A Negative Result on Gradient Matching for Selective Backprop

Lukas Balles[1] **Cedric Archambeau**[2*] **Giovanni Zappella**[1]
[1]Amazon Web Services [2]Helsing
lukas.balles@gmail.com, cedric.archambeau@helsing.ai, zappella@amazon.de

Abstract

With increasing scale in model and dataset size, the training of deep neural networks becomes a massive computational burden. One approach to speed up the training process is *Selective Backprop*. For this approach, we perform a forward pass to obtain a loss value for each data point in a minibatch. The backward pass is then restricted to a subset of that minibatch, prioritizing high-loss examples. We build on this approach, but seek to improve the subset selection mechanism by choosing the (weighted) subset which best matches the mean gradient over the entire minibatch. We use the gradients w.r.t. the model's last layer as a cheap proxy, resulting in virtually no overhead in addition to the forward pass. At the same time, for our experiments we add a simple random selection baseline which has been absent from prior work. Surprisingly, we find that both the loss-based as well as the gradient-matching strategy fail to consistently outperform the random baseline.

1 Introduction

Deep learning models excel in a variety of applications, from natural language processing to computer vision. With increasing scale in model and dataset size the training of such models becomes a massive computational burden. This holds in particular for large *foundation models* trained on massive amounts of data. Thus, the research community is constantly striving to improve the efficiency of deep model training in various ways.

One notable approach is *Selective Backprop* (Jiang et al., 2019). This technique performs a forward pass on a minibatch of data to obtain a loss value for each data point. It then restricts the backward pass to a *subset* of that minibatch, prioritizing points with high loss. The intuition is that the gradient contributions of low-loss examples are small and therefore do not significantly influence the training process. Since, according to commonly accepted estimates (e.g., see the estimates made by DeepSpeed (2023)), the backward pass constitutes the roughly two thirds of the computational effort of a gradient computation, dropping a portion of the data after the forward pass can significantly speed up the training process.

In this work, we build on this approach and seek to improve the mechanism by which the subset of the minibatch is selected. Instead of a simple prioritization of high-loss examples, we aim to select the subset which best preserves the mean gradient over the entire minibatch. We use the gradients with respect to the model's last layer as a cheap and easy-to-compute proxy for the full gradients. A sparse approximation algorithm may then be used to *match* the mean gradient using a small, possibly weighted, subset of the minibatch. This procedure can be implemented in a highly-efficient manner adding only minuscule overhead in addition to the forward pass.

This has the potential to result in more informative subsets. For example, if a minibatch contains two near duplicates with high loss, a loss-based heuristic will likely select both instances for backpropa-

* Work done while at Amazon Web Services.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

gation. Our matching approach will be aware of the redundancy and select only one of them. It can also assign that instance a higher weight to account for the gradient contribution of its duplicate.

We evaluate our method on various image and text datasets using different architectures and optimization methods and find improvements over the loss-based sampling strategy of Jiang et al. (2019). At the same time, we add a simple baseline that randomly selects a portion of the minibatch and was missing from the original work of Jiang et al. (2019). Depending on the experimental setup, this is simply equivalent to training with a smaller batch size for the same number of steps or with the same batch size using fewer steps and an accelerated learning rate decay schedule. In our experiments, Selective Backprop with either selection strategy fails to beat this simple baseline. Our results confirm some of the findings in Kaddour et al. (2023) but we do not restrict our evaluation to model pre-training and we provide a quantification of the models performance with higher granularity.

2 Selective Backprop

Selective Backprop performs a forward pass of the model to obtain a loss value for each data point. A point with loss $l \in \mathbf{R}$ is then included in the backward pass with a probability $p(l) = \text{CDF}(l)^{\beta}$, with the cumulative distribution function (CDF) of loss values across the *entire* dataset as well as selectivity parameter $\beta > 0$. Larger values of β imply a smaller selection probability. Since the exact CDF would be too costly to recompute in each steps, Jiang et al. (2019) approximate it by storing the R most recent loss values seen during training.

The original paper proposed to keep performing forward passes and stochastically selecting points using $p(l)$ until m points have been selected. Then a backward pass is performed. This leads to a variable number of forward-propagated points for each size-m-minibatch making it to the backward pass and is cumbersome to implement. A slight variation of the method forward-propagates a fixed number of M points, from which m are selected. In this case, the CDF can be approximated using the M loss values and we set β to be the inverse of the desired fraction, i.e., $\beta = M/m$. This is the variant implemented in the popular `composer` library[1] and we adopt this variant here.

An ideal implementation of Selective Backprop would cache the intermediate activations from the initial forward pass and reuse them in the backward pass. As Jiang et al. (2019) note, this is currently infeasible to implement in deep learning frameworks. Therefore, they run a separate forward-backward pass on the selected subset, which we adopt for our experiments here. [2] Jiang et al. (2019) note that the overhead of the initial *selection* pass may be mitigated by reducing the floating point precision.

Since a backward pass is roughly twice as expensive as a forward pass (Jiang et al., 2019), selective backprop with an extra selection pass incurs cost proportional to $\frac{M}{3} + m$, while backpropagating the full batch costs M. Hence, we need to choose $m < \frac{2}{3}M$ to achieve cost savings.

3 Related Work

While Selective Backprop "prunes" a given minibatch, another line of work seeks to sample more informative minibatches from the training set in the first place. Multiple works (Alain et al., 2015; Loshchilov & Hutter, 2015; Katharopoulos & Fleuret, 2017; Csiba & Richtárik, 2018) use importance sampling strategies to assemble minibatches. These methods set the probability of selecting an example to a quantity proportional to an "importance score". Zhang et al. (2017) use determinantal point processes to sample *diverse* minibatches.

Coreset methods select small subsets of large datasets with the goal that a model trained on the coreset retain, as closely as possible, the performance of a model trained on the entire dataset. One of the criteria proposed by Paul et al. (2021) is to select the data points having the highest gradient norm,

[1] `https://github.com/mosaicml/composer`

[2] There may also be a (subtle) possible advantage to a separate forward pass. The caching of intermediate activations is the most memory-intensive aspect of deep model training and, as a consequence, GPU memory limits the minibatch size that one can use. Deactivating activation caching allows using a larger-than-usual minibatch size during the initial forward pass. Combined with our a smart subset selection strategy, the resulting gradient estimate may become more accurate than what one could achieve with a randomly-sampled minibatch of a size dictated by GPU memory constraints.

either at initialization or after training for a few epochs. The idea of gradient matching has been used for selecting coresets for deep learning models, among others, by Killamsetty et al. (2021). They also propose to use last-layer gradients as cheap approximations for full gradients, which we adopt in the present work. Our proposed methods can be seen as applications of a gradient matching-based coreset methods to minibatches. While coreset selection is done *once*, we select subsets of each minibatch, demanding extreme efficiency.

4 Method

The motivation for our method is simple. Assume we are given a minibatch of size M but only want to backpropagate $m < M$ points. Ideally, we would like to choose the subset which best approximates the *mean* gradient over the entire minibatch. Of course, identifying that exact subset would require computing all individual gradients first, which would render the whole endeavor meaningless. Instead, we use a cheap *proxy* for these gradients, namely, the gradients with respect to the parameters of the *last (linear) layer* of the model, which can be obtained at minuscule overhead in addition to the forward pass.

Denote the last-layer gradient of the i-th data point as g_i and their mean as $\bar{g} = \frac{1}{M} \sum_{i=1}^{M} g_i$. The *weighted* subset of size $m < M$ which best preserves the mean gradient is given by

$$\min_{w \in \mathbf{R}^M} \left\| \sum_{i=1}^{M} w_i g_i - \bar{g} \right\|^2 \quad \text{s.t.} \quad \|w\|_0 \leq m, \tag{1}$$

where $\| \cdot \|_0$ denotes the ℓ_0-pseudonorm which counts non-zero elements. Eq. (1) is a cardinality-constrained quadratic optimization problem, which is NP-hard. However, good approximate solutions can be obtained with greedy algorithms such as orthogonal matching pursuit (OMP; Mallat & Zhang, 1993), which we briefly introduce in Appendix A.

We replace the loss-based sampling strategy in Selective Backprop by a weighted subset given by an approximate OMP solution to Eq. (1). Note that OMP can be implemented in a Gram-matrix variant, requiring only access to inner products $g_i^T g_j$. This is *crucial* to make our algorithm efficient, since the Gram matrix of the last-layer gradients can be computed efficiently with minuscule overhead compared to the forward pass, as we will explain in §4.1. The full algorithm is then described in §4.2.

4.1 Obtaining Last-Layer Gradients

To facilitate our subset selection, we first need to extract the last-layer gradients. As noted above, and discussed in detail in Appendix A, it suffices to compute the matrix $K \in \mathbf{R}^{M \times M}$ with $K_{ij} = g_i^T g_j$. In the following, we explain how this Gram matrix of last-layer gradients can be computed *implicitly* at virtually no overhead compared to the forward pass.

Assume the last layer has input dimension D and output dimension C and denote its inputs for each data point as $h_i \in \mathbf{R}^D$. Let $W \in \mathbf{R}^{C \times D}$ be the weight matrix and $b \in \mathbf{R}^C$ the bias vector of the last layer, and let $L(\hat{y}, y)$ denote the loss function. With that, we can define the loss for each data point as a function of the last layer's parameters:

$$\ell_i(W, b) := L(W h_i + b, y_i). \tag{2}$$

Denote by $p_i = \nabla L(W h_i + b, y_i) \in \mathbf{R}^C$ the gradient of the loss w.r.t. the model output (its first argument). With that, the gradients of the loss w.r.t. W and b can be written as

$$\nabla_W \ell_i(W, b) = p_i h_i^T, \quad \nabla_b \ell_i(W, b) = p_i, \tag{3}$$

and the inner products are given by $K_{ij} = \mathbf{vec}(p_i h_i^T)^T \, \mathbf{vec}(p_j h_j^T) + p_i^T p_j$, where $\mathbf{vec}(\cdot)$ denotes the vectorization of a matrix. We show in Appendix B that this can be computed *implicitly* as $K_{ij} = (h_i^T h_j)(p_i^T p_j) + p_i^T p_j$. Hence, if we have access to the batch matrices $H = [h_1 \cdots h_M]^T \in \mathbf{R}^{M \times D}$ and $P = [p_1 \cdots p_M]^T \in \mathbf{R}^{M \times C}$, we can compute K as

$$K = H H^T \circ P P^T + P P^T \tag{4}$$

where $\circ$ denotes the elementwise product between matrices. Computing K in this implicit fashion drastically reduces the computational cost and memory footprint compared to computing the individual last-layer gradients themselves.

In terms of a practical implementation in a deep learning framework, this requires (i) caching the last-layer inputs H during the forward pass, and (ii) computing gradients P of the loss w.r.t. the model outputs. The computational cost of the latter typically pales relative to the forward pass. The following code snippet gives the gist of a PyTorch implementation.

```
H = base_model(X)
output = output_layer(H)
loss = loss_fn(output, y)
P = torch.autograd.grad(loss, output)[0]
K = P @ P.T * (H @ H.T + 1.)
```

Full PyTorch code for our method may be found in the Appendix B.2.

4.2 Full Algorithm

As mentioned above, we follow the general strategy of *Selective Backprop* while replacing the loss-based sampling with our gradient-matching selection. Algorithm 1 provides pseudocode. We extract the Gram matrix of the last-layer gradients in an "extended" forward pass as described in the previous section. We then use the Gram-based variant of OMP, which requires the Gram matrix K as well as the vector $t \in \mathbf{R}^M$ with $t_i = g_i^T \bar{g}$, see Appendix A. The latter can easily be computed by taking row-wise mean over K. OMP returns a set of active indices $I \subset [M]$ with $|I| = m$ as well as a vector of associated weights $\gamma \in \mathbf{R}^m$. These weights might not sum to 1, resulting in a scaled gradient. We have found it to be beneficial to normalize these weights.[3]

In Eq. (1), we do not enforce any constraints on the weights. In principle, it would be possible for OMP to assign negative weights, even though that seems unlikely and did not occur in our experiments. As a safeguard, one can simply clip the weights at zero. Alternatively, one could employ algorithms for non-negative matching pursuit, which we leave for future work. In a similar vein, one might consider regularizing the weights to avoid over-reliance on individual data points, which we did not pursue further at this time.

Algorithm 1 Gradient Matching Minibatch

input Minibatch $X \in \mathbf{R}^{M \times D_{\text{in}}}$, $y \in \mathbf{R}^{M \times D_{\text{out}}}$, desired subset size m,
$K = \texttt{extended\_forwad\_pass}(X, y)$
$t = \frac{1}{M} \sum_{i=1}^{M} K_{:,i} \in \mathbf{R}^M$
$I, \gamma = \texttt{OMP}(K, t, m)$
Normalize $\gamma \leftarrow m\gamma / \|\gamma\|_1$.
output I, γ

5 Experiments

We now present an empirical evaluation of our method on various deep learning tasks. We compare our gradient matching approach to the original loss-based sampling strategy of Jiang et al. (2019). We also add a simple baseline of randomly selecting subsets without replacement, which could be see as an equivalent for having a smaller batch size.

Models and Datasets We run experiments on a total of five dataset/model combinations. We train a ResNet-18 (He et al., 2016) on both CIFAR-10 and CIFAR-100 (Krizhevsky et al., 2009), a WideResNet-16-4 (Zagoruyko & Komodakis, 2016) on the SVHN dataset (Netzer et al., 2011), a WideResNet-28-2 (Zagoruyko & Komodakis, 2016) on a 32×32px variant of ImageNet (Chrabaszcz et al., 2017), as well as fine-tuning a pretrained Bert model (Devlin et al., 2018) on the IMDB dataset (Maas et al., 2011).

Training Settings For each model and dataset, we chose default settings with a base batch size as well as optimizer and learning rate schedule. Details may be found in Appendix C. The training budget and learning rate schedules are defined in number of epochs, which we count in terms of

[3]We adopt the convention that non-weighted methods shall correspond to a weight vector containing ones, summing to m. Therefore, we multiply the weights by m in Alg. 1.

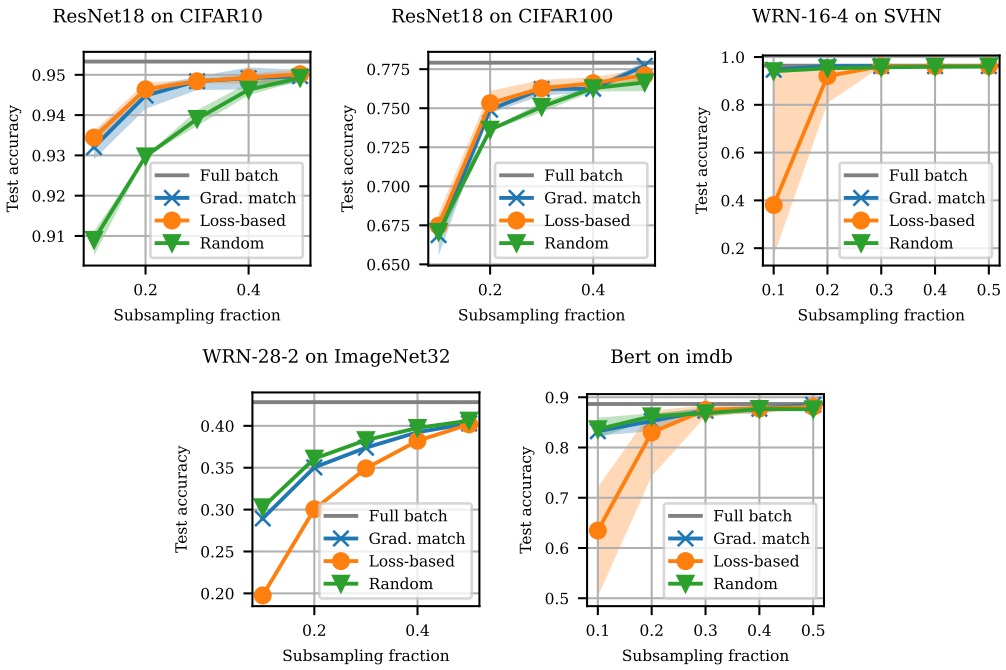


Figure 1: Maximal test accuracy achieved by each subsampling strategies at varying subsampling fractions. Results are averaged over three random seeds and the shaded area spans minimal/maximal values observed.

the *forward* passes, as done by Jiang et al. (2019). This results in a reduction of the total number of backpropagated data points by the subsampling fraction, which we denote as ρ here. For the batch size, we experimented with two settings. *Fixed batch size:* We always use the same base batch $M \equiv M_{\text{base}}$ size during the forward pass, irrespective of the subsampling fraction. Consequently, for all ρ, we use the same number of update steps but different effective batch sizes $m = \rho M_{\text{base}}$. *Scaled batch size:* We scale the batch size used during the forward pass as $M = M_{\text{base}}/\rho$, such that $m \equiv M_{\text{base}}$ irrespective of ρ. Consequently, all ρ use the same effective batch size but the number of update steps is reduced by a factor of ρ. The latter strategy is in line with the work of Jiang et al. (2019), who always collect points until M_{base} is reached. Since the results did not differ substantially, we only report results for the fixed batch size, which we believe is more in line with how such a method would be used in practice. Results for the scaled batch size can be found in Appendix C.

5.1 Main Results

Figure 1 depicts our main results. In addition to the loss-based strategy of Jiang et al. (2019), we add our gradient-matching approach as well as the random baseline. The plots show the test accuracy achieved by each method for different values of the subsampling fraction ρ. We see that the gradient matching approach consistently matches or outperforms the loss-based sampling strategy, especially for lower subsampling fractions. However, either method fails to consistently outperform the simple random baseline.

In the experiments above, all methods use the same learning rate schedule and are assigned the same budget. Since the learning rate has been tuned to the base batch size, one might worry that this skews the results. In Appendix C, we tune the learning rate separately for each method and find similar results.

5.2 Results With Label Noise

Jiang et al. (2019) note as a potential shortcoming of their method that it might be sensitive to outliers or mislabeled examples. Since such data points tend to have high loss, the loss-based selection will oversample them. We repeated our experiments while adding label noise, i.e., we reassign random

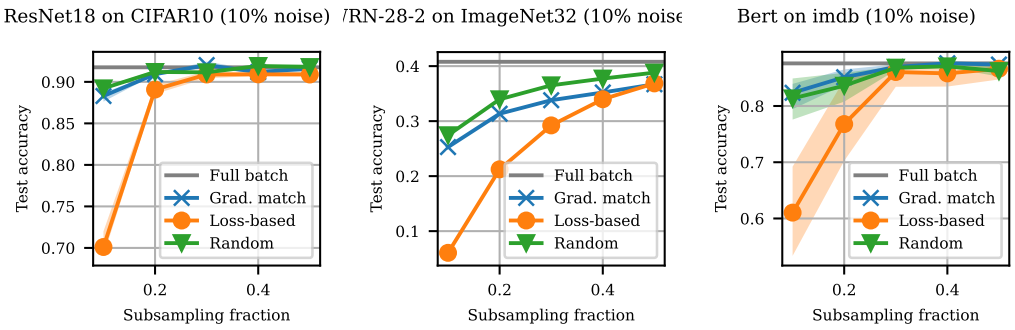


Figure 2: Results with 10% label noise. Maximal test accuracy achieved by each subsampling strategies at varying subsampling fractions. Results are averaged over three random seeds and the shaded area spans minimal/maximal values observed.

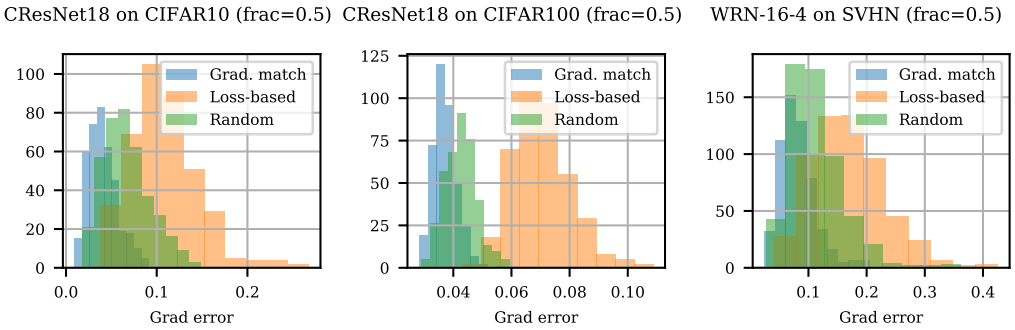


Figure 3: Histogram of squared errors of the gradient estimate for different subsampling strategies.

labels to a portion of the training data. Figure 5.2 shows results with 10% label noise on three datasets, additional results may be found in Appendix C. We see that the loss-based strategy suffers significantly compared to the case without label noise.

5.3 Quality of the Gradient Estimate

One goal of selective backprop should be to achieve better gradient estimates than a randomly sampled minibatch of the same size. Here we compare the quality of the gradient estimate for the different subsampling strategies. Given model weights, we compute the true gradient over the entire datasets. Then we sample a number of minibatches, subsample each with the respective strategy, and compute the squared L_2 distance to the full-dataset gradient. Figure 3 shows histograms for three datasets with gradients computed at a random initialization of the model. We see that the gradient matching technique *does* achieve lower gradient errors than random subsampling while the loss-based sampling strategy even inflates the error.

6 Conclusion

We have investigated a gradient-matching approach to improve the selection mechanism in the *Selective Backprop* framework of Jiang et al. (2019). This technique can outperform the existing loss-based sampling strategy, especially in the presence of label noise. Moreover, we have shown that the gradient approximation obtained using this method is better than the one obtained using a random sample of data. Nevertheless, our experiments show that Selective Backprop fails to ouperform random selection.

References

Alain, G., Lamb, A., Sankar, C., Courville, A., and Bengio, Y. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

Chrabaszcz, P., Loshchilov, I., and Hutter, F. A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv:1707.08819*, 2017.

Csiba, D. and Richtárik, P. Importance sampling for minibatches. *The Journal of Machine Learning Research*, 19(1):962–982, 2018.

DeepSpeed. Flop measurement in the DeepSpeed profiler. `https://www.deepspeed.ai/tutorials/flops-profiler/#flops-measurement`, 2023.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Jiang, A. H., Wong, D. L.-K., Zhou, G., Andersen, D. G., Dean, J., Ganger, G. R., Joshi, G., Kaminksy, M., Kozuch, M., Lipton, Z. C., et al. Accelerating deep learning by focusing on the biggest losers. *arXiv preprint arXiv:1910.00762*, 2019.

Kaddour, J., Key, O., Nawrot, P., Minervini, P., and Kusner, M. J. No train no gain: Revisiting efficient training algorithms for transformer-based language models, 2023.

Katharopoulos, A. and Fleuret, F. Biased importance sampling for deep neural network training. *arXiv preprint arXiv:1706.00043*, 2017.

Killamsetty, K., Durga, S., Ramakrishnan, G., De, A., and Iyer, R. Grad-match: Gradient matching based data subset selection for efficient deep model training. In *International Conference on Machine Learning*, pp. 5464–5474. PMLR, 2021.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

Loshchilov, I. and Hutter, F. Online batch selection for faster training of neural networks. *arXiv preprint arXiv:1511.06343*, 2015.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P11-1015`.

Mallat, S. G. and Zhang, Z. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on signal processing*, 41(12):3397–3415, 1993.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. 2011.

Paul, M., Ganguli, S., and Dziugaite, G. K. Deep learning on a data diet: Finding important examples early in training. *Advances in Neural Information Processing Systems*, 34:20596–20607, 2021.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Rubinstein, R., Zibulevsky, M., and Elad, M. Efficient implementation of the k-svd algorithm using batch orthogonal matching pursuit. Technical report, Computer Science Department, Technion, 2008.

Zagoruyko, S. and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Zhang, C., Kjellstrom, H., and Mandt, S. Determinantal point processes for mini-batch diversification. *arXiv preprint arXiv:1705.00607*, 2017.

A Matching Pursuit

Orthogonal matching pursuit (Mallat & Zhang, 1993) approximately solves problems of the form

$$\min_{w \in \mathbf{R}^M} \|Aw - b\|^2 \quad \text{s.t.} \quad \|w\|_0 \leq m, \tag{5}$$

That is, it matches a target signal $b \in \mathbf{R}^D$ by a sparse weighted sum $Aw = \sum_i w_i a_i$ of so-called atoms $a_i \in \mathbf{R}^D$. Solving this problem exactly is NP-hard.

We give a brief description of OMP here, for details refer to the original paper. We represent a subset using a list of indices $I \subset [M]$ and associated weights $\gamma \in \mathbf{R}^{|I|}$. A given I and γ yields an approximation $A_I\gamma$, where A_I denotes the restriction of A to columns I. Matching pursuit greedily adds the element which best matches the residual, i.e., it chooses $k_* = \arg\max_{k \in [M]} a_k^T(b - A_I\gamma)$. After adding a new element to I, orthogonal matching pursuits recomputes the weights γ such as to minimize $\|A_I\gamma - b\|^2$, which results in $\gamma = (A_I^T A_I)^{-1} A_I^T b$.

Both the selection criterion as well as the formula for γ depend solely on inner products $a_i^T a_j$ as well as $a_i^T b$. Therefore, OMP can be performed given only access to the matrix $K = A^T A$ as well as the vector $t = A^T b$.

Algorithm 2 provides pseudo-code. In practice, the inversion of $K_{I,I}$ need not be done from scratch in each iteration. Rather, the algorithm maintains a Cholesky factorization of $K_{I,I}$, which can be updated efficiently upon the addition of a new element. We refer to the work of Rubinstein et al. (2008) for details. For our experiments, we used the implementation in `scikit-learn` (Pedregosa et al., 2011).

Algorithm 2 Orthogonal Matching Pursuit (Gram variant)

input $K \in \mathbf{R}^{M \times M}$, $t \in \mathbf{R}^M$, desired subset size m
  Initialize $\alpha = t$
  while $|I| < m$ **do**
    Select $k = \arg\max_{i \in [M]} \alpha_i$
    $I \leftarrow I \cup \{k\}$
    $\gamma \leftarrow K_{I,I}^{-1} t_I$
    $\alpha \leftarrow \alpha - K_{:,I}\gamma$
  end while
output I, γ

B Method Details

B.1 Mathematical Details

In §4.1, we have used the identity

$$\mathbf{vec}(p_1 h_1^T)^T \, \mathbf{vec}(p_2 h_2^T) = (h_1^T h_2)(p_1^T p_2) \tag{6}$$

for $p_1, p_2 \in \mathbf{R}^C$ and $h_1, h_2 \in \mathbf{R}^D$.

Proof. We denote elements of the vectors above using double indices, e.g., $p_{1i} = [p_1]_i$.

$$\begin{aligned} \mathbf{vec}(p_1 h_1^T)^T \, \mathbf{vec}(p_2 h_2^T) &= \sum_{i,j} [p_1 h_1^T]_{ij} [p_2 h_2^T]_{ij} = \sum_{i,j} (p_{1i} h_{1j})(p_{2i} h_{2j}) \\ &= \left(\sum_i p_{1i} p_{2i} \right) \left(\sum_j h_{1j} h_{2j} \right) = (p_1^T p_2)(h_1^T h_2). \end{aligned} \tag{7}$$

□

B.2 PyTorch Code

In the following, we show PyTorch code for our implementation of gradient matching for selective backprop.

```
import numpy as np
import torch


class GradMatchSelector(torch.nn.Module):

    def __init__(self, model, output_layer, loss_fn, k):
        self._model = model
        self._output_layer = output_layer
        self._loss_fn = loss_fn
        self._k = k

        # Cache inputs to output layer during forward pass.
        self._last_layer_input = None
        def cache_hook(module, inputs, output):
            self._last_layer_input = inputs[0]
        self._hook = output_layer.register_forward_hook(cache_hook)

    def _compute_last_layer_grad_gram_matrix(self, X, y):
        output = self._model(X)
        loss = self._loss_fn(output, y).sum()
        output_grads = torch.autograd.grad(loss, output)[0]
        PPt = output_grads @ output_grads.T
        HHt = self._last_layer_input @ self._last_layer_input.T
        return PPt * (HHt + 1.)

    def forward(self, X, y):
        Gram = self._compute_last_layer_grad_gram_matrix(X, y)
        # Use existing numpy-based implementation of OMP.
        Gram = Gram.cpu().numpy().astype(np.float64)
        Xy = Gram.sum(1)
        weights, idx, _ = _gram_omp(Gram, Xy, self._k)
        # Convert weights to torch and normalize.
        weights = torch.from_numpy(weights).float().to(X.device)
        weights = torch.clamp(weights, min=0.)
        weights /= weights.sum()
        weights *= len(idx)
        return idx, weights
```

B.3 Computational Cost

Table 1 shows the wall-clock time per training epoch for our method compared to the purely loss-based Selective Backprop. The current implementation of our approaches adds a small overhead (less than 5%). A more detailed profiling of the run time indicates that most of this overhead stems from the GPU $\rightarrow$ CPU transfer of the Gram matrix K, which could be avoided with a PyTorch implementation of (Gram) orthogonal matching pursuit. We leave that for future work.

Table 1: Wall-clock time [s] per epoch for vanilla selective backprop (SB) and gradient matching (GM).

dataset	method fraction	GM	SB
CIFAR10	0.1	20.70	19.69
	0.3	27.41	26.77
	0.5	36.74	35.68
CIFAR100	0.1	20.37	19.95
	0.3	27.05	26.56
	0.5	36.28	35.21

C Experimental Details and Additional Results

C.1 Hyperparameter Settings

The experiments used the following hyperparameter settings:

- CIFAR-10(0): We use SGD with Nesterov momentum of 0.9, weight decay of $5 \cdot 10^{-4}$. We train for 200 epochs, starting with a learning rate of 0.1 and decaying by a factor of 0.2 after 60, 120 and 160 epochs. The base batch size is 128.
- SVHN: We use SGD with Nesterov momentum of 0.9, weight decay of $5 \cdot 10^{-4}$. We train for 80 epochs with a Cosine Annealing schedule. The initial learning rate is 0.01 and the base batch size is 128.
- ImageNet-32: We use SGD with momentum of 0.9, weight decay of $5 \cdot 10^{-4}$. We train for 40 epochs, starting with a learnign rate of 0.01 and decaying by a factor of 0.2 after 10, 20 and 30 epochs. The base batch size is 128.
- IMDB: We use AdamW with default values for β_1 and β_2 and weight decay of $5 \cdot 10^{-4}$. We fine-tune the pretrained Bert model for 3 epochs using a constant learning rate of $2 \cdot 10^{-5}$. The base batch size is 32.

C.2 Results with Scaled Base Batch Size

Figure 4 shows results for the scaled base batch size, as explained in Section 5. That is, the forward pass uses a batch size of $M = M_{\text{base}}/\rho$. The results do not differ substantially from the alternative batch size setting presented in the main text. All methods achieve lower accuracy on average, suggesting that more steps with a smaller batch size are preferable to fewer steps with a large batch size.

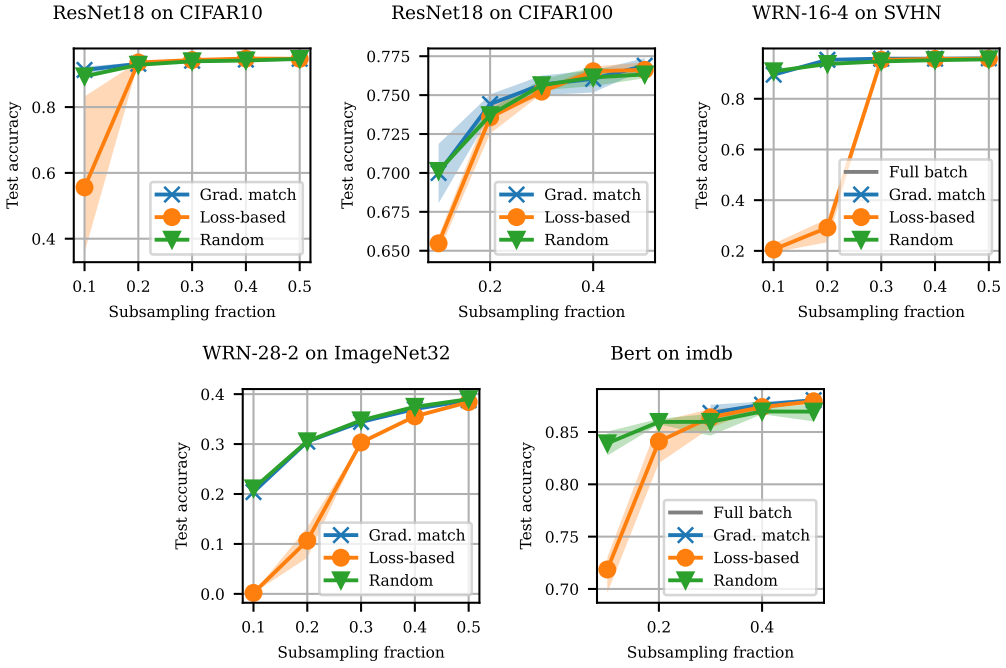
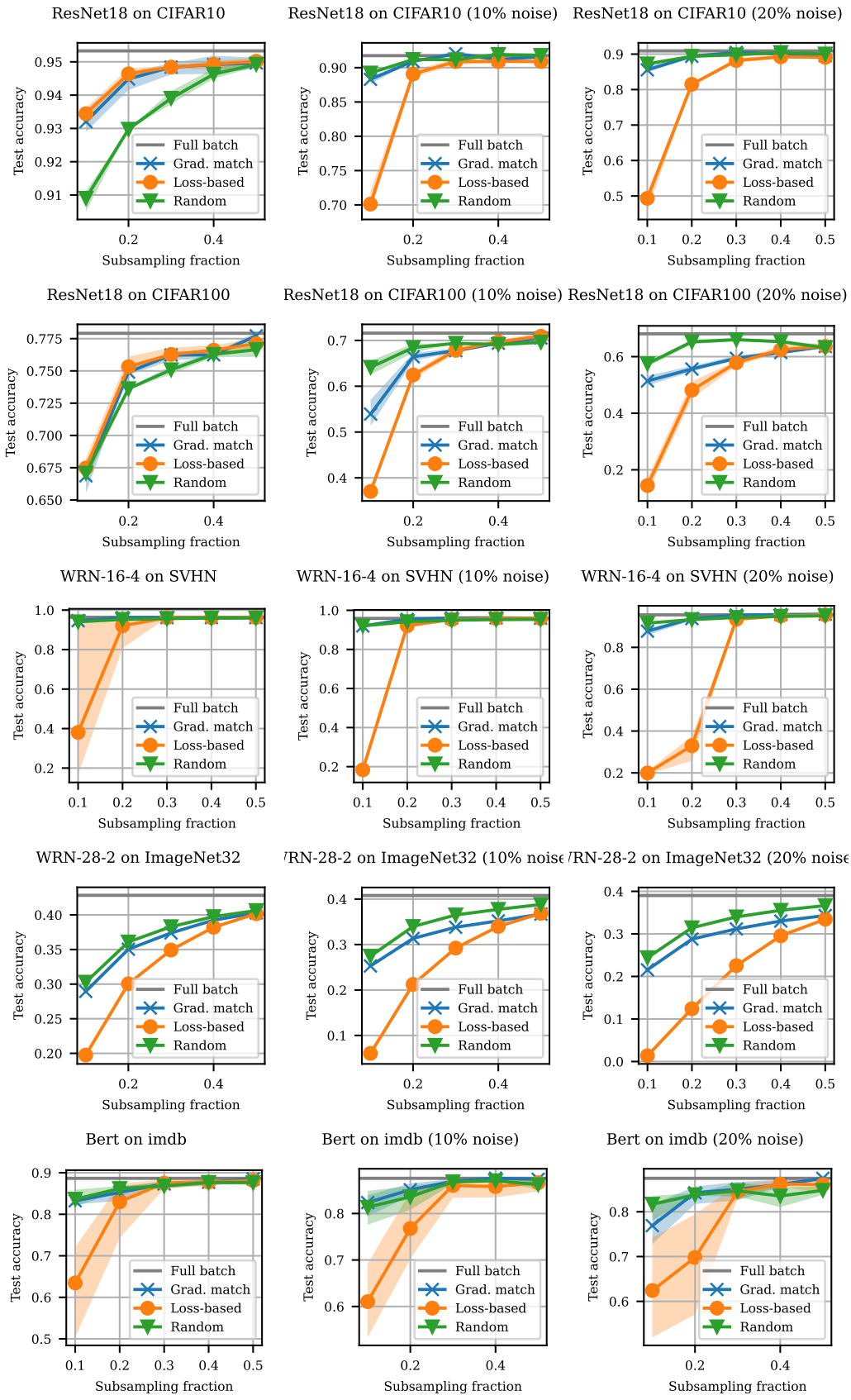


Figure 4: Results under the base learning rate schedule with a scaled base batch size. We show the maximal test accuracy achieved by each method at varying subsampling fractions.

C.3 Additional Results with Label Noise

Figure C.3 shows results with various levels of label noise.


ResNet18 on CIFAR10
ResNet18 on CIFAR10 (10% noise)
ResNet18 on CIFAR10 (20% noise)
ResNet18 on CIFAR100
ResNet18 on CIFAR100 (10% noise)
ResNet18 on CIFAR100 (20% noise)
WRN-16-4 on SVHN
WRN-16-4 on SVHN (10% noise)
WRN-16-4 on SVHN (20% noise)
WRN-28-2 on ImageNet32
RN-28-2 on ImageNet32 (10% noise
RN-28-2 on ImageNet32 (20% noise
Bert on imdb
Bert on imdb (10% noise)
Bert on imdb (20% noise)
Test accuracy
Subsampling fraction
Full batch
Grad. match
Loss-based
Random

C.4 Learning Rate Tuning

In addition to the experiments with the base learning rate schedule, we conducted experiments where learning rate schedules are tuned separate for each method. The rationale is that the subsampling techniques affect the effective batch size used to compute gradients and it is well known the batch size and step size are highly interdependent hyperparameters. Therefore, using a learning rate schedule tuned to the base batch size may skew the results.

To keep the experimental workload manageable, we use a simplified tuning protocol that is based on the base schedule and a "linear scaling rule" for the relationship between batch size and learning rate. Under this rule, originally proposed in the distributed deep learning community, a doubling of the batch size (for randomly sampled batches) would necessitate a doubling of the learning rate as well. Hence, for each method and subsampling fraction, we tune a learning rate factor $\beta \in [0, 1]$ and set the (initial) learning rate to $\alpha = \beta \cdot \alpha^{\text{base}}$. We then use this factor to "stretch" the learning rate schedule by setting the total number of epochs to $T = T^{\text{base}}/\beta$.

One can think of β as the "effective" downsampling fraction of a method. If the downsampling were done uniformly at random, we would expect a value of $\beta = m/M$ to be optimal, as per the linear scaling rule. For smarter subset selections strategies, we hope to achieve values of β that are significantly larger than that. We tune β on a logarithmic grid spanning the interval 10^{-1} to 10. A value of 1 recovers the base case.

Results are depicted in Fig. 5 in the form of a sensitivity plot. We show results for $\rho = 0.3$ here; additional values may be found in Appendix C.

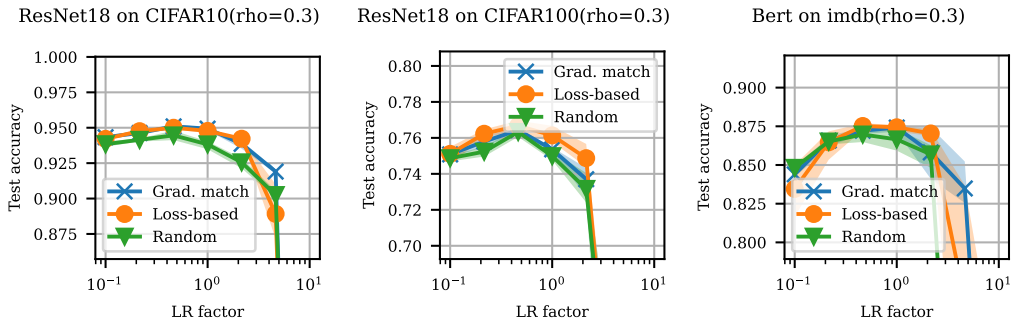


Figure 5: Sensitivity when tuning the learning rate. We show the maximal test accuracy achieved by each method as a function of β, which stretches the learning rate as well as the budget.