Framework Wrapping):** `InfoBatchInst` $\leftarrow$ `InfoBatch(...)`. This is the standard initialization of the base pruning framework (e.g., InfoBatch). The subsequent BLS proxy (Line 4) builds upon this.
2. **Line 5 (Sampler Integration):** `Loader` $\leftarrow$ `DataLoader(DataHandler, sampler=DataHandler.sampler)`. The DataLoader utilizes the custom sampler provided by the `DataHandler` (the BLS-proxied InfoBatch instance). This sampler now implicitly operates based on scores that will be generated and maintained by BLS.
3. **Line 12 (Proxied Update Call):** `loss_final` $\leftarrow$ `DataHandler.update(`L_t`)`. This line, within the training loop, calls the `update` method. Crucially, due to the BLS proxy, this method now takes the standard mean batch loss L_t (obtained on Line 11 using a criterion with `reduction="mean"`) as its primary input for scoring. As noted in the algorithm's comments, the internal logic of this proxied `update` method then executes BLS's core EMA calculation for samples in the current batch using this L_t. The returned `loss_final` might be the original L_t or a version rescaled by the underlying framework's (e.g., InfoBatch's) mechanism for gradient adjustment.

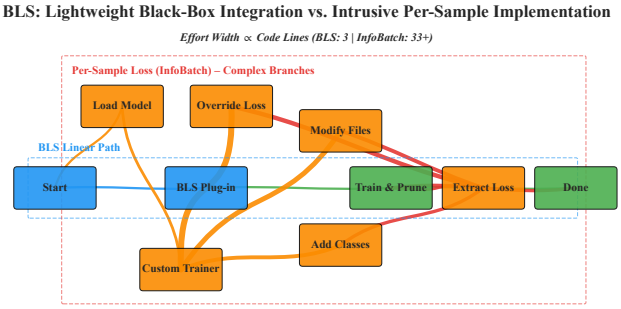

Figure 4. Workflow Comparison: BLS Black-Box Simplicity vs. Per-Sample Loss Complexity.

Thus, BLS leverages the existing infrastructure of methods like InfoBatch/SeTa for sampling and potential gradient adjustments but critically simplifies the score generation by using the universally available mean batch loss. This minimal footprint and reliance only on L_t underscore BLS's practical utility and ease of adoption for efficient dynamic data pruning.

8. On the Practical Difficulty of Obtaining Per-Sample Losses

The main text highlights the practical challenges in obtaining per-sample losses $l_i(t)$ for dynamic data selection. This appendix provides a more granular discussion of these difficulties, underscoring the motivation for methods like BLS that operate solely on aggregated batch losses. We first discuss general framework-level hurdles and then delve into task-specific complexities using object detection as a prime example, finally illustrating the differing access complexities with pseudocode.

8.1. Standard Loss Aggregation and Framework Encapsulation

Modern deep learning frameworks, such as PyTorch [33], are architected primarily for computational efficiency. A key aspect of this design is the default aggregation of losses at the

Algorithm 1 BLS Integration: **One-Line Proxy** and **Three-Line Injection**

Input: Original dataset $\mathcal{D}_{\text{train}}$, framework args $\theta_{\text{Framework}}$, BLS decay α.
Output: Trained model parameters $\mathbf{W}^*$.

InfoBatchInst $\leftarrow$ InfoBatch($\mathcal{D}_{\text{train}}, \theta_{\text{Framework}}$) ▷ #1 Wrap dataset
DataHandler $\leftarrow$ BLS(InfoBatchInst, α).proxy() ▷ One-Line Proxy: BLS modifies InfoBatch's update
Loader $\leftarrow$ DataLoader(DataHandler, sampler = DataHandler.sampler, . . .) ▷ #2 Inject sampler

Initialize model $\mathcal{M}(\mathbf{W})$, optimizer $\mathcal{O}$
Criterion $\mathcal{C} \leftarrow$ LossFunction(reduction="mean") ▷ BLS uses standard batch loss
for epoch $\in [1, \ldots, N_{\text{epochs}}]$ **do**
  for ($\mathbf{x}_{\text{batch}}$, $\mathbf{y}_{\text{batch}}$) **in** Loader **do** ▷ *Indices are set on DataHandler internally by DataLoader hack*
    $\mathcal{O}$.zero_grad()
    $L_t \leftarrow \mathcal{C}(\mathcal{M}(\mathbf{x}_{\text{batch}}), \mathbf{y}_{\text{batch}})$ ▷ Obtain standard mean batch loss
    $\text{loss}_{\text{final}} \leftarrow$ DataHandler.update(L_t) ▷ #3 Proxied update uses L_t
    ▷ *Internally, performs BLS EMA update on current batch samples*
    $\text{loss}_{\text{final}}$.backward()
    $\mathcal{O}$.step()
  end for
end for
return Trained model parameters $\mathbf{W}^*$

batch level. Standard loss function implementations typically perform an immediate reduction (e.g., 'reduction='mean'' or 'reduction='sum''), yielding a single scalar batch loss which serves as the primary input for backpropagation. Accessing individual sample losses $l_i(t)$ therefore necessitates a deviation from this default, requiring explicit configuration of the loss function (e.g., 'reduction='none'') to return a tensor of per-sample losses (e.g., shape $[B]$ for batch size B). While technically feasible for certain loss types, this explicit non-reduction forces modifications to standard training pipelines, as training loops and gradient computation logic are typically designed for scalar losses. Furthermore, high-level training libraries and APIs (e.g., YOLO [39]) often introduce further layers of abstraction over the loss computation. Unpacking these abstractions to expose per-sample losses can require intrusive modifications to internal library code, thereby increasing implementation complexity and reducing the portability of such custom solutions.

8.2. Case Study: Per-Sample Loss in Object Detection

Object detection serves as a salient example where defining and extracting a representative per-sample loss $l_i(t)$ is highly non-trivial. For a single input image i in a typical detector (e.g., YOLO [39]), the loss computation involves several stages. Initially, numerous region proposals or anchor boxes are generated and matched against ground truth objects. For each matched positive proposal p associated with image i, at least two loss components are computed: a classification loss $l_{\text{cls}}(p)$ and a bounding box regression loss $l_{\text{reg}}(p)$. Background proposals typically only incur a classification loss. The total loss for image i, denoted l_i^{task}, is then an aggregation, often a weighted sum, of these individual proposal/anchor losses:

$$l_i^{\text{task}} = \sum_{p \in \text{pos\_proposals}_i} (\lambda_1 l_{\text{cls}}(p) + \lambda_2 l_{\text{reg}}(p)) + \sum_{p \in \text{neg\_proposals}_i} \lambda_3 l_{\text{cls}}(p)_{\text{bg}}, \tag{10}$$

where λ are weighting factors. This l_i^{task} represents the per-sample loss one would ideally want. Finally, the batch loss L used for gradient updates is the average of these l_i^{task} values over all images i in the batch $\mathcal{B}_t$.

The primary challenge in obtaining l_i^{task} lies in the encapsulation by object detection frameworks. These frameworks typically integrate the entire multi-stage loss calculation, outputting only the final scalar $L(\mathcal{B}_t, t)$. Extracting the intermediate l_i^{task} for each image usually requires modifying these complex, highly optimized internal modules, as a simple 'reduction='none'' at the image level is often unavailable or insufficient. Even if l_i^{task} could be extracted, its utility as a single "hardness" indicator is debatable. It aggregates potentially hundreds of sub-losses; an image with one severe localization error might yield a similar l_i^{task} to an image with many minor classification errors, despite representing different types of model failure. Furthermore, the variable number of proposals contributing to l_i^{task} across images complicates direct comparisons without careful normalization. Thus, for complex tasks like object detection, obtaining a meaningful and accessible $l_i(t)$ often necessitates deep architectural and task-specific interventions.

Algorithm 2 Illustrative Comparison of Loss Access (Object Detection as an Example)

Input: Model $f(\cdot;\theta)$ (Backbone, RPN, Head), Batch $\mathcal{B}_t = \{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{B}$

procedure GETBATCHLOSSSTANDARD($\mathcal{B}_t, f$) ▷ Standard, Direct Path
  $\text{Features}_{\text{batch}} \leftarrow f_{\text{backbone}}(\mathbf{X}_{\text{batch}})$
  $\text{Proposals}_{\text{batch}}, l_{\text{RPN}}^{\text{batch}} \leftarrow f_{\text{RPN}}(\text{Features}_{\text{batch}}, \mathbf{Y}_{\text{batch}})$
  $\text{Detections}_{\text{batch}}, l_{\text{Head}}^{\text{batch}} \leftarrow f_{\text{head}}(\text{Features}_{\text{batch}}, \text{Proposals}_{\text{batch}}, \mathbf{Y}_{\text{batch}})$
  $L \leftarrow (l_{\text{RPN}}^{\text{batch}} + l_{\text{Head}}^{\text{batch}})/B$ ▷ Framework aggregates internally
  return L
end procedure

procedure GETPERSAMPLELOSSESCOMPLEX($\mathcal{B}_t, f$) ▷ Involved Path (e.g., Object Detection)
  per_sample_losses$[1..B] \leftarrow$ InitializeArray(B)
  for $k = 1$ **to** B **do** ▷ Iterate through each image in the batch
    $(\mathbf{x}_k, \mathbf{y}_k) \leftarrow \mathcal{B}_t[k]$
    $\text{Features}_k \leftarrow f_{\text{backbone}}(\mathbf{x}_k)$
    $\text{Proposals}_k, l_{\text{RPN}}(k) \leftarrow f_{\text{RPN}}(\text{Features}_k, \mathbf{y}_k)$ ▷ Loss for RPN on image k
    $\text{SampledProposals}_k \leftarrow \text{FilterAndSampleProposals}(\text{Proposals}_k, \mathbf{y}_k)$
    $l_{\text{Head}}(k) \leftarrow 0$
    for each proposal p in $\text{SampledProposals}_k$ **do**
      $\text{RoIFeatures}_p \leftarrow \text{RoIAlign}(\text{Features}_k, p.\text{coords})$
      $\text{class\_logits}_p, \text{box\_deltas}_p \leftarrow f_{\text{head\_branches}}(\text{RoIFeatures}_p)$
      $l_{\text{cls}}(p), l_{\text{reg}}(p) \leftarrow \text{ComputeProposalLosses}(\text{class\_logits}_p, \text{box\_deltas}_p, p, \mathbf{y}_k)$
      $l_{\text{Head}}(k) \leftarrow l_{\text{Head}}(k) + \lambda_1 l_{\text{cls}}(p) + \lambda_2 l_{\text{reg}}(p)$ ▷ Accumulate head losses
    end for
    if $|\text{SampledProposals}_k| > 0$ **then**
      $l_{\text{Head}}(k) \leftarrow l_{\text{Head}}(k)/|\text{SampledProposals}_k|$ ▷ Normalize head loss
    end if
    per_sample_losses$[k] \leftarrow l_{\text{RPN}}(k) + l_{\text{Head}}(k)$ ▷ Total loss for image k
  end for
  return per_sample_losses
end procedure

8.3. Illustrative Pseudocode: Accessing Batch vs. Per-Sample Loss

The difference in accessibility can be further illustrated with simplified pseudocode. Algorithm 2 contrasts the typical workflow for obtaining batch loss versus the more involved process for per-sample losses, especially reflecting the complexities of tasks like object detection.

Algorithm 2 illustrates that 'GetBatchLossStandard' relies on the framework's internal aggregation to produce a single scalar. In contrast, 'GetPerSampleLossesComplex' (simulating object detection) requires explicitly iterating through each image, re-implementing or exposing the multi-stage loss calculation (RPN loss, proposal sampling, RoI feature extraction, head losses, and their weighted aggregation) to obtain an individual l_k^{task}. This significantly increases complexity compared to merely setting 'reduction='none'' in a simpler loss function.

8.4. Implementation Complexity and Maintainability

Beyond direct computational costs or conceptual challenges in defining $l_i(t)$, the integration of per-sample loss extraction and its subsequent utilization invariably entails increased engineering effort and long-term maintenance burdens. Altering standard training scripts, which are typically well-tested and optimized for scalar loss workflows, introduces potential for errors and diverges from established community practices. More critically, the logic to correctly define, isolate, or appropriately weigh components to form a meaningful $l_i(t)$ is inherently task-dependent and often architecture-specific, as exemplified by object detection (Sec. 8.2) and also noted for other complex scenarios such as certain Semi-Supervised Learning [46] and YOLO [39] frameworks. This necessity for bespoke code for each new application or significant model change curtails generalizability and reduces the portability of such per-sample loss-based techniques across dif-

ferent research codebases or evolving framework versions. Consequently, custom logic for per-sample loss handling adds a considerable maintenance overhead, as changes in underlying libraries or model architectures may necessitate revisiting and adapting this specialized code.

In summary, the path to obtaining and utilizing per-sample losses $l_i(t)$ is often laden with practical challenges. These range from navigating framework abstractions and defining meaningful scalars for complex tasks like object detection, to the intricacies of disentangling contributions in interactive loss settings, and the general increase in implementation complexity and maintenance. These cumulative difficulties provide strong motivation for developing alternative sample importance measures, like BLS, that can effectively leverage the universally accessible mean batch loss.

9. Efficiency Metric Justification.

Table 9. Computational overhead of BLS with InfoBatch and SeTa (1M samples, NVIDIA RTX 3090 GPU).

Method	Overhead	R18 P/B 734.1s	R50 P/B 2122.4s
InfoBatch	0.236s	0.03%	0.01%
BLS-InfoBatch	0.253s	0.03%	0.01%
SeTa	10.001s	1.4%	0.5%
BLS-SeTa	10.021s	1.4%	0.5%

Reproducing wall-clock time reductions for dynamic pruning is notoriously challenging due to hardware and system dependencies. We therefore primarily report the **percentage of data pruned (Pruned %)** as our efficiency metric. This choice is validated by the minimal computational overhead. Table 9 demonstrates that integrating BLS into existing pruning methods (InfoBatch, SeTa) introduces negligible additional overhead (< 0.02s for 1M samples) compared to the original methods. More importantly, the total overhead of these BLS-enhanced pruning methods remains extremely low relative to backbone processing time for representative models like ResNet18 and ResNet50. This minimal overhead ensures compute positivity [11, 54], where the cost of pruning is vastly outweighed by the savings from data reduction (at least $> 20\%$ in our experiments). Furthermore, for complex models involving multiple components beyond a single backbone (e.g., ViECap with GPT2), the relative pruning overhead would be even smaller. Thus, Pruned % serves as a robust, hardware-agnostic, and easily comparable indicator of the potential computational savings offered by BLS-guided pruning.

10. Limitations

While BLS offers significant advantages in simplicity and applicability, its current operational scope has some considerations. Primarily, BLS functions as a scoring mechanism. When integrated into existing dynamic pruning frameworks like InfoBatch [36] or SeTa [54], it replaces their per-sample loss-based scoring but does not inherently alter their multi-epoch iterative pruning schedules or curriculum designs. Thus, while BLS enables these frameworks to operate without direct per-sample loss access, the overall training process duration remains influenced by the epoch-dependent nature of the specific pruning strategy and BLS itself.