OPTIMIZING TRAINING DATA WITH METAGRADIENT DESCENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Curating a training dataset from a mass of unfiltered data is a necessary and influential step in any large-scale machine learning pipeline. We frame dataset selection as an optimization problem over per-example data counts and solve it with metagradient descent (MGD). At a high level, our algorithm operates as follows: we start with a randomly chosen set of training data, then iteratively update the dataset selection using metagradients with respect to importance weights placed on each training datapoint. To do this at scale, we develop: (1) REPLAY, an algorithm and implementation for computing scalable metagradients using recursive checkpointing and efficient differentiation of individual training steps; (2) a diagnostic for *metasmoothness*, allowing us to modify the training routine to obtain useful gradients; (3) a continuous surrogate for optimizing data counts; and (4) a block-SGD-inspired algorithm for improving data optimization. Combining these four contributions, MGD reaches average scores of 22% on DataComp-small and 40.2% on DataComp-medium, and improves over LESS and full-data training for Gemma-2B instruction tuning on BBH and MMLU. These results show that repeated gradient-based dataset updates can improve both pre-training and instruction-tuning data.

1 INTRODUCTION

Curating a training dataset from a mass of unfiltered data is a necessary and influential step in any large-scale machine learning pipeline. Deciding how to curate such a dataset is a challenging problem that has attracted substantial recent interest (Fang et al., 2022; Abbas et al., 2023; Engstrom et al., 2024; Gadre et al., 2024). Finding the right training setup is difficult, however, as the corresponding design space is overwhelmingly vast. As an example, there are 2^n possible training sets given a pool of n candidate datapoints.

In this work, we take the *optimization* perspective on curating training data. We represent a dataset by the number of times each candidate sample appears in it, and optimize these counts to minimize a trained model's loss on a target distribution. We call this procedure metagradient descent (MGD). Each update retrains on the current dataset, allowing a sample's usefulness to change as the rest of the dataset changes. Making it work requires addressing three obstacles: the memory needed to differentiate through training, non-smooth training trajectories, and discrete data counts. We address these challenges with recursive checkpointing, smoothness interventions, and a continuous surrogate with stochastic block updates respectively.

First, we develop REPLAY, an algorithm and implementation for computing metagradients at scale. By combining recursive checkpointing with efficient differentiation through individual training steps, REPLAY calculates exact metagradients through 325–3,250 steps of CLIP pre-training and 150 steps of LoRA fine-tuning of a 2B-parameter language model. Second, we find that metagradients of standard training routines are not necessarily helpful for optimization, which we connect to non-smoothness of the metaparameter optimization landscape. We diagnose and improve training metasmoothness before optimizing the dataset, then transfer the selected data back to the original model and training routine in every experiment.

We apply this recipe to pre-training and instruction-tuning data, obtaining the following results:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

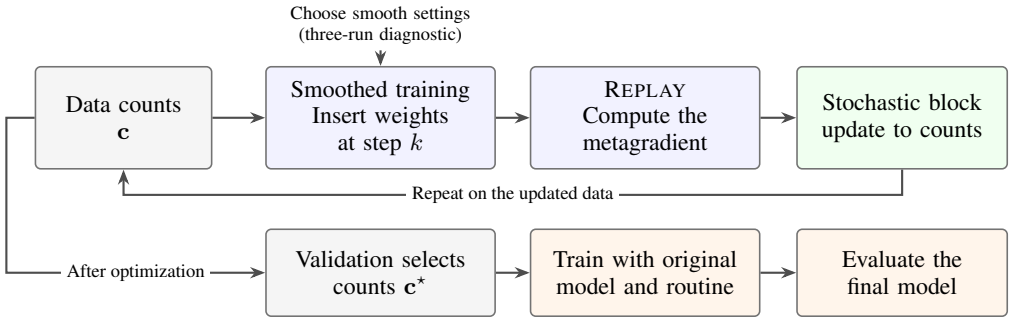


Figure 1: The MGD recipe. We choose smooth training settings, then repeatedly train, compute metagradients, and update data counts. Validation selects the dataset used to train the original model.

- On DataComp-small, MGD reaches an average score of 22%, compared with 18.2% for EcoDatum and 13.2% without filtering. On DataComp-medium, MGD reaches 40.2%, compared with 38.8% for the strongest earlier leaderboard entry.
- On Gemma-2B instruction tuning, MGD achieves 36.7% on BBH and 42.5% on MMLU, improving over LESS and full-data training.

Figure 1 summarizes the dataset-optimization recipe.

2 Optimizing Training Data

2.1 Data Selection as Optimization

The goal of dataset selection is to choose a training data multiset (out of a broad pool of data) that maximizes trained machine learning model performance. Given this goal, dataset selection has a natural interpretation as a combinatorial metaparameter optimization problem. In particular, for a training set of size n, let the metaparameters $\mathbf{c} \in \mathcal{C} := \mathbb{Z}^n_{\geq 0}$ be non-negative data counts representing the number of times each training sample repeats in the training data; the algorithm $\mathcal{A}$ be a standard large-scale learning procedure, which runs on a training set comprising c_i copies of each sample i for $i \in [n]$; and the output function ϕ be the loss of the trained model on a target distribution D. Then, defining $f(\mathbf{c}) := \phi(\mathcal{A}(\mathbf{c}))$, our goal is to find the data counts $\mathbf{c}^*$ that solve

$$\mathbf{c}^* := \arg\min_{\mathbf{c} \in \mathcal{C}} f(\mathbf{c}). \tag{1}$$

From this perspective, any data-selection procedure is an optimization algorithm searching for a dataset that yields low target loss. Searching over a small set of filtering rules explores only a small part of the possible datasets. What we would like to do is somehow efficiently optimize over the whole space of possible datasets.

In many high-dimensional optimization problems, a natural tool to consider is the *gradient*, which allows us to very efficiently make local progress on improving a given candidate solution. To apply gradient-based optimization, we must first design a training routine $\mathcal{B}$ controlled by continuous metaparameters $\mathbf{z}$, such as example weights; Section 2.4 constructs our surrogate for discrete data counts. The objective $F(\mathbf{z}) = \phi(\mathcal{B}(\mathbf{z}))$ trains a model and evaluates its target loss. Its gradient $\nabla_{\mathbf{z}} F$, called a *metagradient*, measures how changing the training setup changes that loss.

For a sufficiently smooth objective, a small step in the negative metagradient direction can reduce the target loss. This suggests iteratively training a model, computing a metagradient, and updating the data. Making this approach practical requires both computing metagradients at scale and obtaining gradients that are useful for optimization, which we address next.

2.2 REPLAY: Scalable Metagradient Computation

To compute a metagradient, we need to differentiate the target loss through the training routine that produced the model. Since training is itself a sequence of differentiable operations, we can in

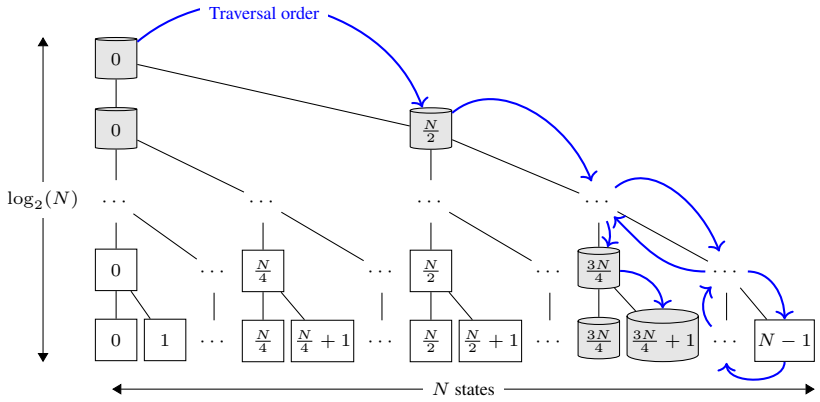


Figure 2: Recursive checkpointing for a binary tree of N optimizer states. Blue arrows show reverse traversal; gray cylinders mark the states currently stored. Recovering each segment from its starting checkpoint reduces storage from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$ states, using $\mathcal{O}(N \log N)$ optimizer steps.

principle apply automatic differentiation (AD) to this computation. The difficulty is that reverse-mode AD saves intermediate values for its backward pass. Applied to the whole training run, it would retain these values across all training steps.

A more efficient method for calculating the metagradient, step-wise AD, leverages the structure of iterative learning algorithms (Werbos, 1990; Maclaurin et al., 2015; Franceschi et al., 2017). Writing a training step as $\mathbf{s}_{t+1} = h_t(\mathbf{s}_t, \mathbf{z})$, reverse-mode AD propagates the derivative of the target loss backwards through these updates. Each step needs the corresponding optimizer state, including the parameters and, for Adam, the gradient moments. Despite improving storage overhead compared to "direct AD", step-wise AD is still too space-intensive at scale. After all, this algorithm saves every optimizer state.

REPLAY modifies step-wise AD to traverse states in less space by exploiting a simple observation: when training is deterministic, one can reinstantiate an optimizer state $\mathbf{s}_t$ by "replaying" training from a fixed point $t' < t$—at the compute cost of $t - t'$ training steps. We fix data ordering, data augmentation, and other training randomness within each metagradient computation so that replay reconstructs the same trajectory. This still permits sampling new randomness for the next dataset update.

Specifically, REPLAY recursively saves and replays training states. For a branching factor b, we split the differentiated trajectory into b segments, save the starting state of each segment, and recursively visit the segments in reverse order. At a leaf, we differentiate through one training step; states are deleted once their segment has been traversed. Figure 2 illustrates this schedule. At each recursion level we retain at most b states and replay $\mathcal{O}(H)$ training steps in total, where H is the differentiated horizon. The $\mathcal{O}(\log_b H)$ levels therefore require $\mathcal{O}(b \log_b H)$ stored states and $\mathcal{O}(H \log_b H)$ step computations. Increasing b trades memory for less recomputation.

Implementing REPLAY at scale. The recursive schedule controls storage across training steps; each step already contains a backward pass, so differentiating it requires backward-over-backward operations whose memory use must also be controlled. For CLIP, we checkpoint the encoders and compute embeddings in minibatches across GPUs (Appendix A.3). We also implement backward-over-backward support for FlashAttention (Dao et al., 2022). These components, together with deterministic replay, enable the metagradient computations in our experiments. Appendix C gives the chain-rule derivation and traversal pseudocode.

2.3 DIAGNOSING AND IMPROVING METASMOOTHNESS

Given a training function F, REPLAY enables us to compute metagradients $\nabla F(\mathbf{z})$ for any setup $\mathbf{z}$. Can we immediately use these metagradients to optimize model training setups? The answer is (generally) no: we find that applying REPLAY to a function F representing a standard model training

and evaluation routine yields metagradients that are often $\pm\infty$-valued and generally unhelpful for optimization. Indeed, previous work has observed similar issues optimizing over even (very) small-scale training (Bengio et al., 1994; Pearlmutter, 1996; Maclaurin et al., 2015).

These failures lead us to examine the metaparameter optimization landscape. In optimization theory, the basic property of a function that controls how effectively it can be optimized with first-order methods is smoothness. Specifically, a function $f(\mathbf{z})$ is β-smooth at a point $\mathbf{z}$ if its gradient ∇f satisfies the property that

$$\|\nabla f(\mathbf{z}) - \nabla f(\mathbf{z}')\| \leq \beta \cdot \|\mathbf{z} - \mathbf{z}'\| \qquad \text{for all } \mathbf{z}', \tag{2}$$

or in other words, if its gradient does not change too quickly around $\mathbf{z}$. For $\beta > 0$, this condition gives the gradient-descent bound

$$f\left(\mathbf{z} - \frac{1}{\beta}\nabla f(\mathbf{z})\right) \leq f(\mathbf{z}) - \frac{1}{2\beta}\|\nabla f(\mathbf{z})\|^2.$$

This guarantee makes β-smoothness a good measure of gradient utility. For our continuous objective F, we refer to smoothness with respect to the training setup as *metasmoothness*.

Empirical metasmoothness. To select and modify training routines, we need a practical measure of metasmoothness. Checking the condition above for all perturbations is infeasible, and using metagradients for this check would entangle the landscape's smoothness with numerical errors in differentiation. We therefore use a finite-difference diagnostic computed from trained parameters, independent of the target loss and of metagradient computation. It checks whether two consecutive perturbations change the parameters in consistent directions.

We compare three training runs of $\mathcal{B}$ with randomness held fixed. For a metaparameter vector $\mathbf{z}$, a direction $\mathbf{v}$, and a small $h > 0$, define

$$\theta_0 = \mathcal{B}(\mathbf{z}), \qquad \theta_h = \mathcal{B}(\mathbf{z} + h\mathbf{v}), \qquad \theta_{2h} = \mathcal{B}(\mathbf{z} + 2h\mathbf{v}), \qquad \mathbf{d} = |\theta_{2h} - \theta_0|.$$

$$\widehat{S}_{h,\mathbf{v}}(\mathcal{B}; \mathbf{z}) = \sum_i \frac{d_i}{\|\mathbf{d}\|_1} \operatorname{sign}((\theta_h - \theta_0)_i) \operatorname{sign}((\theta_{2h} - \theta_h)_i). \tag{3}$$

This measures agreement in the direction of consecutive parameter changes, weighting each coordinate by its variation. It lies in $[-1, 1]$ when $\|\mathbf{d}\|_1 > 0$, and requires no metagradient computation. We use it to choose training settings; it does not certify smoothness of the downstream loss F. Appendix D gives the full definition.

Estimating and improving metasmoothness. We explore a fixed menu of modifications to the training setup and choose the combination that maximizes empirical metasmoothness. An arbitrarily chosen metaparameter suffices to guide this search in our experiments. In practice, we find that this framework allows us to slightly modify learning algorithms in a way that makes them amenable to first-order methods.

Figure 3 illustrates this procedure with a sweep over ResNet-9 training configurations. In addition to "standard" hyperparameters such as batch size and network width playing a role, we find that placing Batch Normalization layers prior to nonlinearities (instead of after) and scaling the final layer output are both crucial to metasmoothness. However, the trade-off is not too severe: the most metasmooth algorithms still achieve near-optimal accuracy.

Smoothing the training optimizer. For data selection, we modify the optimizer used during the metagradient computation. In Adam (Kingma & Ba, 2015), we add a positive $\varepsilon_{\text{root}}$ inside the square root:

$$\theta_{t+1} = \theta_t - \alpha_t \cdot \left(m_t / \left(\sqrt{v_t + \varepsilon_{\text{root}}} + \varepsilon\right) + \lambda\theta_t\right) \tag{4}$$

where m_t and v_t are running estimates of the first and second moments of the gradients, respectively, λ represents weight decay, α represents the learning rate, and ε and $\varepsilon_{\text{root}}$ are hyperparameters to avoid blowup.

Transferring the selected data to the original model. In all our experiments, the smoothness interventions are used to optimize the training data, and the resulting data are transferred back to the original models. We hold the selected dataset fixed and train the original model with its original training routine.

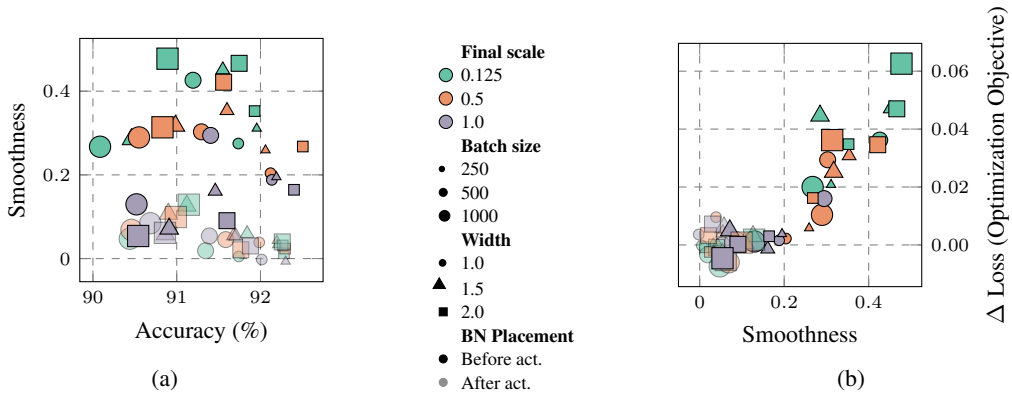


Figure 3: (a) For a variety of training configurations of a ResNet-9 model, we plot metasmoothness (Equation 3) against test accuracy. Strategies such as increasing width, placing batch normalization before activations, and scaling down network outputs consistently improve metasmoothness, at a minor cost to accuracy. (b) Smoother training configurations can be optimized via metagradients more effectively. Spearman's $\rho = 0.756$ across the 52 configurations in (b).

2.4 A CONTINUOUS SURROGATE FOR DATA COUNTS

We now apply these tools to the data counts in Equation 1. Let $\widetilde{\mathcal{A}}$ denote the learning algorithm with the smoothness interventions described above.

The discrete counts $\mathbf{c}$ make $\widetilde{\mathcal{A}}$ non-differentiable, so we relax to a surrogate $\mathcal{A}'_{\mathbf{c}}$ that takes a continuous metaparameter $\mathbf{z} \in \mathbb{R}^n$. The surrogate is parameterized by fixed counts $\mathbf{c}$ and a single training iteration k, and trains exactly as $\widetilde{\mathcal{A}}$ does except at iteration k, where, for an example-wise training loss, it adds an extra term $\sum_{i=1}^{n} z_i \, \ell(x_i; \theta)$ to the batch loss (full case-logic in Appendix A). At fixed counts, $\mathcal{A}'_{\mathbf{c}}$ is the continuous routine $\mathcal{B}$ to which we apply checkpointing and the metasmoothness diagnostic. At $\mathbf{z} = \mathbf{0}$ the surrogate reduces to $\widetilde{\mathcal{A}}$, so the metagradient $\mathbf{g} := \nabla_{\mathbf{z}} \, \phi(\mathcal{A}'_{\mathbf{c}}(\mathbf{z}))\big|_{\mathbf{z}=\mathbf{0}}$ measures the infinitesimal effect on ϕ of weighting each sample at iteration k, including its effect on all subsequent training steps. It is an exact derivative of this surrogate, rather than of the discrete count vector $\mathbf{c}$. For CLIP, the surrogate also accounts for interactions in the contrastive loss (Appendix A.2).

The insertion step k controls the differentiated part of training. Since $\mathbf{z}$ first enters at step k, the preceding training states do not depend on $\mathbf{z}$. We therefore differentiate the suffix from k to the end exactly for this surrogate, using recursive checkpointing over this suffix; the prefix is trained once per dataset update. A later insertion reduces the differentiation cost, but restricts the gradient to the effect of adding weight at that training step.

Selection settings in practice. For DataComp-small, we add $\varepsilon_{\text{root}} = 10^{-17}$ inside Adam's square root and insert the data weights at step $k = 2800$ of 3125 training steps (Table 2). Thus, we differentiate through the final 325 steps. For DataComp-medium, we preserve the relative insertion position $k/T = 0.896$, differentiating through the final 3,250 of 31,250 steps (Appendix A.5); for instruction tuning, we use the final 150 steps (Section 4). The values of $\varepsilon_{\text{root}}$ and k (the step at which metagradients are calculated) were chosen to empirically maximize metasmoothness. For instruction tuning, we jointly select learning rate and $\varepsilon_{\text{root}}$ using the same diagnostic.

2.5 STOCHASTIC BLOCK UPDATES

We then use the metagradient $\mathbf{g}$ to iteratively update our selected dataset. We update data counts as

$$\mathbf{c} \leftarrow \left[\mathbf{c} - \operatorname{sign}(\mathbf{g}) \odot \mathbf{m}\right]_+, \quad \mathbf{m} \sim \text{Bernoulli}(p), \tag{5}$$

where p is a hyperparameter controlling the fraction of sample counts to update. This algorithm resembles a block coordinate descent algorithm (Ortega & Rheinboldt, 2000), with the main difference being that we take signed gradient steps with step size 1 (projected onto non-negative integers)

Algorithm 1: Dataset selection using metagradient descent (MGD).

Input: initial data counts $\mathbf{c} \in \mathbb{Z}^n_{\geq 0}$, final learning algorithm $\mathcal{A}$, target loss ϕ

Hyperparameters: number of dataset updates M, insertion step k, block fraction p, target minibatch fraction q; for DataComp, fixed-size switch step and exchange fraction r

Choose a metasmooth training routine $\widetilde{\mathcal{A}}$ using the diagnostic in Section 2.3

for $t \leftarrow 1, \ldots, M$ **do**

  Sample an active block B (fraction p) and a target minibatch Q (fraction q)

  $\mathbf{z} \leftarrow \mathbf{0}_n$

  Train $\widetilde{\mathcal{A}}$ on $\mathbf{c}$; use REPLAY to compute $\mathbf{g}_B = \nabla_{\mathbf{z}_B} \phi_Q(\mathcal{A}'_{\mathbf{c}}(\mathbf{z}))\big|_{\mathbf{z}=\mathbf{0}}$

  Set $\mathbf{g}_{[n]\setminus B} = 0$

  if *in the fixed-size phase (DataComp)* **then**

    Exchange counts for an r fraction of points in B using the ranked metagradients (Section 2.5)

  else

    $\mathbf{c} \leftarrow [\mathbf{c} - \operatorname{sign}(\mathbf{g})]_+$

  Record the candidate counts and validation performance

Select counts $\mathbf{c}^\star$ by the validation protocol (DataComp: Appendix A.4; IFT: Section 4)

Return $\mathbf{c}^\star$; evaluate $\mathcal{A}(\mathbf{c}^\star)$ under the final training algorithm

to ensure that the counts remain well-defined. As a result, p implicitly controls the algorithm's step size.

Here $[\cdot]_+$ clips negative counts to zero. We use the continuous surrogate to choose a direction for these discrete updates.

This gives a form of stochastic block gradient descent: we sample both the coordinates to update and the target examples used to estimate the objective. For DataComp, rather than selecting $\mathbf{m}$ fully randomly every step, we randomly select a shard comprising fraction p of the data and take steps on all datapoints in the shard (see Appendix A.3). To mitigate overfitting, we also bake a "minibatch fraction" q into our model output function ϕ. For example, if ϕ calculates model loss on the ImageNet train set, each time ϕ is called, we randomly sample fraction q of the ImageNet train set to evaluate on.

In the DataComp-small implementation, we use eight original shards, so $p = 1/8$. Sampling from the original shards allows an example whose count has reached zero to be added back in a later update.

Controlling dataset size. We also observed that always stepping on the sign causes the sizes of the shards to grow over time: stepping based on the sign of the metagradient does not decrease the weight on a positive-metagradient datapoint if its dataweight is already 0, so our steps are biased towards increasing the size of the shards. To combat this blowup, after some number of optimization steps, we choose a fixed shard size and enforce that subsequent steps must not change the size of the shards—the step size thereafter is controlled by hyperparameter r representing the fraction of datapoints in a shard which are incremented. We experimented both with randomly sampling which points are added or removed, and stepping on the datapoints with the top r and bottom r metagradients; the latter seems to give empirically better performance.

The fixed-size phase increments counts with low metagradients and decrements eligible positive counts with high metagradients, preserving the shard's total count. This separates the number of samples exchanged from the total size of the dataset.

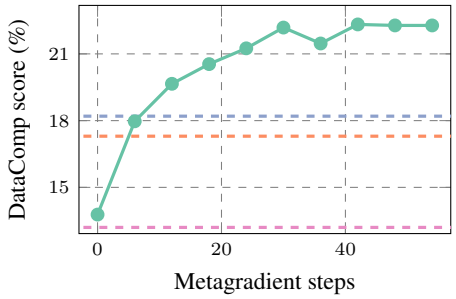
The complete recipe. Algorithm 1 gives the complete procedure. Validation selects the counts used for final training; the application sections give the respective selection protocols.

3 MULTIMODAL PRE-TRAINING DATA: DATACOMP

We evaluate Algorithm 1 on DataComp-small and DataComp-medium (Gadre et al., 2024), a standardized benchmark in which a fixed learning algorithm is run on a data subset chosen by the participant from a fixed candidate pool, and the resulting model is scored against 38 downstream benchmarks. We select data with MGD to minimize loss on a "target set" constructed from the DataComp evaluation tasks (extra samples beyond the DataComp test set; e.g., the ImageNet train split), and select hyperparameters with a held-out validation set.

Selection and final training. We optimize counts on the available candidate pool using the stabilized training routine, then transfer the selected dataset back to the original CLIP model using the official DataComp training and evaluation scripts. The target examples used for selection are separate from the benchmark test examples. The small-scale experiment uses approximately 80% of the candidate pool because some image URLs could no longer be downloaded. Appendix A gives the shared selection procedure and scale-specific training configurations.

DataComp-small results. MGD achieves an average score of 22% on DataComp-small, compared with the published baseline scores of 18.2% for EcoDatum and 13.2% without filtering (DataComp Team, 2026). Inspecting scores over the course of the optimization in Figure 4, we find that only a few steps are necessary to outperform these baselines.



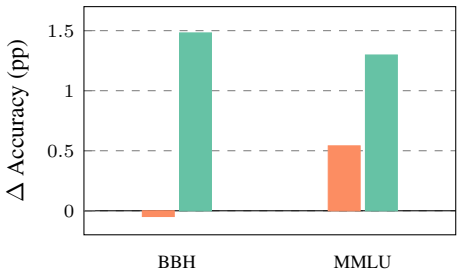
Method	Score (%)	Δ (pp)
Baseline: No filtering	13.2	–
Best baseline from Gadre et al. (2024)	17.3	+4.1
EcoDatum (2025)	18.2	+5.0
MGD (ours)	**22**	+8.8

Figure 4: DataComp-small: the left panel shows scores over MGD steps; the right table compares MGD's final result with published baseline scores (DataComp Team, 2026). The unfiltered reference line uses the leaderboard baseline; improvements in the table are computed from the reported scores.

DataComp-medium results. We use the same MGD selection procedure at medium scale, with the medium candidate pool and CLIP training configuration. The medium benchmark uses a pool of 128 million candidate pairs and trains a ViT-B/32 model for 128 million sample exposures (Gadre et al., 2024). Table 1 compares MGD with earlier methods on the filtering-track leaderboard. MGD achieves an average score of 40.2% across the DataComp evaluation tasks, improving over the strongest earlier entry, negCLIPLoss & NormSim + DFN + HYPE (Wang et al., 2024), by 1.4 percentage points.

Method	Average (%)
No filtering	25.8
Image-based ∩ CLIP score (L/14 30%)	32.8
Data Filtering Networks (Fang et al., 2024)	37.3
HYPE sampler (Kim et al., 2024)	37.3
HYPE sampler + DFN	37.9
negCLIPLoss & NormSim (Wang et al., 2024) + DFN	38.6
negCLIPLoss & NormSim + DFN + HYPE	38.8
MGD (ours)	**40.2**

Table 1: DataComp-medium results from the official leaderboard (DataComp Team, 2026), retrieved September 24, 2026. We report the leading earlier filtering-track entries and the original unfiltered and best-average-score DataComp baselines. The average is over 38 evaluation tasks. These are the published leaderboard scores under each method's selection protocol.



	BBH		MMLU	
	Acc.	Δ (pp)	Acc.	Δ (pp)
All Data	35.21%	–	41.22%	–
LESS	35.16%	−0.05	41.76%	+0.54
MGD	**36.69%**	**+1.48**	**42.52%**	**+1.30**

Figure 5: MGD dataset selection outperforms baselines. MGD achieves 36.7% on BBH and 42.5% on MMLU, compared with 35.2% and 41.8% for LESS. BBH: Suzgun et al. (2022); MMLU: Hendrycks et al. (2020b). The Δ column reports improvement over not filtering in percentage points. Accuracies and differences are rounded independently from the unrounded results.

4 Instruction-tuning data: Gemma-2B

For instruction tuning, we apply the same MGD-based method (Algorithm 1) to select instruction fine-tuning (IFT) data. As with multimodal data, training on the "right" post-training data can greatly impact deployment-time model performance (Liu et al., 2024; Dubey et al., 2024; Taori et al., 2023); MGD improves over baselines at choosing IFT data for MMLU (Hendrycks et al., 2020b), a general knowledge task, and BBH (Suzgun et al., 2022), a reasoning/chain-of-thought task.

Setting. We adopt the setting of LESS (Xia et al., 2024). Here, the goal is to select a training data subset from four combined IFT datasets (Flan V2 (Longpre et al., 2023), CoT (Wei et al., 2022), DOLLY (Conover et al., 2023), and Open Assistant 1 (Köpf et al., 2024)) to maximize accuracy on a given target task. We consider two target tasks from LESS: MMLU (which comprises multiple choice questions spanning a variety of disciplines) and BBH (a 23 task subset of BIG-Bench (Srivastava et al., 2022)). In this setup, the data selector can access samples from each task built from the in-context learning prompts. Following Xia et al. (2024), we fine-tune a 128-width LoRA (Hu et al., 2021) (in our work, on Gemma-2B (Team et al., 2024)). See Appendix B for full details on the tasks and learning algorithm.

Method. We split up the available task samples into two sets—a "target" set and a "validation" set—then select data with MGD (via Algorithm 1) by minimizing causal language modeling loss on the "target" set of samples. Appendix B gives the update and target-minibatch settings.

Training and selection protocol. We compare with two baselines: training on the full dataset (i.e., training on the entirety of all the data for a single epoch), and LESS (we use the data selected according to LESS-T (Xia et al., 2024), following the recommendation of 4 epochs). The target sets contain 23 BBH examples and 57 MMLU examples; the validation sets contain 46 and 228 examples, respectively. We first select three iterates per run by validation loss, then choose among these by accuracy on the combined target and validation examples. As in DataComp, we select data through a smoothed training routine and then transfer the selected dataset back to the original Gemma-2B model for final instruction tuning. The optimizer modifications are described in Appendix B. We additionally hyperparameter select for the best learning rate across each baseline by minimizing validation set loss (Appendix B).

Results. MGD improves most on BBH, a reasoning task, compared to the best baseline (+1.5 percentage points). On MMLU, a knowledge-based task, we outperform baselines by slightly less compared to the best baseline (+0.8 percentage points).

Beyond raw accuracy, we inspect losses across each step of the optimization process. Overall, our method improves validation loss over MGD steps (Figure 6), but also exhibits signs of overfitting. Here we optimize 270,679 data counts using only 23 target examples for BBH and 57 for MMLU, making the choice of iterate important.

Other selection methods. In the Llama-2-7B setting of Xia et al. (2024), BM25, DSIR, and representation-based data selection (RDS) all score below full-data training on both BBH and MMLU, and LESS outperforms them. Table 4 in Appendix B reports these published results.

5 Related work

Learning to select and weight data. Ren et al. (2018) meta-learn minibatch example weights, and Meta-Weight-Net (Shu et al., 2019) learns a loss-to-weight mapping. Bilevel coresets (Borsos et al., 2020) and GLISTER (Killamsetty et al., 2021b) optimize subset selection; CRAIG (Mirzasoleiman et al., 2020) and GRAD-MATCH (Killamsetty et al., 2021a) select subsets by approximating training or validation gradients. DoReMi (Xie et al., 2023a) instead optimizes weights over data domains. More recently, PDS (Gu et al., 2025) formulates language-model data selection as optimal control, and DataRater (Calian et al., 2025) meta-learns a data-scoring network. MGD optimizes per-example integer counts through smoothed training trajectories.

Data filtering. DSIR (Xie et al., 2023b) matches a target distribution through importance resampling in a reduced feature space. LESS (Xia et al., 2024) uses gradient information for instruction-tuning data selection. For multimodal pre-training, Data Filtering Networks (Fang et al., 2024) learn models specifically for filtering, HYPE (Kim et al., 2024) filters using hyperbolic entailment and semantic specificity, and EcoDatum (Xu et al., 2025) combines multiple filtering signals. Wang et al. (2024) propose CLIPLoss-based filtering and NormSim, which measures similarity to target data.

Data attribution and distillation. Training data attribution estimates how predictions change as training data changes (Ilyas et al., 2022; Koh & Liang, 2017; Grosse et al., 2023; Park et al., 2023). Data Shapley (Ghorbani & Zou, 2019) averages marginal contributions over subsets; In-Run Data Shapley (Wang et al., 2025a) accumulates per-step Shapley values along a training run. TracIn (Pruthi et al., 2020) uses gradient alignment along training, while trajectory-specific influence (Wang et al., 2025b) accounts for data ordering. Dataset distillation (Wang et al., 2018; Cazenavette et al., 2022) also differentiates through training, but optimizes synthetic examples rather than selecting existing ones.

Differentiating through training. A long line of work aims to compute metagradients via autodifferentiation (AD) (Hara et al., 2019; Maclaurin et al., 2015; Franceschi et al., 2017; Micaelli & Storkey, 2021; Zhang et al., 2021; Chandra et al., 2022; Scieur et al., 2022). REPLAY can be seen as a specific gradient-checkpointing strategy (Chaitin et al., 1981; Zweig & Padmanabhan, 2000; Griewank & Walther, 2008) applied to the metagradient setting. Building on recursive checkpointing (Blondel & Roulet, 2024), we combine deterministic replay with checkpointing within training steps and specialized backward-over-backward operations to make metagradients practical at scale. Appendix E discusses explicit and implicit differentiation in more detail.

6 Discussion

Limitations. Computing metagradients with REPLAY is still non-trivially more expensive than simply training a model once. The main reason is that metagradients require making a *backwards pass over a backwards pass*. In our implementation, this operation requires approximately 2-3 times the operations of a backwards pass; furthermore, our current implementation requires `float32`/`tensorfloat32` operations. Supporting custom kernels requires writing a backward-over-backward pass, which can be time-consuming. Beyond computational issues, successfully applying metagradients requires smooth model training. Each dataset update requires a new training run, so selection cost grows with the number of updates (Appendix A.3). This cost is additional to DataComp's fixed final-training budget. The metagradient is exact for the continuous insertion surrogate; the resulting integer count update need not decrease the final loss. Small target sets also create a risk of overfitting, as the instruction-tuning loss curves illustrate. IFT scores are reported as point estimates, without an estimate of variation across training runs.

7 Conclusion

In this work we use metagradient descent to optimize training datasets. We apply it to large-scale models (up to 2B parameters) to select data for CLIP pretraining and instruction fine-tuning. After transfer back to the original models, the resulting datasets improve DataComp performance and instruction tuning relative to the evaluated baselines.

AI USE STATEMENT

Generative AI assistance was used to adapt the manuscript to the conference format, draft targeted edits to the manuscript, and clean up presentation of experimental results and methods (e.g., by generating diagrams). Generative AI tools were used for running experiments (e.g., via Codex), but not for hypothesis generation, experimental design, ideation, or any other conceptual aspect of the research. The authors take full responsibility for the contents of this work.

REFERENCES

Amro Abbas, Kushal Tirumala, Dániel Simig, Surya Ganguli, and Ari S Morcos. Semdedup: Data-efficient learning at web-scale through semantic deduplication. *arXiv preprint arXiv:2303.09540*, 2023.

Juhan Bae, Nathan Ng, Alston Lo, Marzyeh Ghassemi, and Roger Grosse. If influence functions are the answer, then what is the question? In *ArXiv preprint arXiv:2209.05364*, 2022.

Peter Bandi, Oscar Geessink, Quirine Manson, Marcory Van Dijk, Maschenka Balkenhol, Meyke Hermsen, Babak Ehteshami Bejnordi, Byungjae Lee, Kyunghyun Paeng, Aoxiao Zhong, et al. From detection of individual metastases to classification of lymph node status at the patient level: the CAMELYON17 challenge. *IEEE Transactions on Medical Imaging*, 2018.

Andrei Barbu, David Mayo, Julian Alverio, William Luo, Christopher Wang, Dan Gutfreund, Josh Tenenbaum, and Boris Katz. Objectnet: A large-scale bias-controlled dataset for pushing the limits of object recognition models. In *Neural Information Processing Systems (NeurIPS)*, 2019.

Sara Beery, Arushi Agarwal, Elijah Cole, and Vighnesh Birodkar. The iWildCam 2021 competition dataset. In *arXiv preprint arXiv:2105.03494*, 2021.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. In *IEEE Transactions on Neural Networks*, 1994.

Quentin Bertrand, Quentin Klopfenstein, Mathieu Blondel, Samuel Vaiter, Alexandre Gramfort, and Joseph Salmon. Implicit differentiation of lasso-type models for hyperparameter optimization. In *International Conference on Machine Learning*, pp. 810–821. PMLR, 2020.

Quentin Bertrand, Quentin Klopfenstein, Mathurin Massias, Mathieu Blondel, Samuel Vaiter, Alexandre Gramfort, and Joseph Salmon. Implicit differentiation for fast hyperparameter selection in non-smooth convex learning. *Journal of Machine Learning Research*, 23(149):1–43, 2022.

Yonatan Bitton, Nitzan Bitton Guetta, Ron Yosef, Yuval Elovici, Mohit Bansal, Gabriel Stanovsky, and Roy Schwartz. WinoGAViL: Gamified association benchmark to challenge vision-and-language models. In *Advances in Neural Information Processing Systems*, 2022.

Mathieu Blondel and Vincent Roulet. The Elements of Differentiable Programming. *arXiv preprint arXiv:2403.14606*, 2024.

Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *Advances in neural information processing systems*, 35:5230–5242, 2022.

Zalán Borsos, Mojmir Mutny, and Andreas Krause. Coresets via bilevel optimization for continual learning and streaming. In *Advances in Neural Information Processing Systems*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/aa2a77371374094fe9e0bc1de3f94ed9-Abstract.html.

Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101–mining discriminative components with random forests. In *European conference on computer vision*, 2014.

Preston Briggs, Keith D Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pp. 311–321, 1992.

Dan A. Calian, Gregory Farquhar, Iurii Kemaev, Luisa M. Zintgraf, Matteo Hessel, Jeremy Shar, Junhyuk Oh, András György, Tom Schaul, Jeffrey Dean, Hado van Hasselt, and David Silver. DataRater: Meta-learned dataset curation. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2505.17895.

George Cazenavette, Tongzhou Wang, Antonio Torralba, Alexei A. Efros, and Jun-Yan Zhu. Dataset distillation by matching training trajectories. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022. URL https://arxiv.org/abs/2203.11932.

Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

Kartik Chandra, Audrey Xie, Jonathan Ragan-Kelley, and Erik Meijer. Gradient descent: The ultimate optimizer. *Advances in Neural Information Processing Systems*, 35:8214–8225, 2022.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL http://arxiv.org/abs/1604.06174.

Xiangning Chen and Cho-Jui Hsieh. Stabilizing differentiable architecture search via perturbation-based regularization. In *International conference on machine learning*, pp. 1554–1565. PMLR, 2020.

Gong Cheng, Junwei Han, and Xiaoqiang Lu. Remote sensing image scene classification: Benchmark and state of the art. In *Proceedings of the IEEE*, 2017.

Gordon Christie, Neil Fendley, James Wilson, and Ryan Mukherjee. Functional map of the world. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6 2018.

Mircea Cimpoi, Subhransu Maji, Iasonas Kokkinos, Sammy Mohamed, and Andrea Vedaldi. Describing textures in the wild. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011.

Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. Free dolly: Introducing the world's first truly open instruction-tuned llm, 2023. URL https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

DataComp Team. DataComp leaderboard. https://www.datacomp.ai/dcclip/leaderboard.html, 2026. Small- and medium-scale filtering tracks. Accessed September 24, 2026.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition (CVPR)*, 2009.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Logan Engstrom, Axel Feldmann, and Aleksander Madry. DsDm: Model-aware dataset selection with datamodels. 2024.

M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. In *International Journal of Computer Vision*, 2010.

Alex Fang, Gabriel Ilharco, Mitchell Wortsman, Yuhao Wan, Vaishaal Shankar, Achal Dave, and Ludwig Schmidt. Data determines distributional robustness in contrastive language image pre-training (clip). In *ICML*, 2022.

Alex Fang, Albin Madappally Jose, Amit Jain, Ludwig Schmidt, Alexander Toshev, and Vaishaal Shankar. Data filtering networks. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2309.17425.

Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *2004 conference on computer vision and pattern recognition workshop*, pp. 178–178. IEEE, 2004.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pp. 1126–1135. PMLR, 2017.

Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In *International Conference on Machine Learning (ICML)*, 2017.

Samir Yitzhak Gadre, Gabriel Ilharco, Alex Fang, Jonathan Hayase, Georgios Smyrnis, Thao Nguyen, Ryan Marten, Mitchell Wortsman, Dhruba Ghosh, Jieyu Zhang, et al. DataComp: In search of the next generation of multimodal datasets. In *Advances in Neural Information Processing Systems*, 2024.

Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? The KITTI vision benchmark suite. In *2012 IEEE conference on computer vision and pattern recognition*, 2012.

Amirata Ghorbani and James Zou. Data shapley: Equitable valuation of data for machine learning. In *International Conference on Machine Learning (ICML)*, 2019.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Roger Grosse, Juhan Bae, Cem Anil, Nelson Elhage, Alex Tamkin, Amirhossein Tajdini, Benoit Steiner, Dustin Li, Esin Durmus, Ethan Perez, et al. Studying large language model generalization with influence functions. *arXiv preprint arXiv:2308.03296*, 2023.

Yuxian Gu, Li Dong, Hongning Wang, Yaru Hao, Qingxiu Dong, Furu Wei, and Minlie Huang. Data selection via optimal control for language models. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.07064.

Satoshi Hara, Atsushi Nitanda, and Takanori Maehara. Data cleansing for models trained with sgd. *Advances in Neural Information Processing Systems*, 32, 2019.

Patrick Helber, Benjamin Bischke, Andreas Dengel, and Damian Borth. EuroSAT: A novel dataset and deep learning benchmark for land use and land cover classification. In *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2019.

Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, and Dawn Song. Natural adversarial examples. *arXiv preprint arXiv:1907.07174*, 2019.

Dan Hendrycks, Steven Basart, Norman Mu, Saurav Kadavath, Frank Wang, Evan Dorundo, Rahul Desai, Tyler Zhu, Samyak Parajuli, Mike Guo, Dawn Song, Jacob Steinhardt, and Justin Gilmer. The many faces of robustness: A critical analysis of out-of-distribution generalization, 2020a.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020b.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Andrew Ilyas, Sung Min Park, Logan Engstrom, Guillaume Leclerc, and Aleksander Madry. Datamodels: Predicting predictions from training data. In *International Conference on Machine Learning (ICML)*, 2022.

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

Krishnateja Killamsetty, Durga S, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. GRADMATCH: Gradient matching based data subset selection for efficient deep model training. In *International Conference on Machine Learning*, 2021a. URL https://proceedings.mlr.press/v139/killamsetty21a.html.

Krishnateja Killamsetty, Durga Sivasubramanian, Ganesh Ramakrishnan, and Rishabh Iyer. Glister: Generalization based data subset selection for efficient and robust learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 8110–8118, 2021b.

Wonjae Kim, Sanghyuk Chun, Taekyung Kim, Dongyoon Han, and Sangdoo Yun. HYPE: Hyperbolic entailment filtering for underspecified images and texts. In *European Conference on Computer Vision*, 2024. URL https://arxiv.org/abs/2404.17507.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, 2017.

Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Sara Beery, et al. WILDS: A benchmark of in-the-wild distribution shifts. *arXiv preprint arXiv:2012.07421*, 2020.

MJ Zico Kolter, David Duvenaud, and Matt Johnson. Deep implicit layers-neural odes, deep equilibirum models, and beyond, 2020. *NeurIPS Tutorial*, 2020.

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi Rui Tam, Keith Stevens, Abdullah Barhoum, Duc Nguyen, Oliver Stanley, Richárd Nagyfi, et al. Openassistant conversations-democratizing large language model alignment. *Advances in Neural Information Processing Systems*, 36, 2024.

Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, 2013.

Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Madry. ffcv. https://github.com/libffcv/ffcv/, 2022.

Yann LeCun. The mnist database of handwritten digits. In *Technical report*, 1998.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision (ECCV)*, 2014.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. In *arXiv preprint arXiv:2412.19437*, 2024.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, pp. 22631–22648. PMLR, 2023.

Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International conference on artificial intelligence and statistics*, pp. 1540–1552. PMLR, 2020.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning (ICML)*, 2015.

Subhransu Maji, Esa Rahtu, Juho Kannala, Matthew Blaschko, and Andrea Vedaldi. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151*, 2013.

Paul Micaelli and Amos J Storkey. Gradient-based hyperparameter optimization over long horizons. *Advances in Neural Information Processing Systems*, 34:10798–10809, 2021.

Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In *International Conference on Machine Learning*, pp. 6950–6960. PMLR, 2020.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Baolin Wu, Andrew Y Ng, et al. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, 2011.

Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, 2008.

James M Ortega and Werner C Rheinboldt. *Iterative solution of nonlinear equations in several variables*. SIAM, 2000.

Sung Min Park, Kristian Georgiev, Andrew Ilyas, Guillaume Leclerc, and Aleksander Madry. TRAK: Attributing model behavior at scale. In *Arxiv preprint arXiv:2303.14186*, 2023.

Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*, pp. 3498–3505. IEEE, 2012.

Barak A Pearlmutter. An investigation of the gradient descent process in neural networks. In *PhD thesis, Carnegie Mellon University*, 1996.

Garima Pruthi, Frederick Liu, Mukund Sundararajan, and Satyen Kale. Estimating training data influence by tracing gradient descent. In *Neural Information Processing Systems (NeurIPS)*, 2020.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *arXiv preprint arXiv:2103.00020*, 2021.

Vikram V Ramaswamy, Sing Yu Lin, Dora Zhao, Aaron Adcock, Laurens van der Maaten, Deepti Ghadiyaram, and Olga Russakovsky. GeoDE: a geographically diverse evaluation dataset for object recognition. In *Advances in Neural Information Processing Systems*, 2024.

Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do imagenet classifiers generalize to imagenet? In *International Conference on Machine Learning (ICML)*, 2019.

Mengye Ren, Wenyuan Zeng, Bin Yang, and Raquel Urtasun. Learning to reweight examples for robust deep learning. In *International Conference on Machine Learning*, 2018. URL https://proceedings.mlr.press/v80/ren18a.html.

William A Gaviria Rojas, Sudnya Diamos, Keertan Ranjan Kini, David Kanter, Vijay Janapa Reddi, and Cody Coleman. The dollar street dataset: Images representing the geographic and socioeconomic diversity of the world. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

Damien Scieur, Gauthier Gidel, Quentin Bertrand, and Fabian Pedregosa. The curse of unrolling: Rate of differentiating through optimization. *Advances in Neural Information Processing Systems*, 35:17133–17145, 2022.

Jun Shu, Qi Xie, Lixuan Yi, Qian Zhao, Sanping Zhou, Zongben Xu, and Deyu Meng. Meta-Weight-Net: Learning an explicit mapping for sample weighting. In *Advances in Neural Information Processing Systems*, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/e58cc5ca94270acaceed13bc82dfedf7-Abstract.html.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *The 2011 international joint conference on neural networks*, 2011.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

Bart Thomee, David A. Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. Yfcc100m: The new data in multimedia research. *Communications of the ACM*, 2016.

Bastiaan S Veeling, Jasper Linmans, Jim Winkens, Taco Cohen, and Max Welling. Rotation equivariant cnns for digital pathology. In *Medical Image Computing and Computer Assisted Intervention–MICCAI 2018: 21st International Conference, Granada, Spain, September 16-20, 2018, Proceedings, Part II 11*, 2018.

Haohan Wang, Songwei Ge, Eric P Xing, and Zachary C Lipton. Learning robust global representations by penalizing local predictive power. *Neural Information Processing Systems (NeurIPS)*, 2019.

Jiachen T. Wang, Prateek Mittal, Dawn Song, and Ruoxi Jia. Data Shapley in one training run. In *International Conference on Learning Representations*, 2025a. URL https://arxiv.org/abs/2406.11011.

Jiachen T. Wang, Dawn Song, James Zou, Prateek Mittal, and Ruoxi Jia. Capturing the temporal dependence of training data influence. In *International Conference on Learning Representations*, 2025b. URL https://arxiv.org/abs/2412.09538.

Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A. Efros. Dataset distillation. 2018.

Yiping Wang, Yifang Chen, Wendan Yan, Alex Fang, Wenjing Zhou, Kevin G Jamieson, and Simon S Du. Cliploss and norm-based data selection methods for multimodal contrastive learning. *Advances in Neural Information Processing Systems*, 37:15028–15069, 2024.

Team Webdataset. webdataset, 2024. URL https://www.github.com/webdataset/webdataset.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024.

Jianxiong Xiao, James Hays, Krista A Ehinger, Aude Oliva, and Antonio Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *Computer Vision and Pattern Recognition (CVPR)*, 2010.

Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy Liang, Quoc V Le, Tengyu Ma, and Adams Wei Yu. Doremi: Optimizing data mixtures speeds up language model pretraining. *arXiv preprint arXiv:2305.10429*, 2023a.

Sang Michael Xie, Shibani Santurkar, Tengyu Ma, and Percy Liang. Data selection for language models via importance resampling. *arXiv preprint arXiv:2302.03169*, 2023b.

Jinda Xu, Yuhao Song, Daming Wang, Weiwei Zhao, Minghua Chen, Kangliang Chen, and Qinya Li. Quality over quantity: Boosting data efficiency through ensembled multimodal data curation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 21761–21769, 2025. doi:10.1609/aaai.v39i20.35481. URL https://ojs.aaai.org/index.php/AAAI/article/view/35481.

Peter Young, Alice Lai, Micah Hodosh, and Julia Hockenmaier. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. In *Transactions of the Association for Computational Linguistics*, 2014.

Xiaohua Zhai, Joan Puigcerver, Alexander Kolesnikov, Pierre Ruyssen, Carlos Riquelme, Mario Lucic, Josip Djolonga, Andre Susano Pinto, Maxim Neumann, Alexey Dosovitskiy, et al. A large-scale study of representation learning with the visual task adaptation benchmark. In *arXiv preprint arXiv:1910.04867*, 2019.

Miao Zhang, Steven W Su, Shirui Pan, Xiaojun Chang, Ehsan M Abbasnejad, and Reza Haffari. idarts: Differentiable architecture search with stochastic implicit gradients. In *International Conference on Machine Learning*, pp. 12557–12566. PMLR, 2021.

Geoffrey Zweig and Mukund Padmanabhan. Exact alpha-beta computation in logarithmic space with application to MAP word graph construction. In *Sixth International Conference on Spoken Language Processing, ICSLP 2000 / INTERSPEECH 2000, Beijing, China, October 16-20, 2000*, pp. 855–858. ISCA, 2000. doi:10.21437/ICSLP.2000-404. URL https://doi.org/10.21437/ICSLP.2000-404.

A METAGRADIENTS FOR DATACOMP

This appendix describes the shared data-selection setup for DataComp-small and DataComp-medium. Both use Algorithm 1; we change the candidate pool and benchmark training configuration with scale.

A.1 SURROGATE-ALGORITHM OBJECTIVE

The body (§2.4) describes the surrogate algorithm $\mathcal{A}'_{\mathbf{c}}$ at a high level. The full per-iteration objective is

$$\ell'_t(\theta) := \begin{cases} \sum_{x \in t^{\text{th}} \text{ batch}} \ell(x;\theta) & \text{if } t \neq k \\ \sum_{x \in t^{\text{th}} \text{ batch}} \ell(x;\theta) + \sum_{i=1}^{n} z_i \, \ell(x_i;\theta) & \text{if } t = k \end{cases}$$

where $\ell(x;\theta)$ is the training loss on example x, $\mathbf{c}$ is the fixed count vector, and k is the iteration at which we inject the metaparameter perturbation. At $\mathbf{z} = \mathbf{0}$ the surrogate recovers $\widetilde{\mathcal{A}}$ exactly.

A.2 DATASET SELECTION USING MGD

Adapting the CLIP loss function to our surrogate learning algorithm. Here we give the contrastive-loss surrogate used for CLIP. Unlike an example-wise loss, the CLIP loss includes interactions between samples, so the insertion surrogate must also include these cross terms. As a refresher, we first state the "vanilla" CLIP loss function, ℓ, as it is defined in Radford et al. (2021). Letting b be the batch size and d be the embedding dimension, and $\mathbf{x}$ be the training batch at timestep k. Recall that the CLIP model internally has two "submodules": an image embedder, and a text embedder. We then use these to obtain image embeddings $E_I \in \mathbb{R}^{b \times d}$ and text embeddings $E_T \in \mathbb{R}^{b \times d}$ from $\mathbf{x}$. We then compute the image-wise scores, or logits, for this batch as $S = E_I E_T^\top$ [1]. Then, we can define the CLIP loss (as a function of the logits) as

$$L(S) = \frac{1}{2}(L_I(S) + L_T(S)),$$

where L_I and L_T are row-wise and column-wise cross-entropy losses, respectively:

$$L_I(S) = -\sum_{i=1}^{b} \log\left(\frac{\exp(S_{i,i})}{\sum_{j=1}^{b}\exp(S_{i,j})}\right), \quad L_T(S) = -\sum_{i=1}^{b} \log\left(\frac{\exp(S_{i,i})}{\sum_{j=1}^{b}\exp(S_{j,i})}\right).$$

We now wish to relax L into a new function L' that supports an additional input $\mathbf{z} \in \mathbb{R}^n$, where $\frac{\partial L'}{\partial \mathbf{z}}$ resembles the metagradients with respect to dataweights. In order to do this, we imagine passing the *entire dataset* D into our embedder to obtain E'_I and E'_T, and take our new logits $S' = E'_I E'^\top_T \in \mathbb{R}^{n \times n}$.

The loss formulas below index occurrences: repeated candidate indices retain separate embedding rows and separate rows and columns in S'. Here n counts entries in this occurrence-indexed list. The batch indicator and the comparisons $i = j$ and $i \neq j$ refer to these positions, even when two positions contain the same candidate. The data counts in the main text continue to refer to original candidate indices.

There are some additional key conditions our relaxation L' should satisfy. Particularly: when $\mathbf{z} = \mathbf{0}_n$, we should recover the normal CLIP loss L including batches with repeated candidate indices. When $\mathbf{z}$ has a single unit entry for an occurrence absent from that batch, L' should equal the loss with that occurrence appended. In addition, L' should always have meaningful partials with respect to $\mathbf{z}$, even when some values in $\mathbf{z}$ are 0.

Letting $\mathbf{1}_{i=j}$ and $\mathbf{1}_{i\neq j}$ be indicator variables and letting $\mathbf{1}_k \in \{0,1\}^n$ be the indicator vector for the k-th batch, we find that the definition

$$L'(S', \mathbf{z}) = \tfrac{1}{2}\big(L'_I(S', \mathbf{z}) + L'_T(S', \mathbf{z})\big),$$

[1] The CLIP model scales these logits by a temperature parameter τ before applying the softmax. While we omit τ in our definitions, it can be easily incorporated. All our experiments use temperature scaling.

where

$$L'_I(S', \mathbf{z}) = -\sum_{i=1}^{n}(z_i + (\mathbf{1}_k)_i)\log\left(\frac{\exp(S'_{i,i})}{\sum_{j=1}^{n}\exp(S'_{i,j})\,(\mathbf{1}_{i=j} + \mathbf{1}_{i\neq j}(z_j + (\mathbf{1}_k)_j))}\right)$$

and

$$L'_T(S', \mathbf{z}) = -\sum_{i=1}^{n}(z_i + (\mathbf{1}_k)_i)\log\left(\frac{\exp(S'_{i,i})}{\sum_{j=1}^{n}\exp(S'_{j,i})\,(\mathbf{1}_{i=j} + \mathbf{1}_{i\neq j}(z_j + (\mathbf{1}_k)_j))}\right)$$

satisfy these conditions. The displayed loss sums the per-example cross-entropies.

Finally, we define the loss for the entire batch ℓ' as a function of $\mathbf{z}$ and model parameters θ which outputs the loss calculated according to L' above. To summarize, letting $\mathbf{x}^{(t)}$ denote the t-th training batch, the loss function ℓ_t at step t of our surrogate learning algorithm $\mathcal{A}'$ for CLIP training is:

$$\ell'_t(\theta) := \begin{cases} \ell(\mathbf{x}^{(t)}; \theta) & \text{if } t \neq k \\ \ell'(\mathbf{z}; \theta) & \text{if } t = k. \end{cases}$$

We find that this empirically works well for obtaining meaningful metagradients with respect to dataweights in the CLIP setting.

A.3 Scaling MGD for CLIP and DataComp

The cost of computing metagradients depends on the differentiation horizon and checkpoint schedule (Section 2.2). Here, we outline challenges we faced in scaling MGD in this setting, and how they were resolved. Specifically, we will explain how we efficiently calculated metagradients for CLIP models and efficiently tracked/shuffled our dataset selection from step-to-step despite its large storage footprint.

Computing metagradients. Due to the large batch size used in the CLIP contrastive loss, we implement manual gradient checkpointing to make the operations computationally feasible on our hardware. The most memory-intensive operations are model forward passes (and their gradients): obtaining the image and label embeddings given raw pixel data and tokens. So, we manually make gradient checkpoints before this operation, allowing us to run the embedder in minibatches to avoid memory issues. This setup also naturally lends itself to parallelization across multiple GPUs, which we make use of to further speed up our computations.

Training and differentiation cost. For M dataset updates, each with T training steps and a differentiation horizon of H, the training-trajectory computation scales as

$$\mathcal{O}(M\,[T + H\log_b H + \gamma H])$$

in ordinary training-step equivalents, where b is the checkpoint branching factor and γ is the relative cost of a local reverse pass. The three terms account for training, replaying states, and differentiating the suffix. Computing gradients for candidate examples, evaluating target and validation data, data I/O, and final model training incur additional costs.

Loading, writing, and storing data. Due to the data-intensive nature of training large models like CLIP and our need to frequently produce new datasets at each optimization step, we found that using the webdataset Webdataset (2024) format given by DataComp was restrictively slow. To circumvent this, we rewrote all data following the format of FFCV Leclerc et al. (2022), allowing us to load and write data much faster. Specifically, we divided the entire candidate pool into 8 base shards. Once we trained a model, we choose one of the 8 shards, compute metagradients corresponding to all datapoints in the shard, take a gradient step on them, and rewrite the shard. This roughly corresponds to $p = \frac{1}{8}$ in Algorithm 1, which we empirically worked well for optimizing. In following steps, we always choose one of the 8 *original* shards to calculate metagradients for—this ensures that points removed from the dataset in some optimization step can return if they have a negative metagradient.

Section 2.5 describes the fixed-size update phase used to prevent shard growth.

To maintain randomness during shuffling, we implement an 8-way dataloader which would shuffle all 8 shards individually. Then, to sample a batch of b datapoints, we would sample $b/8$ datapoints

from each shard and concatenate them to fill our batch. This works better than simply sampling our entire batch from a single shard, as (especially in later optimization steps) shards may contain a high number of duplicate datapoints, which causes CLIP's contrastive loss function to misbehave if they appear in the same batch.

To minimize disk space used, old shards can be deleted once they become "stale". Specifically, if shard s is rewritten into shard s', all future optimization steps will never read s again, and s can safely be deleted. Thus, when running MGD for a large number of steps and potentially rewriting each shard multiple times, the total disk space used by our algorithm is constant in the number of steps we take: it stores the 8 most recently written shards on disk at any given time, and any other shards are deleted to save space.

A.4 Details Pertaining to the DataComp Benchmark

Setting. We provide a brief summary of the DataComp competition here, and we refer readers to the original paper Gadre et al. (2024). DataComp is a framework to compare different training dataset selection techniques. Participants submit a selection of candidate indices, upon which a CLIP model is trained from scratch with a fixed learning algorithm, model architecture, and number of training steps. We encode count c_i by including candidate index i exactly c_i times in the submitted selection; DataComp accepts these repeated indices. We focus on DataComp-small, which has a candidate pool of 12.8 million samples. The training budget is fixed at 12.8 million sample exposures.

The smoothed training routine uses the Adam update in Equation 4. After data optimization, we train the original CLIP model with the official DataComp routine on the selected data. Our selection training hyperparameters can be found in Table 2 and are identical to those mandated by DataComp-small, aside from a positive $\varepsilon_{\text{root}}$ added for numerical stability. The values of $\varepsilon_{\text{root}}$ and k (the step at which metagradients are calculated) were chosen to empirically maximize metasmoothness.

Table 2: Selection hyperparameters for DataComp-small.

Hyperparameter	**Value**
DataComp Scale	small
Model	ViT-B/32
Train compute (MACs)	9.5×10^{16}
Pool size	12.8M
# samples seen	12.8M
Batch size	4096
Training batches	3125
k	2800
Learning rate	5×10^{-4}
AdamW β_1	0.9
AdamW β_2	0.98
AdamW $\varepsilon_{\text{root}}$	1×10^{-17}
Warmup	500

Our experiments are also run on an incomplete subset of the entire DataComp candidate pool. DataComp did not store the raw image and text files when assembling their dataset; they only stored a list of URL's to download data from. Due to the nature of the internet, for various reasons, some of these URL's no longer point to the same data (or no longer point to any data at all). Thus, after ignoring these broken links, our candidate pool is only around 80% of the size of the original DataComp candidate pool when it was collected in 2023. All our small-scale results are obtained by running our methods on this subset of the DataComp pool.

Evaluation tasks. In order to ensure that our method is truly improving trained models' performances on the *entire target distribution* and not overfitting to the target set, for each of the 38 evaluation tasks used by DataComp, we attempted to separately create a disjoint target and validation set (DataComp only creates test sets for each task). Thus, metagradients were computed on the target sets and model performance was evaluated on the validation set, before submitting with the of-

ficial DataComp script and evaluating on the test sets. These held-out evaluations test generalization beyond the target set.

For various reasons, creating target splits was not possible for all 38 tasks; we summarize our setup in Table 3.

Table 3: Selected DataComp evaluation tasks. The "Target set" column indicates whether metagradients were taken on the corresponding target set.

Dataset	Task	Test size	Train size	Val size	Main metric	Target set
Caltech-101 Fei-Fei et al. (2004)	Object recognition	6085	2754	306	mean per class	✓
CLEVR Counts Johnson et al. (2017); Zhai et al. (2019)	Counting	15000	65000	5000	accuracy	✓
CLEVR Distance Johnson et al. (2017); Zhai et al. (2019)	Distance prediction	15000	65000	5000	accuracy	✓
Country211 Radford et al. (2021); Thomee et al. (2016)	Geolocation	21100	37980	4220	accuracy	✓
DTD Cimpoi et al. (2014)	Texture classification	1880	3384	376	accuracy	✓
EuroSAT Helber et al. (2019); Zhai et al. (2019)	Satellite imagery recognition	5400	19440	2160	accuracy	✓
FGVC Aircraft Maji et al. (2013)	Aircraft recognition	3333	6001	666	mean per class	✓
Food-101 Bossard et al. (2014)	Food recognition	25250	70750	5000	accuracy	✓
GTSRB Stallkamp et al. (2011)	Traffic sign recognition	12630	35289	3920	accuracy	✓
ImageNet 1k Deng et al. (2009)	Visual recognition	50000	1276167	5000	accuracy	✓
ImageNet Sketch Wang et al. (2019)	Visual recognition	50889	N/A	N/A	accuracy	*
ImageNet V2 Recht et al. (2019)	Visual recognition	10000	N/A	N/A	accuracy	*
ImageNet-A Hendrycks et al. (2019)	Visual recognition	7500	N/A	N/A	accuracy	*
ImageNet-O Hendrycks et al. (2019)	Visual recognition	2000	N/A	N/A	accuracy	*
ImageNet-R Hendrycks et al. (2020a)	Visual recognition	30000	N/A	N/A	accuracy	*
KITTI distance Geiger et al. (2012); Zhai et al. (2019)	Distance prediction	711	N/A	N/A	accuracy	†
MNIST LeCun (1998)	Digit recognition	10000	55000	5000	accuracy	✓
ObjectNet Barbu et al. (2019)	Visual recognition	18574	N/A	N/A	accuracy	*
Oxford Flowers-102 Nilsback & Zisserman (2008)	Flower recognition	6149	1836	204	mean per class	✓
Oxford-IIIT Pet Parkhi et al. (2012); Zhai et al. (2019)	Pet classification	3669	3312	368	mean per class	✓
Pascal VOC 2007 Everingham et al. (2010)	Object recognition	14976	14096	1566	accuracy	✓
PatchCamelyon Veeling et al. (2018); Zhai et al. (2019)	Metastatic tissue cls.	32768	289912	5000	accuracy	✓
Rendered SST2 Zhai et al. (2019)	Sentiment classification	1821	7013	779	accuracy	✓
RESISC45 Cheng et al. (2017); Zhai et al. (2019)	Satellite imagery recognition	6300	22680	2520	accuracy	✓
Stanford Cars Krause et al. (2013)	Vehicle recognition	8041	7329	814	accuracy	✓
STL-10 Coates et al. (2011)	Visual recognition	8000	4500	500	accuracy	✓
SUN-397 Xiao et al. (2010)	Scene recognition	108753	N/A	N/A	accuracy	‡
SVHN Netzer et al. (2011); Zhai et al. (2019)	Digit recognition	26032	68257	5000	accuracy	✓
iWildCam Beery et al. (2021); Koh et al. (2020)	Animal recognition	42791	147084	5000	macro F1 score	✓
Camelyon17 Bandi et al. (2018); Koh et al. (2020)	Metastatic tissue cls.	85054	365900	5000	accuracy	✓
FMoW Christie et al. (2018); Koh et al. (2020)	Satellite imagery recognition	22108	103261	5000	worst-region acc.	✓
Dollar Street Rojas et al. (2022)	Object recognition	3503	13842	1537	worst-income top-5 acc.	✓
GeoDE Ramaswamy et al. (2024)	Object recognition	12438	44488	4943	worst-region acc.	✓
Flickr30k Young et al. (2014)	Image and text retrieval	31014	N/A	N/A	R@1	§
MSCOCO Lin et al. (2014)	Image and text retrieval	5000	N/A	N/A	R@1	§
WinoGAViL Bitton et al. (2022)	Commonsense association	3563	N/A	N/A	Jaccard score	§

[*] No train or val set exists for this dataset, so we were unable to create disjoint target and val sets.

[†] We were unable to use this dataset due to technical difficulties.

[‡] Both the train and val sets were used by DataComp to make their test set, so we were unable to create disjoint target and val sets.

[§] Retrieval tasks were not used for metagradients.

A.5 DATACOMP-MEDIUM

The medium-scale experiment uses the same data-selection setup as DataComp-small: the CLIP insertion surrogate, smoothness interventions, stochastic block updates, fixed-size phase, and validation protocol described above. We apply this procedure to the medium candidate pool and CLIP training configuration, then transfer the selected data to the official medium-scale training routine.

The medium filtering track uses a pool of 128 million candidate pairs and a fixed budget of 128 million sample exposures for ViT-B/32. The official configuration specifies batch size 4096, learning rate 5×10^{-4}, and 500 warmup steps, giving 31,250 training steps.[2]

We preserve the relative insertion position from DataComp-small: $k/T = 2800/3125 = 0.896$. Thus, medium-scale selection inserts the weights at $k = 28{,}000$ and differentiates through the remaining 3,250 training steps.

[2]Official scale configuration: https://github.com/mlfoundations/datacomp/blob/main/scale_configs.py.

B SELECTING IFT DATA

In this section, we describe the details of the IFT setting of Xia et al. (2024), as well as the details of our method.

Setting. The setting contains a fixed data pool: instruction fine-tuning data from a data pool consisting of four combined IFT datasets (cf. Table 6 and Xia et al. (2024) for more information). The goal is to select the data that yields the best possible task performance for a LoRA fine-tuning run. We adapt a LoRA to a Gemma-2B model (the pretraining-only Gemma-2B model) using the LoRA configuration from Xia et al. (2024).

Data splits. See Table 5 for a description of the available data for each task, along with the task setup details. Xia et al. (2024) constructed these extra samples by drawing from the ICL samples given in the tasks originally. Note that we drop TydiQA from the original work of Xia et al. (2024) as there are not enough samples to select with (there is only one from each category, for a total of 7).

Method. We execute Algorithm 1 with k as 150 steps from the end of training and the Bernoulli parameter p controlling the step size as 0.2. At each step, we choose a "minibatch" with a size equal to half the target set and a quarter of the target set for BBH and MMLU, respectively (that is, we only select to optimize performance on a fraction of the target set at a time). We select over iterates and hyperparameters in two stages: first, retain the three iterates with lowest validation loss for each run; then, choose among these by accuracy on the combined target and validation examples. Both stages use only the available selection examples; the benchmark test examples are held out. This rule uses validation loss to shortlist iterates before comparing task accuracy on the small selection set.

We compare with two baselines: training on the full dataset (i.e., training on the entirety of all the data for a single epoch), and LESS (we use the data selected according to "LESS-T" (Xia et al., 2024), following the recommendation of 4 epochs).

	Full data	Random	BM25	DSIR	RDS	LESS
BBH	43.2	38.9 ± 0.5	39.8	36.8 ± 0.1	36.7 ± 1.3	41.5 ± 0.6
MMLU	51.6	46.5 ± 0.5	47.6	46.1 ± 0.3	45.0 ± 1.0	50.2 ± 0.5

Table 4: Published Llama-2-7B IFT results (accuracy, %) from Tables 2–3 of Xia et al. (2024). Selection methods use 5% of the candidate data; full-data training uses 100%. Reported deviations are standard deviations across three seeds. These results use a different backbone from our Gemma-2B comparison in Figure 5.

Smoothness interventions. For model training, we train with Adam ($\beta_1 = 0.95, \beta_2 = 0.975$, decoupled weight decay as 10^{-5}) and a one-cycle linear schedule starting at 10^{-6} of the maximum learning rate, reaching the peak over 25% of training, then ending at 0.1 of the maximum learning rate. We insert a positive $\varepsilon_{\text{root}}$ into the inverse square root term in the Adam update to prevent metagradient (and to a lesser extent update) blowup (see Eq. 4). We use $\varepsilon_{\text{root}} = 10^{-7}$ for the MGD selection routine; the baseline runs use $\varepsilon_{\text{root}} = 10^{-9}$.

Final training. After data selection, we transfer the selected dataset back to the original Gemma-2B model and training routine. We additionally hyperparameter select for the best learning rate across each baseline by minimizing validation set loss; LESS performs best with a smaller learning rate (0.00024 for BBH and 0.00012 for MMLU) than training on the full dataset or with MGD (0.0006 for both). We normalize the loss of each training sample by taking the mean across predicted tokens during training, and do not divide by the batch size (important for scaling the $\varepsilon_{\text{root}}$ term, but otherwise Adam is invariant to the scale).

Selecting smooth model training for MGD. For MGD runs, we jointly select learning rate and $\varepsilon_{\text{root}}$ using the smoothness metric of Appendix D. We find that the choice of $\varepsilon_{\text{root}}$ term is important (just as the choice of ε is important in standard Adam training); choosing a much larger term results in non-smooth training. We also find that metagradients are sensitive to learning rate schedule; choosing a much larger or smaller maximum learning rate results in non-smooth training.

Table 5: Overview of datasets used in IFT dataset selection (from Xia et al. (2024)).

Dataset	**# Shot**	**# Tasks**	n_{target}	n_{val}	n_{test}	**Answer type**	**Task type**
MMLU	5	57	57	228	18,721	Letter options	Knowledge/Recall
BBH	3	23	23	46	920	COT and answer	Reasoning

Table 6: Details of IFT training datasets.

Dataset	# Instance	Sourced from	Prompt Len.	Completion Len.
FLAN V2	100,000	NLP datasets and human-written instructions	355.7	31.2
CoT	100,000	NLP datasets and human-written CoTs	266	53.2
Dolly	15,011	Human-written from scratch	118.1	91.3
Open Assistant 1	55,668	Human-written from scratch	34.8	212.5



Figure 6: MGD dataset selection improves the validation loss over metagradient steps. However, the gap between loss on samples MGD directly optimizes on and the validation samples widens over the number of iterates, and there is overfitting depending on the number of steps taken.

C Calculating metagradients with Replay

This appendix describes how we compute metagradients through iterative training. Storing every intermediate optimizer state makes step-wise reverse-mode AD expensive for long training runs. Recursive checkpointing reduces this storage by recomputing states when needed (Griewank & Walther, 2008; Blondel & Roulet, 2024). We develop Replay, a simple algorithm for scalably calculating metagradients using autodifferentiation (AD). Replay leverages determinism to lessen the computational burden of step-wise AD, lowering its space complexity from *linear* in the number of training iterations to *logarithmic*.

C.1 What is a metagradient?

Training a machine learning model is a two-step process. First, we decide on a *training setup*—we must pick, for example, what neural network architecture to use, how to select a training dataset, and what kind of optimizer to use. Second, we apply the algorithm defined by this training setup to learn a model. Our (overall) goal is to optimize model behavior as a function of the training setup (or, as we call it in this paper, the *meta-parameters*) using gradient-based methods.

To that end, we define the following notation. Let $z \in \mathbb{R}^n$ be an n-dimensional vector of meta-parameters representing the aspects of the training setup we aim to optimize. For example, if we only want to adjust the learning rate and weight decay of SGD then $n = 2$. Let $\mathcal{A}$ be an *algorithm* mapping z to a trained machine learning model—for ease of notation, we assume that all the aspects

of the training setup outside z are fixed and thus part of the algorithm $\mathcal{A}$. So, if we only aimed to take the metagradient with respect to learning rate and weight decay, then we would bake model architecture and training dataset (among other algorithmic choices) into $\mathcal{A}$. Finally, let ϕ be an *output function* mapping a machine learning model θ to a vector $\phi(\theta) \in \mathbb{R}^d$. For example, $\phi(\theta)$ might represent the loss of a classifier with parameters θ on every example; or the toxicity of a language model with parameters θ. (We assume $d = 1$ for simplicity, but we can easily extend to $d > 1$.) We define the *training function* $f := \phi \circ \mathcal{A}$ mapping the training setup of interest z *directly* to the output function ϕ evaluated on the corresponding model. Finally, the *metagradient* is the gradient of the training function with respect to the meta-parameters, $\nabla_z f(z)$. Intuitively, the metagradient defines the "direction of steepest ascent" in meta-parameter space.

Our focus: iterative algorithms. We restrict our focus to iterative algorithms of the form

$$\underbrace{\mathcal{A}(z) := \mathbf{s}_T}_{\text{model state after } T \text{ steps}}, \quad \text{where} \quad \underbrace{\mathbf{s}_{t+1} := h_t(\mathbf{s}_t, z)}_{\text{optimizer step } t} \quad (\text{with } \mathbf{s}_0 \text{ the initial state}) \,. \tag{6}$$

Here, $\mathbf{s}_t$ is the optimizer state at step t, and h_t is the *update* mapping from state t to state $t+1$. The form of (6) captures most large-scale ML training algorithms, including SGD and Adam.

C.2 Background: Calculating metagradients with autodifferentiation

A key primitive we leverage to calculate metagradients is *automatic differentiation* (AD)—a standard tool for taking gradients through computer-defined functions. AD takes gradients by decomposing functions into elementary operations with known derivatives, then combining these derivatives using the chain rule. For the purposes of this paper, we will view AD as a black box that calculates the gradient of a function at a given point using only a small constant factor more time than calculating the function itself (along with the space cost of storing all the forward-pass products).

Approach #1: Direct AD. The most direct approach to calculating metagradients exploits the fact that nearly any learning algorithm is itself a computer-defined function and therefore ultimately a sequence of differentiable operations—*including the training function f that we aim to differentiate*. Operationalizing this observation turns out to be quite challenging because AD stores intermediate products for *each* operation. The amount of data stored thus scales with the number of operations in the function of interest. In the case of our training function f, this number encompasses *all* the operations used to train a machine learning model from scratch. As a result, even in a toy scenario like MNIST training, computing metagradients with naïve AD would store terabytes of data.

Approach #2: Exploiting structure with step-wise AD. A more efficient method, *step-wise AD*, leverages the iterative structure of (6) to propagate gradients backwards through training one step at a time, using the chain rule on the recurrence $\mathbf{s}_t = h_{t-1}(\mathbf{s}_{t-1}, \mathbf{z})$ (Werbos, 1990; Maclaurin et al., 2015; Franceschi et al., 2017). The total compute is comparable to a regular training run; the catch is *space*: step-wise AD must save *every* intermediate optimizer state $\mathbf{s}_0, \ldots, \mathbf{s}_T$, so its memory cost scales as $\mathcal{O}(T)$ which is still infeasible at the scales we target. We give the full derivation in Appendix C.4.

C.3 Replay

Despite greatly improving storage overhead compared to "direct AD", step-wise AD is still too space-intensive at scale. Replay is our algorithm for efficiently and exactly computing metagradients. The algorithm requires $\mathcal{O}(b \log_b(T))$ space, where b is a user-chosen constant. The main idea is to make the space-intensive subroutine of step-wise AD much more efficient. Replay modifies step-wise AD to traverse states in less space by exploiting a simple observation: when training is deterministic, one can *reinstantiate* an optimizer state $\mathbf{s}_t$ by "replaying" training from a fixed point $t' < t$—at the compute cost of $t - t'$ training steps. Replay performs a reverse-order traversal the optimizer states while balancing the compute cost of "replaying" training with the storage cost of saving states. We design an efficient data structure, similar to a segment tree (see Figure 7), that reverse-order traverses the optimizer states with $\mathcal{O}(b \log_b(T))$ space and an additional $\mathcal{O}(T \log_b(T))$ model steps—we detail this algorithm in Appendix C.5. Our algorithm unlocks computing large-scale metagradients by requiring only logarithmic storage and additional compute time.

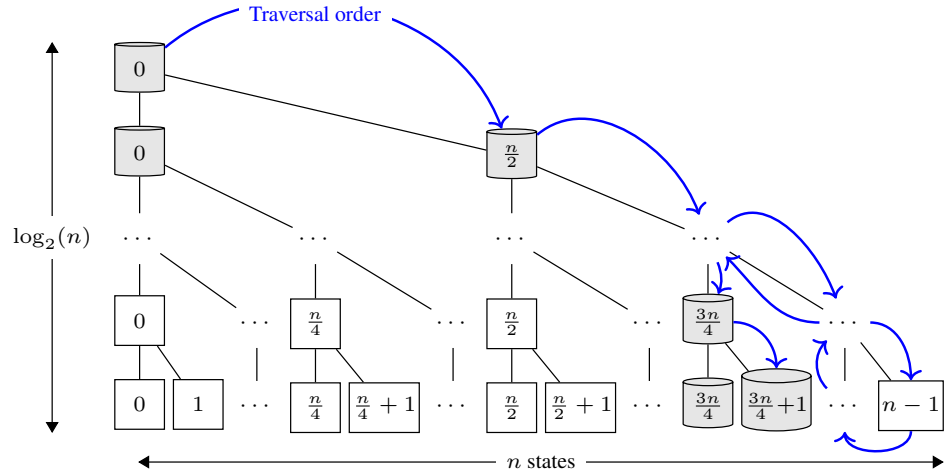


Figure 7: The lazy b-ary tree structure for traversing optimizer states in reverse order, with $b = 2$ ($n = T + 1$ states; each node is one state). Blue arrows trace the traversal path up to $s_{\frac{3n}{4}+1}$; gray cylinders are stored at that point. Total: $\mathcal{O}(\log n)$ states stored and $\mathcal{O}(n \log n)$ optimizer steps (vs. n for plain training).

Remark 1 (Connection to rematerialization.)**.** *Both* REPLAY *and step-wise AD above can be viewed as special cases of a classical approach in AD known as rematerialization (Chaitin et al., 1981; Briggs et al., 1992; Zweig & Padmanabhan, 2000; Griewank & Walther, 2008; Chen et al., 2016). More specifically,* REPLAY *can be seen as a special case of recursive rematerialization (or "recursive halving AD" (Blondel & Roulet, 2024)) for the meta-differentiation setting.*

Remark 2 (Reversible learning)**.** *An alternative approach to calculating metagradients that does not save any state is reversible learning (Maclaurin et al., 2015), for which one can "invert" previous training states from future ones. We focus here on general (non-reversible) learning algorithms for two reasons: first, even simple algorithms such as SGD without momentum are non-reversible; second, reversibility in practice introduces numerical precision issues.*

We now give the chain-rule derivation and checkpoint traversal using the notation above.

C.4 WARMUP: STEP-WISE AD

Chain-rule derivation. We assume the initial state is independent of $\mathbf{z}$. Within each single-step AD call, the incoming cotangent and the stored input state are held fixed; the derivative is taken only with respect to the indicated argument of h_t.

For an iterative algorithm of the form (see (6)) $\mathcal{A}(\mathbf{z}) \coloneqq \mathbf{s}_T$, where $\mathbf{s}_{t+1} := h_t(\mathbf{s}_t, \mathbf{z})$, algebraic manipulation (in particular, using the chain rule, the law of the total derivative, and the identity $\mathbf{s}_t = h_{t-1}(\mathbf{s}_{t-1}, \mathbf{z})$) allows us to write the metagradient over an iterative algorithm as

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \frac{\partial \phi(\mathcal{A}(\mathbf{z}))}{\partial \mathbf{z}} = \sum_{t=1}^{T} \underbrace{\overbrace{\frac{\partial \phi(\mathbf{s}_T)}{\partial \mathbf{s}_t}}^{A_t} \cdot \frac{\partial h_{t-1}(\mathbf{s}_{t-1}, \mathbf{z})}{\partial \mathbf{z}}}_{B_t}, \tag{7}$$

where we have introduced the notation A_t and B_t for notational convenience. Step-wise AD computes the metagradient by calculating each term in the sum of (7) one at a time. For each term, the main challenge lies in computing A_t, since given A_t we can straightforwardly compute B_t by differentiating through a single model update,

$$B_t := A_t \cdot \frac{\partial h_{t-1}(\mathbf{s}_{t-1}, \mathbf{z})}{\partial \mathbf{z}} = \frac{\partial (A_t \cdot h_{t-1}(\mathbf{s}_{t-1}, \mathbf{z}))}{\partial \mathbf{z}},$$

which is just a single call to our assumed "AD oracle" on the function $\mathbf{z} \mapsto A_t \cdot h_{t-1}(\mathbf{s}_{t-1}, \mathbf{z})$. Computing the A_t terms is less straightforward as we need to relate $\mathbf{s}_t$ and $\mathbf{s}_T$; to do so, we exploit

the recurrence

$$A_t := \frac{\partial \phi(\mathbf{s}_T)}{\partial \mathbf{s}_t} = \frac{\partial \phi(\mathbf{s}_T)}{\partial \mathbf{s}_{t+1}} \cdot \frac{\partial h_t(\mathbf{s}_t, \mathbf{z})}{\partial \mathbf{s}_t} = \frac{\partial (A_{t+1} \cdot h_t(\mathbf{s}_t, \mathbf{z}))}{\partial \mathbf{s}_t}, \tag{8}$$

making A_t straightforward to compute (again, a single "AD oracle" call) given A_{t+1}. Step-wise AD exploits this fact to successively calculate the gradient with respect to each state, from state T down to state 0.

Algorithm. We fully describe step-wise AD in Algorithm 2. The algorithm requires storing all T optimizer states, but requires constant memory overhead for each AD call (as each AD call is over a single step), making it feasible to compute for small setups.

Algorithm 2: metagradients in $\mathcal{O}(T)$ space.

```
// Store each optimizer state on disk
{s_i}_{i=0}^T ← Train model via A(z)

// Variables; shorthand for ∂f(z)/∂z and ∂f(z)/∂s_T
z̄ ← 0
s̄_T ← ∂φ(s_T)/∂s_T      // One reverse-mode AD call

// Reverse-mode differentiate step-by-step
for s_i ← s_{T−1} to s_0 do
    // One reverse-mode AD call. Left: ∇_{s_i} f. Right:
       contribution to ∇_z f at i.
    s̄_i ← s̄_{i+1} · ∂h_i(s_i,z)/∂s_i,      z̄_i ← s̄_{i+1} · ∂h_i(s_i,z)/∂z

    z̄ ← z̄ + z̄_i      // Accumulate metagradient

Return z̄
```

C.5 REPLAY

We now describe REPLAY, our method for calculating metagradients. For a free parameter $b \in \mathbb{N}$ with $b \geq 2$, REPLAY requires storing $\mathcal{O}(b \log_b(T))$ optimizer states and an additional $\mathcal{O}(\log_b(T))$ factor of computation. The free parameter b controls the trade-off between storage and required compute. We fully describe REPLAY in Algorithm 3. REPLAY modifies Algorithm 2 by retrieving the optimizer states in reverse order using a b-ary tree structure in lieu of a list of all the stored states.

C.5.1 LAZY b-ARY TREE

We now describe the b-ary tree structure that underlies REPLAY; for a visual reference of this tree with $b = 2$, see Figure 7. For ease of analysis we parameterize the total number of states as $n = T+1$ (and therefore take $n-1$ total training steps) when describing this data structure, and assume WLOG that n is an integer power of b. At a high level, traversing this tree recursively replays retraining to recover all the optimizer states in reverse order, while deleting states that are no longer needed. We call this tree "lazy" because it retrains only when required to obtain states that are not yet retrieved.

The tree is a complete b-ary tree with n leaves (and therefore $\log_b(n)$ depth) structured as follows. We start at the root, then recursively define the rest of the tree. Every node in the tree represents a single optimizer state. The root represents state s_0. To recursively define the remaining nodes: each non-leaf node s_i at depth d has b equally spaced (in terms of state number) children starting—from left to right—at state s_i and ending at $s_{i+(b-1)n/b^{d+1}}$. This means that the leaves correspond—from left to right—to the states $s_0, s_1, \ldots, s_{n-1}$.

We reduce the problem of iterating over the states in reverse to the problem of reverse in-order traversing this tree and yielding *just* the leaves—this is exactly the states in reverse order. A reverse in-order traversal for this b-ary tree requires repeatedly: recursively traversing child nodes from

largest to smallest, then visiting the parent node. We design the specifics of this traversal to maximize space and compute efficiency. To access the children of a parent node at traversal time, we replay model training from the smallest child state (which is stored in the parent state) to the largest child state and store all the children. We perform this operation recursively each time we traverse a node. After traversing the node's left side (i.e., after ascending from this node), we delete all its child states.

Reverse in-order traversing this tree requires storing at most $b \log_b(n)$ optimizer states at a time, and in aggregate requires retraining the model $\log_b(n)$ times. The argument for each is straightforward. Storage: the traversal requires storing at most b states for each level that it descends (we store b states whenever we first traverse to a parent node) and we remove b states for each level that the traversal ascends (we remove b states after we are done with the left traversal of a parent). Compute: we replay training to reinstantiate the children of every parent node a single time. The b^d parent nodes at level d each require replaying $\mathcal{O}(n/b^d)$ states to reinstantiate children. Therefore, in a traversal, each level requires $\mathcal{O}(n)$ ($b^d \cdot n/b^d$) optimizer steps. There are $\log_b(n)$ levels with parent nodes, which means a total of $\mathcal{O}(n \log_b(n))$ optimizer steps, or a multiplicative factor of $\mathcal{O}(\log_b(n))$ steps compared to model training.

Algorithm 3: REPLAY. metagradients in $\mathcal{O}(b \log_b(T))$ space.

$\mathcal{T} \leftarrow$ Lazy b-ary tree for $\mathcal{A}(z)$ `// Make lazy b-ary tree of Appendix C.5`

`// Variables; shorthand for` $\frac{\partial f(z)}{\partial z}$ `and` $\frac{\partial f(z)}{\partial s_T}$

$\bar{z} \leftarrow 0$

$\bar{s}_T \leftarrow \frac{\partial \phi(s_T)}{\partial s_T}$ `// One reverse-mode AD call`

`// Reverse-mode differentiate step-by-step; traverse` $\mathcal{T}$ `instead of stored states`

for $s_i \leftarrow s_{T-1}$ ***to*** $s_0 \in$ `reverse_inorder_traversal(`$\mathcal{T}$`)` **do**

  `// One reverse-mode AD call. Left:` $\nabla_{s_i} f$. `Right: contribution to` $\nabla_z f$ `at` i.

  $\bar{s}_i \leftarrow \bar{s}_{i+1} \cdot \frac{\partial h_i(s_i, z)}{\partial s_i}, \quad \bar{z}_i \leftarrow \bar{s}_{i+1} \cdot \frac{\partial h_i(s_i, z)}{\partial z}$

  $\bar{z} \leftarrow \bar{z} + \bar{z}_i$ `// Accumulate metagradient`

Return $\bar{z}$

D CHOOSING METASMOOTH TRAINING ROUTINES

We use the following finite-difference diagnostic to help choose a metasmooth training routine. It measures agreement between successive parameter changes along one perturbation direction. It does not establish a smoothness bound or guarantee that a discrete dataset update will improve the final model.

Definition 1 (Empirical metasmoothness of $\mathcal{A}$)**.** *Let $\mathcal{A}$ be a learning algorithm which maps metaparameters $\mathbf{z} \in \mathbb{R}^n$ to model parameters $\theta \in \mathbb{R}^d$, let $\mathbf{z}$ be a metaparameter vector, and let $\mathbf{v}$ be a given direction. Fix $h > 0$ and define $\Delta_{\mathcal{A}}(\mathbf{z}; \mathbf{v}) := (\mathcal{A}(\mathbf{z} + h\mathbf{v}) - \mathcal{A}(\mathbf{z}))/h$. Hold training randomness fixed across evaluations. Let $\mathbf{d} \in \mathbb{R}^d$ be the per-coordinate variation in θ, i.e., $\mathbf{d} = |\mathcal{A}(\mathbf{z} + 2h\mathbf{v}) - \mathcal{A}(\mathbf{z})|$. For $\|\mathbf{d}\|_1 > 0$, the empirical $(h, \mathbf{v})$-metasmoothness of $\mathcal{A}$ at $\mathbf{z}$ is given by*

$$\widehat{S}_{h,\mathbf{v}}(\mathcal{A}; \mathbf{z}) = \operatorname{sign}(\Delta_{\mathcal{A}}(\mathbf{z}; \mathbf{v}))^\top \cdot \operatorname{diag}\left(\frac{\mathbf{d}}{\|\mathbf{d}\|_1}\right) \cdot \operatorname{sign}(\Delta_{\mathcal{A}}(\mathbf{z} + h\mathbf{v}; \mathbf{v})), \tag{9}$$

which weights each parameter by its range.

Intuitively, (9) is the sign-agreement between consecutive finite-difference metagradients along $\mathbf{v}$, averaged across parameter coordinates and weighted by per-coordinate variation $\mathbf{d}/\|\mathbf{d}\|_1$ (so coordinates that are essentially constant are downweighted). The output lies in $[-1, 1]$.

Having established a method for quantifying metasmoothness, we turn to the practical question: how can we design learning algorithms that are amenable to metagradient optimization? To answer this question, we introduce a straightforward framework: given a learning algorithm, explore a fixed menu of possible modifications to the training setup, and choose the combination that maximizes empirical metasmoothness. In practice, we find that this framework allows us to slightly modify learning algorithms in a way that makes them amenable to first-order methods.

Choice of metaparameter. We choose an arbitrary metaparameter to guide the smoothness search. This suffices in our experiments to find training routines with useful metagradients for dataset optimization. The metagradients for subsequent count updates are computed with respect to the insertion weights in Section 2.4.

For dataset selection, we vary optimizer settings and the insertion step as described in Sections 2.4 and 4. When $\|\mathbf{d}\|_1 = 0$, the diagnostic is undefined and that perturbation provides no evidence for choosing training settings.

The diagnostic applies to the modified training routine used for data optimization. After optimizing the data counts, we train the original model on the selected dataset, as described in Section 2.3.

E ADDITIONAL BACKGROUND ON DIFFERENTIATION

Previous work estimates the metagradient for large-scale models via one of two broad families of methods: implicit differentiation and automatic (explicit) differentiation. Note that in previous literature, synonyms for metagradient include "hyper-gradient" and "outer gradient."

Implicit differentiation. One family of methods differentiates a stationary condition for the trained model, such as $\nabla_\theta \mathcal{L}(z, \theta) = 0$. When the Hessian is invertible, the implicit function theorem gives the derivative of the corresponding local solution branch (Equation 10) (Bertrand et al., 2020; 2022; Kolter et al., 2020; Blondel et al., 2022; Scieur et al., 2022). This local result can apply to non-convex losses, but the resulting derivative need not equal the derivative through a finite training trajectory. Practical estimators may also approximate the inverse-Hessian action (Koh & Liang, 2017; Lorraine et al., 2020; Bae et al., 2022). Differentiating a stationary loss alone does not capture dependence on learning rate or batch ordering when those metaparameters appear only in the optimization trajectory.

Automatic (explicit) differentiation. A long line of work aims to compute metagradients via autodifferentiation (AD) (Hara et al., 2019; Maclaurin et al., 2015; Franceschi et al., 2017; Micaelli & Storkey, 2021; Zhang et al., 2021; Chandra et al., 2022; Scieur et al., 2022). The principal limitations are (a) linear-in-T storage for uncheckpointed direct or step-wise reverse-mode AD and potentially unusable gradients through non-metasmooth training, (b) compute that scales with metaparameter dimension for forward-mode AD (Franceschi et al., 2017; Micaelli & Storkey, 2021; Chandra et al., 2022), (c) approximations that differentiate only over a few training steps (Liu et al., 2018; Chen & Hsieh, 2020; Finn et al., 2017), or (d) reversible-learning constraints (incompatible with non-reversible updates, with additional machinery needed to reconstruct states under finite-precision arithmetic) (Maclaurin et al., 2015). REPLAY can be seen as a specific gradient-checkpointing strategy (Chaitin et al., 1981; Zweig & Padmanabhan, 2000; Griewank & Walther, 2008) applied to the metagradient setting.

E.1 IMPLICIT-FUNCTION-THEOREM FORM FOR METAGRADIENTS

For reference, when $\mathcal{A}$ returns a model state θ that minimizes a strongly convex loss $\mathcal{L}(z, \theta)$, the implicit function theorem gives the closed-form metagradient

$$\nabla_z f(z) = -\overbrace{\left(\frac{d\phi}{d\theta}\bigg|_{\theta=\mathcal{A}(z)}\right)}^{\substack{1\times p \text{ gradient of output} \\ \text{wrt. final params}}} \underbrace{\left(\frac{\partial^2 \mathcal{L}(z,\theta)}{\partial \theta^2}\bigg|_{\theta=\mathcal{A}(z)}\right)^{-1}}_{\substack{p\times p \text{ inverse Hessian of loss} \\ \text{wrt. final params}}} \overbrace{\left(\frac{\partial^2 \mathcal{L}(z,\theta)}{\partial \theta\, \partial z}\bigg|_{\theta=\mathcal{A}(z)}\right)}^{\substack{p\times n \text{ Jacobian of loss gradient} \\ \text{wrt. metaparameters}}}. \tag{10}$$