

# Finding the Muses: Identifying Coresets through Loss Trajectories

Manish Nagaraj<sup>1</sup> Deepak Ravikumar<sup>1</sup> Efstathia Soufleri<sup>1</sup> Kaushik Roy<sup>1</sup>

## Abstract

Deep learning models achieve state-of-the-art performance across domains but face scalability challenges in real-time or resource-constrained scenarios. To address this, we propose *Loss Trajectory Correlation* (LTC), a novel metric for coreset selection that identifies critical training samples driving generalization. LTC quantifies the alignment between training sample loss trajectories and validation set loss trajectories, enabling the construction of compact, representative subsets. Unlike traditional methods with computational and storage overheads that are infeasible to scale to large datasets, LTC achieves superior efficiency as it can be computed as a byproduct of training. Our results on CIFAR-100 and ImageNet-1k show that LTC consistently achieves accuracy on par with or surpassing state-of-the-art coreset selection methods, with any differences remaining under 1%. LTC also effectively transfers across various architectures, including ResNet, VGG, DenseNet, and Swin Transformer, with minimal performance degradation (< 2%). Additionally, LTC offers insights into training dynamics, such as identifying aligned and conflicting sample behaviors, at a fraction of the computational cost of traditional methods. This framework paves the way for scalable coreset selection and efficient dataset optimization.

## 1. Introduction

Deep learning (DL) models rely on vast, diverse datasets to achieve state-of-the-art performance across various domains. However, the computational demands of training on these massive datasets often become prohibitive, especially in real-time or resource-constrained applications. This raises a critical question: “Which subsets of data contribute most to effective generalization?” Addressing this challenge has

led to the concept of *coresets*, compact and representative subsets of the training data that retain the essence of the full dataset.

Coresets enable efficient training by reducing computational and storage overhead while maintaining comparable model performance. Their applicability spans diverse areas such as active learning (Coleman et al., 2020), neural architecture search (Na et al., 2021; Shim et al., 2021), dataset distillation (Cazenavette et al., 2022), and continual learning (Aljundi et al., 2019; Borsos et al., 2020), underscoring their significance in modern deep learning workflows.

Existing methods for coreset selection often face challenges in balancing resource efficiency with effectiveness. Many approaches struggle to scale to large datasets, rely on heuristics that may not align with the true data distribution (Mirza-soleiman et al., 2020; Belouadah et al., 2020; Rebuffi et al., 2017), or depend on resource-intensive computations such as gradient (Killamsetty et al., 2021a; Pruthi et al., 2020) or curvature evaluations (Garg & Roy, 2023). Moreover, methods based on bilevel optimization (Killamsetty et al., 2021b; Xia et al., 2024; Borsos et al., 2020) are often computationally prohibitive due to the intricacy of their underlying formulations. These constraints make them impractical for large-scale or resource-constrained applications, limiting their utility in real-world deep-learning workflows.

To address these limitations, we propose a scalable and resource-efficient method for identifying critical training samples that enhance generalization. Our approach introduces a novel metric, *Loss Trajectory Correlation* (LTC), which quantifies the alignment between the loss trajectories of a training sample and those of query samples in the validation set. Since the validation set is drawn from the same distribution as the test data, it serves as a proxy for generalization. By prioritizing training samples with high LTC values, we identify subsets of data that drive effective generalization.

Our method achieves computational efficiency through two mechanisms. First, loss trajectories of training samples are inherently generated during the training process, requiring no additional computation. Second, the intrinsic features captured by LTC is a property of the dataset and is independent of the learning mechanism, hence it is highly transferable across architectures. This transferability allows us

<sup>1</sup>Electrical and Computer Engineering, Purdue University, West Lafayette, USA. Correspondence to: Manish Nagaraj <mna-gara@purdue.edu>.

to use smaller, efficient models to establish relationships among samples, enabling the identification of coresets that are representative of the full dataset and suitable for training larger, more complex models. By combining efficiency, scalability, and the ability to identify impactful data subsets, our approach enhances model performance while offering valuable insights into the structure of training data.

In this work, we make the following key contributions:

- 1. Loss Trajectory Correlation for Generalization:** We introduce LTC, a novel metric to identify critical training samples by quantifying the alignment of their loss trajectories with those of a validation set. By prioritizing samples with high positive LTC, our method effectively identifies subsets of data that drive generalization.
- 2. Efficient and Transferable Coreset Selection:** LTC enables the construction of compact, representative coresets that retain model performance while reducing computational demands. Across CIFAR-100 and ImageNet-1k, LTC consistently delivers competitive accuracy, matching or outperforming state-of-the-art methods for most coreset sizes. Even in cases where LTC does not achieve the highest accuracy, its performance remains within 1% of the best method. We demonstrate that LTC-based coresets are transferable across architectures, including ResNet, VGG, DenseNet, and Swin Transformer, with minimal performance drops (< 2%).
- 3. Computational and Storage Efficiency:** LTC achieves exceptional scalability by utilizing the loss trajectories of training samples, which are inherently generated during the training process, and only computing the loss trajectories of query samples during evaluation. This results in a computational overhead limited to additional forward passes for the query samples, significantly fewer in number than the training samples, across all epochs. Its storage overhead scales linearly with the product of the dataset size and training epochs. In contrast, competitive methods, such as those relying on pairwise similarity (Iyer et al., 2021), feature space distances (Margatina et al., 2021), or repeated gradient computations (Garg & Roy, 2023), incur substantially higher costs due to their reliance on expensive operations for all training samples. By minimizing these overheads, LTC effectively balances scalability and efficiency, making it an optimal choice for large-scale datasets and resource-constrained applications.
- 4. Insights into Training Dynamics:** We empirically show that LTC provides insights similar to traditional Training Data Attribution (TDA) metrics, such as identifying aligned and conflicting sample dynamics, but at significantly reduced computational cost. This insight

into the link between TDA and LTC establishes LTC as an excellent choice for coreset selection.

## 2. Related Literature

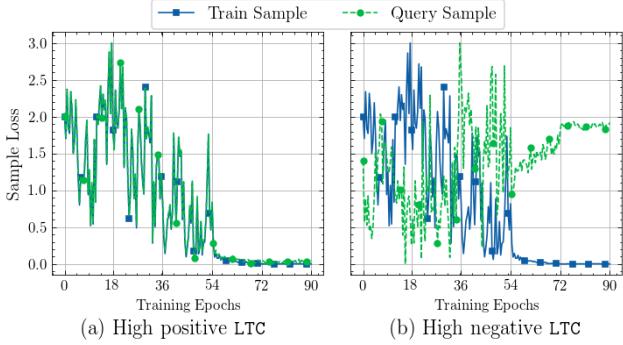
Existing coreset selection techniques can be broadly categorized into score-based, optimization-based, and training property-based methods.

**Score-based** methods select training samples based on pre-defined metrics in the feature space, but they often rely on heuristics that may not reflect the true data distribution. Common approaches include selecting samples based on their proximity to class centers or decision boundaries (Rebuffi et al., 2017; Castro et al., 2018; Belouadah et al., 2020). Examples of such methods include DeepFool (Moosavi-Dezfooli et al., 2016) and BoundarySetCCS (Yang et al., 2024). Other methods leverage the model predictions. D<sup>2</sup>Pruning (Maharana et al., 2024) utilizes the difficulty of prediction as a metric. In contrast, (He et al., 2024) utilizes the uncertainty of prediction, and E2LN (Paul et al., 2021) measures the  $L_2$  norm of the prediction error. While these techniques offer valuable insights, they face scalability issues with large datasets and risk introducing biases toward specific data subsets. To address these limitations, methods like GraNd (Paul et al., 2021), GradMatch (Killamsetty et al., 2021a), Forgetting (Toneva et al., 2018), and CRAIG (Mirzasoleiman et al., 2020) incorporate network-driven metrics, such as gradient norms. However, these approaches often depend heavily on fixed metrics, reducing their adaptability throughout the training process.

**Optimization-based** methods, such as Glister (Killamsetty et al., 2021b), LBCS (Xia et al., 2024), and (Borsos et al., 2020), are grounded in robust theoretical frameworks but involve computationally expensive formulations, such as bilevel optimization, limiting their feasibility for large-scale datasets.

**Training property-based** methods identify influential samples by examining their impact on generalization. Techniques such as Slocurv (Garg & Roy, 2023) utilize second-order loss statistics to identify samples better suited for generalization, while TracIn (Pruthi et al., 2020) tracks gradient alignment between training samples and a validation set. Despite their effectiveness, these methods often rely on computationally intensive first-order (gradients) or second-order (curvature, Hessian) metrics, limiting their scalability for large datasets.

By introducing LTC, our method addresses these limitations by offering a computationally lightweight yet effective alternative for coreset selection. It is designed to balance efficiency and effectiveness, making it applicable across a range of deep learning applications.



**Figure 1.** Examples of loss trajectories of train and query sample pairs with high positive and high negative LTC: (a) shows high positive LTC, where train (solid blue) and query (dashed green) sample losses decrease together, indicating aligned learning dynamics. (b) shows high negative LTC, where reductions in the train sample’s loss correspond to increases in the query sample’s loss, highlighting conflicting relationships. These examples demonstrate how LTC captures inter-sample influences during training. (Best viewed in color.)

### 3. Loss Trajectory Correlation (LTC)

#### 3.1. Overview

We begin by establishing key notations used throughout this paper for clarity and consistency. Random variables will be represented in bold ( $\mathbf{V}$ ), with scalar instances denoted by lowercase letters ( $v$ ), and vectors by arrowed letters ( $\vec{v}$ ).

Consider a randomized learning algorithm  $\xi$  (e.g., SGD, Adam) training over  $T$  epochs with training parameters  $\lambda$  on a dataset  $S \sim \mathbf{Z}$ ,  $S = \{\vec{z}_1, \dots, \vec{z}_N\}$ , where  $\vec{z}_m$  is the  $m^{\text{th}}$  sample. Let  $\theta_S^t \in \mathbb{R}^p$  be the model parameters at epoch  $t$ , and the loss function  $\ell(\theta_S^t, \vec{z})$  and its gradient  $\nabla_{\theta} \ell(\theta_S^t, \vec{z})$  be:

$$\ell(\theta_S^t, \vec{z}) : \mathbb{R}^p \times \mathcal{Z} \rightarrow \mathbb{R}, \quad \nabla_{\theta} \ell(\theta_S^t, \vec{z}) \in \mathbb{R}^p$$

For a sample  $\vec{z}$ , the loss change between epochs  $t - 1$  and  $t$  is:

$$\Delta \ell(\theta_S^t, \vec{z}) = \ell(\theta_S^t, \vec{z}) - \ell(\theta_S^{t-1}, \vec{z})$$

Given a training sample  $\vec{z}_m \in S$  and a query (test or validation) sample  $\vec{z}_q \notin S$ , the LTC score is defined as the correlation ( $\rho$ ) between their loss change trajectories:

$$\text{LTC}(\vec{z}_q, \vec{z}_m, S, \lambda) := \rho \left( \{\Delta \ell(\theta_S^t, \vec{z}_m)\}_{t=1}^T, \{\Delta \ell(\theta_S^t, \vec{z}_q)\}_{t=1}^T \right) \quad (1)$$

For our experiments, we chose to use Pearson’s correlation (Pearson, 1895) due to its scale invariance and ease of computation.

#### 3.2. Intuition Behind the LTC Metric

The Loss Trajectory Correlation (LTC) metric offers a powerful lens through which to understand how a model’s learning dynamics for one sample influence its performance on another. It quantifies the relationship between the changes in loss for a training sample  $\vec{z}_m$  and a query sample  $\vec{z}_q$  across training. At its core, LTC measures the degree to which the loss trajectories of these two samples align or diverge during the learning process.

**Positive LTC: Aligned Learning Dynamics** A positive LTC value indicates that the loss trajectories for  $\vec{z}_m$  and  $\vec{z}_q$  exhibit synchronized behavior (shown in Figure 1 (a)). Specifically, if the model’s loss on  $\vec{z}_m$  decreases at a given epoch, the loss on  $\vec{z}_q$  tends to decrease as well, and vice versa. This alignment suggests that the features of  $\vec{z}_m$  are highly relevant to predicting  $\vec{z}_q$ , and learning from  $\vec{z}_m$  enhances the model’s performance on  $\vec{z}_q$ . Such positive correlations often arise when the two samples share similar patterns, structures, or features, and the model leverages knowledge gained from  $\vec{z}_m$  to infer on  $\vec{z}_q$  better.

**Negative LTC: Conflicting Learning Dynamics** Conversely, a negative LTC value reveals opposing loss trajectories (shown in Figure 1 (b)). In this case, reducing the loss on  $\vec{z}_m$  increases the loss on  $\vec{z}_q$ , indicating conflicting learning dynamics. This phenomenon may occur when  $\vec{z}_m$  and  $\vec{z}_q$  encode dissimilar or contradictory features, causing the model to prioritize one at the expense of the other. Such conflicts provide insight into potential trade-offs in the learning process and highlight the diverse interactions between samples during training.

**Why LTC Matters** The intuitive patterns captured by LTC, a positive correlation for aligned samples and a negative correlation for conflicting ones, make it a valuable tool for analyzing the inter-sample influences within a dataset. By examining the LTC scores between a validation set and training samples, researchers can identify which training samples contribute most effectively to generalization and which introduce conflicting signals. This is illustrated in Figure 2, where randomly chosen query samples in ImageNet-1k and their most influential training samples (with the highest positive and negative LTC) within the same class are shown. This insight can be leveraged to construct influential coresets, enabling more efficient training by retaining only the most impactful samples.

In summary, LTC provides a principled way to probe and interpret the relationships between samples in the context of model learning. It allows practitioners to trace how individual samples affect others during training and offers actionable insights into dataset optimization and model behavior.

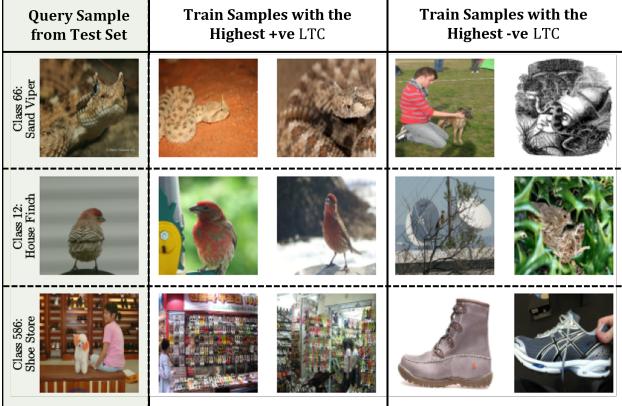


Figure 2. Examples of randomly chosen query samples from the test set in ImageNet-1k and their corresponding training samples within the same class with the highest positive LTC (closely aligned learning dynamics) and highest negative LTC (conflicting learning dynamics). This visualization highlights the alignment and contrast between query samples and training samples in terms of their feature relevance during the learning process. The LTC values were calculated using ResNet-18.

## 4. Coreset Identification

### 4.1. Methodology

The LTC metric offers a systematic approach to identifying impactful training samples by quantifying their influence on the validation set through an analysis of loss trajectories.

As detailed in Algorithm 1, a source model  $\theta$  is trained on the dataset  $S$  while recording the loss trajectories for both training and validation ( $\mathcal{V}$ ) samples. The LTC score is calculated as the correlation between the loss trajectories of each training sample and each validation sample. To rank and select impactful samples, the average LTC score for each training sample is computed, allowing the top- $k$  samples to be identified and used to form the coresset.

A key advantage of this approach lies in its flexibility, the source model used to compute LTC does not need to match the model for which the coresset will ultimately be applied. This enables the method to be employed across diverse architectures and tasks, enhancing its practical utility.

### 4.2. Experimental Setup

To evaluate the effectiveness of the proposed metric in identifying coressets, we conducted experiments on two widely-used datasets: CIFAR-100 (Krizhevsky et al., 2009) and ImageNet-1k (Russakovsky et al., 2015). CIFAR-100 contains 50,000 training samples and 10,000 test samples distributed across 100 classes, while ImageNet-1k is a significantly larger dataset with 1,281,167 training samples and 50,000 validation samples across 1000 classes. Unless stated otherwise, all experiments used the ResNet-18 (He et al., 2016) architecture.

### Algorithm 1 Coreset Selection Using LTC

```

Input: Training set  $S = \{\vec{z}_m\}_{m=1}^N$ , Validation set  $\mathcal{V} = \{\vec{z}_q\}_{q=1}^Q$ , Coreset size  $k$ , Number of epochs  $T$ , Loss function  $\ell$ , Randomized Learning Algorithm  $\xi$ , Training parameters  $\lambda$ , source model  $\theta$ 
Output: Coreset  $\mathcal{C}$  of size  $k$ 

1: // Train Model and Record Loss Trajectories
2: for epoch  $t = 1, \dots, T$  do
3:    $\theta_S^t \leftarrow \xi(\theta_S^{t-1}, S, \lambda)$  ▷ Update model weights
4:   for each sample  $\vec{z}_m \in S$  do
5:     Compute and store  $\ell(\theta_S^t, \vec{z}_m)$ 
6:   end for
7:   for each sample  $\vec{z}_q \in \mathcal{V}$  do
8:     Compute and store  $\ell(\theta_S^t, \vec{z}_q)$ 
9:   end for
10:  end for

11: // Calculate LTC Scores
12: Initialize  $\text{LTC}(q, m) \leftarrow 0, \forall m \in S, q \in \mathcal{V}$ 
13: for each  $\vec{z}_m \in S$  and  $\vec{z}_q \in \mathcal{V}$  do
14:    $\text{LTC}(q, m) \leftarrow \rho(\{\Delta\ell(\theta_S^t, \vec{z}_m)\}_{t=1}^T,$ 
15:    $\{\Delta\ell(\theta_S^t, \vec{z}_q)\}_{t=1}^T)$ 
16: end for

17: // Aggregate and Rank Training Samples
18: for each  $\vec{z}_m \in S$  do
19:    $\text{LTC}_{\text{Avg}}(m) \leftarrow \frac{1}{Q} \sum_{q=1}^Q \text{LTC}(q, m)$ 
20: end for
21: ▷ Rank training samples  $\{\vec{z}_m\}_{m=1}^N$  by  $\text{LTC}_{\text{Avg}}(m)$ 

22: // Select Top k Samples
23:  $\mathcal{C} \leftarrow \text{Top } k \text{ samples from } S \text{ by the above ranking}$ 
24: return  $\mathcal{C}$ 

```

We evaluated coresets of varying sizes: 0.2% to 10% of the full dataset for CIFAR-100 and 0.1% to 30% for ImageNet-1k. Each coresset was constructed to maintain class balance, ensuring equal representation across all classes. The training configuration, including augmentation strategies and optimizer details, is detailed in Appendix A.

### 4.3. Comparison to Baseline Methods

We compared our approach, LTC, to several state-of-the-art coresset selection techniques, including Glister (Kilamsetty et al., 2021b), Forgetting (Toneva et al., 2018), GraphCut (Iyer et al., 2021), Cal (Margatina et al., 2021), GraNd (Paul et al., 2021), Herding (Chen et al., 2010), and Slocurv (Garg & Roy, 2023). Implementations of these methods were sourced from the DeepCore library (Guo et al., 2022). To ensure uniformity, all methods were evaluated under the same experimental conditions, including identical training and testing setups. For coresset generation methods requiring pretraining, models were trained for 40 epochs to achieve convergence under identical conditions across all techniques. No additional fine-tuning or regularization was applied to ensure fair comparisons.

**Results and Observations:** As shown in Figure 3, LTC

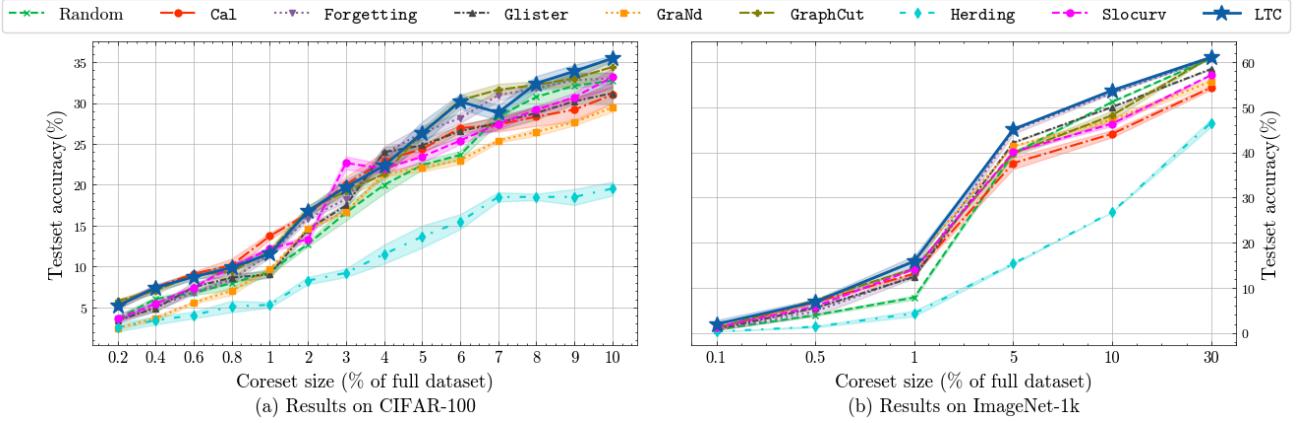


Figure 3. Comparison of test accuracy across various coresets selection methods on the CIFAR-100 and ImageNet-1k datasets. The proposed approach consistently outperforms or matches existing techniques for all evaluated dataset sizes, demonstrating its effectiveness. The shaded regions in the plots represent the standard deviation across five random seeds. (Best viewed in color.)

demonstrated exceptional performance across a wide range of coresets sizes. On CIFAR-100, LTC consistently outperformed or matched the best-performing baselines, particularly for moderate to larger coresets (e.g., 5%–10% of the dataset). At 5%, LTC achieved a mean accuracy of 26.33%, surpassing both Cal and GraphCut. At 10%, LTC reached 35.47%, exceeding the next best performers GraphCut by 1.06% and SloCurv by 2.30%. For smaller coresets, LTC remained competitive, with performance differences of less than 1% compared to the leading methods Cal and GraphCut.

On ImageNet-1k, LTC maintained a leading or near-leading position across all coresets sizes. At even small subsets (e.g., 0.1%), it achieved 1.95% accuracy. For medium coresets (e.g., 5%–10%), LTC achieved 45.15% and 53.78%, outperforming most methods except SloCurv at 5%. Even for large coresets (e.g., 30%), LTC remained competitive, achieving 61.11%, only 0.13% behind GraphCut.

For a detailed numerical breakdown of the results, please refer to Appendix A, where the same findings are presented in tabular format for clarity and precision.

**Takeaways:** The experimental results highlight the robustness and effectiveness of LTC across both CIFAR-100 and ImageNet-1k datasets. It consistently ranked as the top or near-top method across all evaluated coresets sizes. This adaptability and superior performance across varying dataset scales and coresets sizes underscore the versatility of LTC in identifying representative subsets for diverse applications.

#### 4.4. Transferability across Architectures

As discussed in Section 4.1, the coresets are identified using a source model  $\theta$ . An advantage of using LTC becomes evident when we can effectively use a smaller, more efficient source model to identify coresets for training a larger, more

complex target model. This is especially useful in scenarios where the target model has significantly more parameters and requires greater computational resources for training.

The rationale behind this approach stems from the observation that, irrespective of the specific architectures employed (e.g., deep neural networks or vision transformers), data points with similar characteristics exhibit similar feature representations. Consequently, the LTC scores, which quantify the relative importance of data points, should remain consistent across different architectures. This hypothesis aligns with the inherent property of feature similarity and serves as the foundation for our empirical studies.

To validate this hypothesis, we conducted experiments where we used ResNet-18 as the source model to compute coresets of varying sizes on the ImageNet-1k dataset. These coresets were then utilized to train diverse target architectures. The target architectures included ResNet-34 and ResNet-50, VGG-19 with batch normalization (referred to as VGG-19(bn)), DenseNet-121, and Swin-T Transformer. The results using the ResNet-18 source model were compared to results obtained when the respective target models themselves were used as source models. This comparative analysis allowed us to assess the impact of the source model’s architecture on the effectiveness of the coresets for training various target models.

**Results and Observations:** The results presented in Figure 4 confirm the high transferability of coresets identified by a smaller source model, to various target architectures. Across all evaluated target models and coresets sizes, the performance achieved using coresets identified by ResNet-18 consistently exhibited minimal accuracy drops when compared to those identified using the respective target models as source models. For instance, in the case of ResNet-34, the accuracy at a 10% coresset size was 54.13% using ResNet-18 as the source model, compared to 55.31% when

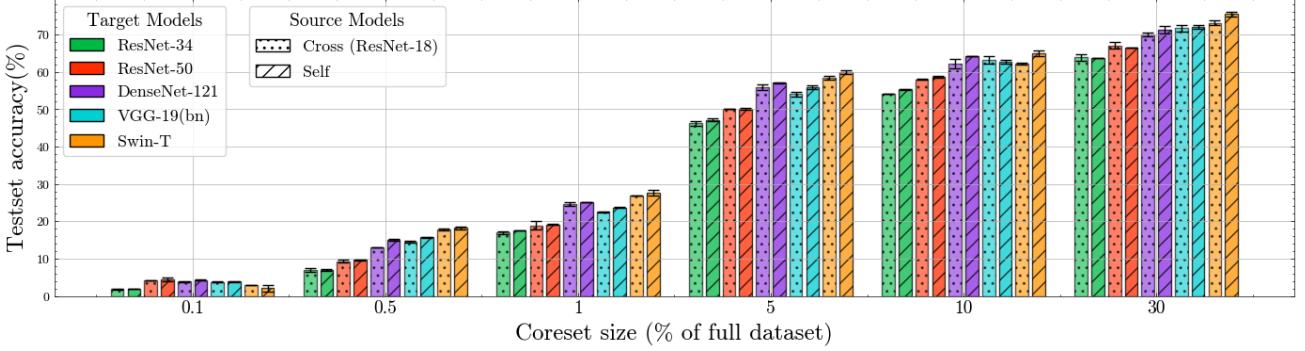


Figure 4. Comparison of the performance of coresets of different sizes of ImageNet-1k on different target models (shown in different colors) identified by the same architecture as the source model (shown as striped bars) to the coresets identified by a different (smaller) source model, ResNet-18 (shown as dotted bars). The error bars show the variance of the performance over 5 runs. We see a very minimal drop in accuracy when the smaller source models is used. (Best viewed in color.)

ResNet-34 was used as the source model, representing a negligible difference of 1.18%. Similarly, for other target models, including DenseNet-121, VGG-19.bn, and Swin-T, the accuracy differences remained within 2% across all coreset sizes, even for small subsets like 0.1% and 0.5%. Importantly, this trend held consistently across diverse architectures, including convolutional neural networks and vision transformers, underscoring the architecture-agnostic nature of the coresets identified using LTC. Moreover, the results exhibited low variance across multiple experimental runs, reinforcing the reliability of the approach. These findings validate the hypothesis that coresets generated by LTC retain their effectiveness across different target architectures, regardless of the source model used to identify them.

For a detailed numerical breakdown of the results, please refer to Appendix A, where the same findings are presented in tabular format for clarity and precision.

**Takeaways:** The experimental results demonstrate that coresets identified by LTC using a smaller source model, such as ResNet-18, are highly transferable and exhibit minimal performance degradation when used to train larger and more complex target architectures. This not only highlights the robustness of LTC but also emphasizes its practical utility in enabling resource-efficient coreset identification.

#### 4.5. Computational and Storage Overheads

The computational overhead of LTC is minimal, as the loss trajectories of training samples are inherently generated during the training process. For query samples, their loss trajectories are computed during evaluation, requiring only forward passes across epochs. In contrast, competitive methods like Ca1 and GraphCut involve costly nearest-neighbor searches and pairwise similarity computations, while Slocurv relies on repeated gradient-based calculations. Table 1 summarizes the overheads of these methods, with detailed derivations in Appendix B.

Table 1. Comparison of computational and storage overheads of LTC with existing coreset generation methods. The details of the notation are outlined in Section 4.5.

Method	Computational Overhead	Storage Overhead
Glister	$O\left(\frac{NQTf}{\gamma} \log(1/\epsilon)\right)$	$O(Q)$
Forgetting	-	$O(NT)$
GraphCut	$O(N^2k)$	$O(N^2)$
Ca1	$O(NQd)$	$O(Nd)$
GraNd	$O(3NTRf)$	$O(NTRp)$
Herding	$O(NTd)$	$O(Nd)$
Slocurv	$O(3NRf)$	$O(NRd)$
LTC (Ours)	$O(QTf)$	$O(NT)$

**Notation and Setup:** Let  $N$  and  $Q$  denote the number of training and query samples, respectively, and  $T$  the number of training epochs. Computational complexities are expressed in terms of *floating-point operations per second* (FLOPs) proportional to these variables. The storage overheads reflect the additional disk space required for dataset and model parameters ( $p$ ). Other parameters include  $k$ , the coreset size;  $\gamma$ , the frequency of coreset generation (used in Glister);  $d$ , the input dimensionality;  $c$ , the number of classes; and  $R$ , the number of retrainings required for GraNd and Slocurv. Additionally,  $f$  refers to the FLOPs required for a single forward pass through the model, and, a backward pass is approximately twice as expensive as a forward pass and involves  $2f$  FLOPs.

**Results and Observations:** LTC demonstrates significantly lower computational and storage overheads compared to competitive methods. Specifically, GraphCut requires  $O(N^2k)$  operations due to pairwise similarity evaluations, while Ca1 incurs  $O(NQd)$  operations for feature space distance computations, and Slocurv scales as  $O(3NRf)$  because of repeated gradient evaluations. In contrast, LTC

only scales as  $O(QTf)$ . Due to the fact that  $N \gg Q$  in large datasets and  $T$  is typically small, LTC incurs significantly lower overheads compared to competing methods. For instance, on ImageNet-1k with ResNet-18, identifying a coresnet of 10% size, LTC incurs computational and storage overheads of around 8 PFLOPs<sup>1</sup> and 0.4 GB<sup>2</sup>, respectively. In comparison, GraphCut and Slocurv require around 210 and 69 PFLOP operations, with storage footprints of around 6500 – 7700 GB. Although Ca1 has computational overheads closer to LTC, its storage requirements (around 771 GB) are significantly higher. Detailed calculations for this example are provided in Appendix B.1.

**Takeaways:** LTC provides a computationally and memory-efficient approach to coresnet selection, outperforming methods like GraphCut, Slocurv, and Ca1 in scalability and resource efficiency. These advantages make LTC well-suited for large-scale and resource-constrained applications.

## 5. Discussion

In this section, we delve deeper into the insights provided by LTC regarding training dynamics, specifically its ability to identify the impact of training samples on generalization.

### 5.1. Comparison of LTC with Influence

The impact measured by LTC closely aligns with the “*influence*” of individual training samples on a model’s predictions that are measured by *Training Data Attribution* (TDA) methods. TDA methods have been widely employed for tasks such as debugging datasets, interpreting models, and optimizing training efficiency (Koh & Liang, 2017; Yeh et al., 2018; Feldman & Zhang, 2020). Early approaches, like *Leave-One-Out* (LOO) retraining, involved repeatedly retraining models after removing specific data points but were computationally impractical for modern deep learning. Recent metrics, such as FZ-Influence (Infl) (Feldman & Zhang, 2020) and Datamodels (Ilyas et al., 2022), introduced precomputed scores for popular datasets but struggled with scalability. *Influence Functions* offered an alternative by estimating sample effects using gradient and Hessian computations, with efficiency improvements from methods like RandSelect (Wojnowicz et al., 2016), Arnoldi iterations (Schioppa et al., 2022), and TRAK (Park et al., 2023). However, these approaches relied on strong assumptions and incurred high computational costs. *Unrolling-based methods*, such as TracIn (Pruthi et al., 2020) addressed some of these issues by tracking gradients throughout training, but their need to store intermediate training states resulted in significant memory and computational overhead, limiting their practicality for large-scale applications. A detailed

overview of TDA metrics is provided in Appendix C.

In contrast, LTC solely relies on loss trajectories rather than first- or second-order quantities (e.g., gradients and Hessians). To compare the impact measured by LTC to the influence measured by TDA metrics, we utilize the *linear datamodeling score* (LDS) introduced by (Park et al., 2023). LDS measures the correlation between group-level attribution scores (LTC or influence) and their observed impact on model predictions when subsets of training data are used.

**LDS definition** For a query data point  $z_q$ , random subsets  $\{S_j\}_{j=1}^C$  are sampled from the training dataset, where each subset  $S_j$  contains  $\lceil \alpha N \rceil$  points, with  $\alpha \in (0, 1)$  as the sampling ratio. Each subset  $S_j$  is used to retrain the model  $R$  times with different initializations  $\{\xi_r\}_{r=1}^R$  and training parameters  $\lambda$ , resulting in the model  $\theta_{S_j, \xi_r}^T$ . This trained model is then used to compute a measurable quantity  $f(\vec{z}_q, \theta_{S_j, \xi_r}^T)$ . A *group attribution score*,  $g_\tau(\vec{z}_q, S_j, S; \lambda)$ , is calculated as  $g_\tau(\vec{z}_q, S_j, S; \lambda) := \sum_{\vec{z} \in S_j} \tau(\vec{z}_q, \vec{z}, S; \lambda)$ , where  $\tau(\vec{z}_q, \vec{z}, S; \lambda)$  is the attribution score for a training point  $\vec{z}$  with respect to  $\vec{z}_q$ . The LDS is then obtained using Spearman’s rank (Spearman, 1904) correlation ( $\rho_s$ ):

$$\text{LDS}(\vec{z}_q, \alpha) := \rho_s \left( \left\{ \frac{1}{R} \sum_{r=1}^R f(\vec{z}_q, \theta_{S_j, \xi_r}^T) \right\}_{j=1}^C, \{g_\tau(\vec{z}_q, S_j, S; \lambda)\}_{j=1}^C \right)$$

**Experimental Setup:** We compared the LDS scores of LTC against those of TRAK, Arnoldi, TracIn, Infl, and Datamodels. Precomputed scores for Infl and Datamodels were used for the CIFAR-10 dataset (Krizhevsky et al., 2009) with ResNet-9 (He et al., 2016), while 10 models were trained for TRAK, Arnoldi, TracIn, and LTC. The evaluation employed  $C = 100$  random subsets, sampling ratios  $\alpha$  ranging from 0.3 to  $\frac{N-1}{N}$ , a query set of 200 samples, and  $R = 10$  seeds. The measurable quantity was the accuracy of query samples.

**Results and Observations:** The results presented in Figure 5 reveal that while the impact captured by LTC is distinct from the influence measured by traditional TDA metrics, it aligns closely with methods such as TracIn and TRAK in terms of behavior while being resource-efficient (explained in detail in Appendix C). Notably, the performance gap between these computationally intensive methods and LTC narrows as  $\alpha$  increases. The drop in LDS scores at  $\alpha = \frac{N-1}{N}$  is due to the stochastic nature of model retraining<sup>3</sup>.

**Takeaways:** Although LTC fundamentally differs from influence-based TDA metrics, it mirrors their trends at higher sampling ratios while maintaining superior computational efficiency, solidifying its utility as a practical tool for

<sup>1</sup>peta FLOPs ( $10^{15}$  FLOPs)

<sup>2</sup>assuming each parameter needs 4B of storage space

<sup>3</sup>This observation is consistent with the findings of previous research (Karthikeyan & Søgaard, 2021; Bae et al., 2024).

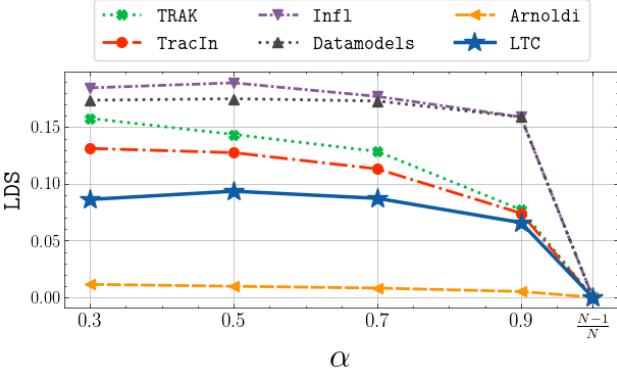


Figure 5. Linear datamodeling scores (LDS) of existing TDA metrics compared to LTC. The scores were evaluated on CIFAR-10, ResNet-9 with 200 (randomly selected) query samples evaluated over 100 subsets.

analyzing training dynamics.

## 5.2. Importance of the Top- $k$ Samples

We demonstrate that the samples identified by LTC are indeed pivotal for generalization, addressing the question: “Are the training samples with the top- $k$  scores truly the most critical for forming a coreset?” This is evaluated using the *prediction brittleness* metric.

**Experimental Setup:** To quantify the influence of top- $k$  samples, we systematically removed the most impactful data points identified by their LTC scores, from the training set and retrained the model. The metric of interest was the fraction of prediction flips observed in a held-out query set after retraining. If these samples are truly critical for generalization, their removal should cause substantial prediction changes. This experiment also included a comparative analysis with the top- $k$  influential samples identified by TDA scores, discussed in Section 5.1. Experiments were performed on the CIFAR-10 dataset using a ResNet-9 architecture, with a randomly selected query set of 200 samples. For each configuration, once the top- $k$  samples were excluded, the model was retrained 5 times to account for randomness, and the average fraction of prediction flips was recorded.

**Results and Observations:** The results, summarized in Figure 6, illustrate that the top- $k$  samples identified by LTC have a comparable influence on prediction outcomes to those identified by TDA-based metrics such as TracIn and TRAK. Notably, removing the top-800 samples of CIFAR-10, which constitutes just 1.6% of the dataset, results in prediction flips for over half of the query set. This highlights the significant role of the samples identified by LTC in supporting model generalization. While metrics like Datamodels and Infl exhibit greater impact, they are computationally prohibitive, rendering them unsuitable for large-scale coresset generation.

**Takeaways:** LTC emerges as an effective and computation-

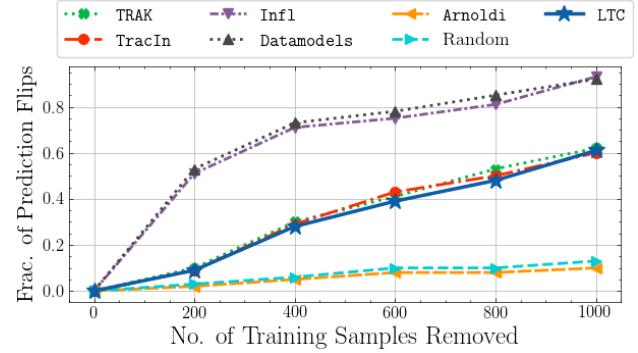


Figure 6. Prediction brittleness of LTC and TDA metrics on CIFAR-10 with ResNet-9. The top- $k$  influential training samples were removed, and the average prediction flips for 200 query samples over 5 seeds are shown.

ally efficient approach for identifying training samples critical to generalization, making it a practical tool for coresset selection in large-scale machine learning pipelines.

## 6. Limitations

While LTC offers a scalable and effective method for quantifying the influence of training samples on test performance, it is not without limitations. LTC assumes access to sufficient training checkpoints to monitor loss dynamics throughout training. This requirement can impose constraints in scenarios where training checkpoints are unavailable.

## 7. Conclusion

In this work, we introduced Loss Trajectory Correlation (LTC), a novel and efficient metric for coresset selection. By identifying training samples with high LTC values, our approach constructs compact and representative subsets that preserve generalization while reducing computational and memory demands. On CIFAR-100 and ImageNet-1k, LTC consistently delivers competitive accuracy, matching or outperforming state-of-the-art methods, and in the rare cases it doesn’t top the list, its performance is within 1% of the best. Moreover, LTC-based coressets transfer effectively across diverse architectures, including ResNet, VGG, DenseNet, and Swin Transformer, with minimal performance degradation (< 2%). A significant advantage of LTC lies in its scalability. It achieves superior computational and storage efficiency compared to traditional methods, avoiding quadratic or high-dimensional dependencies, making it well-suited for large-scale datasets. In addition to its efficiency, LTC provides insights into training dynamics, identifying aligned and conflicting samples at a low computational cost. This utility extends beyond coresset selection, offering a framework for dataset optimization and a better understanding of the role of training data in generalization.

## Acknowledgements

This work was supported in part by the Center for the Co-Design of Cognitive Systems (CoCoSys), a DARPA-sponsored JUMP 2.0 center, the Semiconductor Research Corporation (SRC), the National Science Foundation, and Collins Aerospace. We are also thankful to Amitangshu Mukherjee and Utkarsh Saxena for their helpful discussions and feedback.

## Impact Statement

This paper presents a novel approach to coresnet selection using Loss Trajectory Correlation (LTC), which addresses critical challenges in efficient dataset optimization and scalable training. In the era of ever-growing datasets and complex models, our method offers a lightweight and scalable alternative to traditional training data attribution (TDA) techniques, requiring neither gradient computations nor expensive re-training. By leveraging naturally occurring loss trajectories, LTC facilitates the selection of compact yet representative coresnets, ensuring computational efficiency without compromising model performance.

The broader implications of this work span multiple domains. LTC promotes sustainability by reducing the computational and memory overheads associated with large-scale machine learning workflows, thereby lowering energy consumption and environmental impact. Additionally, by identifying the most impactful training samples, LTC can reveal hidden biases within datasets, providing opportunities to mitigate these biases and improve fairness in model training. The proposed approach tackles a realistic and essential problem in deep learning, requiring advanced techniques and a nuanced understanding of training dynamics. While introducing a significant step forward in coresnet generation, this work adheres to ethical standards and avoids negative implications, ensuring its contributions are both impactful and responsible.

## References

- Aljundi, R., Lin, M., Goujaud, B., and Bengio, Y. Gradient based sample selection for online continual learning. *Advances in neural information processing systems (NeurIPS)*, 32, 2019.
- Bae, J., Lin, W., Lorraine, J., and Grosse, R. B. Training data attribution via approximate unrolling. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.
- Basu, S., Pope, P., and Feizi, S. Influence functions in deep learning are fragile. In *International Conference on Learning Representations (ICLR)*, 2021.
- Belouadah, E., Popescu, A., and Kanellos, I. Initial classifier weights replay for memoryless class incremental learning. In *BMVC 2020-The 31st British Machine Vision Virtual Conference (BMVC)*, pp. 1–13, 2020.
- Borsos, Z., Mutny, M., and Krause, A. Coresets via bilevel optimization for continual learning and streaming. *Advances in neural information processing systems (NeurIPS)*, 33:14879–14890, 2020.
- Bottou, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT’2010: 19th International Conference on Computational Statistics, Paris France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pp. 177–186. Springer, 2010.
- Castro, F. M., Marín-Jiménez, M. J., Guil, N., Schmid, C., and Alahari, K. End-to-end incremental learning. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 233–248, 2018.
- Cazenavette, G., Wang, T., Torralba, A., Efros, A. A., and Zhu, J.-Y. Dataset distillation by matching training trajectories. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4750–4759, 2022.
- Chen, Y., Welling, M., and Smola, A. Super-samples from kernel herding. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pp. 109–116, 2010.
- Coleman, C., Yeh, C., Mussmann, S., Mirzasoleiman, B., Bailis, P., Liang, P., Leskovec, J., and Zaharia, M. Selection via proxy: Efficient data selection for deep learning. In *International Conference on Learning Representations (ICLR)*, 2020.
- Feldman, V. and Zhang, C. What neural networks memorize and why: Discovering the long tail via influence estimation. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:2881–2891, 2020.
- Garg, I. and Roy, K. Samples with low loss curvature improve data efficiency. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 20290–20300, 2023.
- Guo, C., Zhao, B., and Bai, Y. Deepcore: A comprehensive library for coresnet selection in deep learning. In *International Conference on Database and Expert Systems Applications*, pp. 181–195. Springer, 2022.
- Hara, S., Nitanda, A., and Maehara, T. Data cleansing for models trained with sgd. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

- He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 770–778, 2016.
- He, M., Yang, S., Huang, T., and Zhao, B. Large-scale dataset pruning with dynamic uncertainty. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7713–7722, 2024.
- Hutchinson, M. F. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 18(3):1059–1076, 1989.
- Ilyas, A., Park, S. M., Engstrom, L., Leclerc, G., and Madry, A. Datamodels: Predicting predictions from training data. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*, 2022.
- Iyer, R., Khargoankar, N., Bilmes, J., and Asanani, H. Submodular combinatorial information measures with applications in machine learning. In *Algorithmic Learning Theory*, pp. 722–754. PMLR, 2021.
- Karthikeyan, K. and Søgaard, A. Revisiting methods for finding influential examples. *CoRR*, abs/2111.04683, 2021.
- Killamsetty, K., Durga, S., Ramakrishnan, G., De, A., and Iyer, R. Grad-match: Gradient matching based data subset selection for efficient deep model training. In *International Conference on Machine Learning (ICML)*, pp. 5464–5474. PMLR, 2021a.
- Killamsetty, K., Sivasubramanian, D., Ramakrishnan, G., and Iyer, R. 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.
- Koh, P. W. and Liang, P. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning (ICML)*, pp. 1885–1894. PMLR, 2017.
- Krizhevsky, A., Nair, V., and Hinton, G. Cifar-100 (canadian institute for advanced research). URL <http://www.cs.toronto.edu/kriz/cifar.html>, 2009.
- Kullback, S. and Leibler, R. A. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1): 79–86, 1951. doi: 10.1214/aoms/1177729694.
- Maharana, A., Yadav, P., and Bansal, M. D2 pruning: Message passing for balancing diversity and difficulty in data pruning. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.
- Margatina, K., Vernikos, G., Barrault, L., and Aletras, N. Active learning by acquiring contrastive examples. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 650–663, 2021.
- Mirzasoleiman, B., Bilmes, J., and Leskovec, J. Coresets for data-efficient training of machine learning models. In *International Conference on Machine Learning (ICML)*, pp. 6950–6960. PMLR, 2020.
- Moosavi-Dezfooli, S.-M., Fawzi, A., and Frossard, P. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 2574–2582, 2016.
- Na, B., Mok, J., Choe, H., and Yoon, S. Accelerating neural architecture search via proxy data. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. International Joint Conferences on Artificial Intelligence Organization, 2021.
- Park, S. M., Georgiev, K., Ilyas, A., Leclerc, G., and Madry, A. Trak: attributing model behavior at scale. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pp. 27074–27113, 2023.
- Paul, M., Ganguli, S., and Dziugaite, G. K. Deep learning on a data diet: Finding important examples early in training. *Advances in neural information processing systems (NeurIPS)*, 34:20596–20607, 2021.
- Pearson, K. VII. note on regression and inheritance in the case of two parents. *proceedings of the royal society of London*, 58(347-352):240–242, 1895.
- Pruthi, G., Liu, F., Kale, S., and Sundararajan, M. Estimating training data influence by tracing gradient descent. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:19920–19930, 2020.
- Rebuffi, S.-A., Kolesnikov, A., Sperl, G., and Lampert, C. H. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2001–2010, 2017.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115: 211–252, 2015.
- Schioppa, A., Zablotskaia, P., Vilar, D., and Sokolov, A. Scaling up influence functions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 8179–8186, 2022.

Shim, J.-h., Kong, K., and Kang, S.-J. Core-set sampling for efficient neural architecture search. *arXiv preprint arXiv:2107.06869*, 2021.

Spearman, C. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. doi: 10.2307/1412159. URL <https://www.jstor.org/stable/1412159>.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *International conference on machine learning (ICML)*, pp. 1139–1147. PMLR, 2013.

Toneva, M., Sordoni, A., des Combes, R. T., Trischler, A., Bengio, Y., and Gordon, G. J. An empirical study of example forgetting during deep neural network learning. In *International Conference on Learning Representations (ICLR)*, 2018.

Wojnowicz, M., Cruz, B., Zhao, X., Wallace, B., Wolff, M., Luan, J., and Crable, C. “influence sketching”: Finding influential samples in large-scale regressions. In *2016 IEEE International Conference on Big Data (Big Data)*, pp. 3601–3612. IEEE, 2016.

Xia, X., Liu, J., Zhang, S., Wu, Q., Wei, H., and Liu, T. Refined coreset selection: Towards minimal coreset size under model performance constraints. In *Forty-first International Conference on Machine Learning (ICML)*, 2024.

Yang, S., Cao, Z., Guo, S., Zhang, R., Luo, P., Zhang, S., and Nie, L. Mind the boundary: Coreset selection via reconstructing the decision boundary. In *Forty-first International Conference on Machine Learning (ICML)*, 2024.

Yang, W. Pytorch-classification. <https://github.com/bearpaw/pytorch-classification>, 2017.

Yeh, C.-K., Kim, J., Yen, I. E.-H., and Ravikumar, P. K. Representer point selection for explaining deep neural networks. *Advances in neural information processing systems (NeurIPS)*, 31, 2018.

## A. Detailed Results for Coreset Generation Experiments

**CIFAR-100** All networks were trained using an SGD optimizer (Bottou, 2010) for 164 epochs with a learning rate of 0.1, scaled by 0.1 at epochs 81 and 121. Nesterov momentum (Sutskever et al., 2013), with momentum of 0.9, was turned on and a weight decay of  $5e - 4$  was used. We used the following sequence of data augmentations for training:

resize to  $(32 \times 32)$ , random crop with padding = 4, random horizontal flip, and normalization.

**ImageNet-1k** The training setup for ImageNet-1k mirrored best practices for large-scale datasets. All networks were trained using an SGD optimizer for 90 epochs, with an initial learning rate of 0.1, reduced by a factor of 0.1 at epochs 30 and 60. Nesterov momentum, with a momentum coefficient of 0.9, was applied, alongside a weight decay of  $1e - 4$ . The training employed data augmentations including random resized cropping  $(224 \times 224)$ , random horizontal flipping, and normalization.

No additional finetuning or regularizing was utilized for any of the coresnet generation techniques (including ours). This was done to ensure complete fairness in comparison. All methods used the entire test set as the validation set to choose samples for the coresnet.

Each experiment was conducted over 5 independent runs with distinct random seeds, and the mean and variance of results are summarized in Tables 2 and 3.

The results of the cross-architecture performance that demonstrates the transferability of the coresets generated using LTC, discussed in Section 4.4 is shown in Table 4.

## B. Detailed Explanation of Overheads from Coreset Methodologies

**Recap on Notation** We denote the number of training samples as  $N$  ( $S \sim \mathbf{Z}^N$ ) and the number of query (test/validation) samples as  $Q$ . The model is trained for  $T$  epochs, and for certain TDA metrics, there are hyperparameters (denoted as  $\lambda_\tau$ ) that are used in calculating the TDA metric  $\tau$  that further influence the computational cost. The computational complexity of each method is expressed in terms of floating point operations per second (FLOPs), denoted as  $f$ , which represent the cost of a single forward pass for a model with  $p$  parameters. The cost of a backward pass is approximately  $2f$  FLOPs. The storage overhead is expressed in terms of the number of parameters  $p$  of the model.  $R$  is the number of model retrains required.  $k$  is the size of the coresnet that needs to be generated, while  $\gamma$  is the frequency of coresnet generation.  $d$  is the feature dimensionality of samples from the model.

**Generalization-based Data Subset Selection for Efficient and Robust Learning** (Glister (Killamsetty et al., 2021b)) formulates coresnet selection as a mixed discrete-continuous bi-level optimization problem. The goal is to select a subset  $S_j$  of size  $k$  from the training data  $S \sim \mathbf{Z}^N$ , such that a model trained on  $S_j$  generalizes well to a held-out validation set  $Q$ . The bi-level objective can be expressed

Table 2. Performance summary of various coresets generation techniques on CIFAR-100 with ResNet-18. The mean values are shown in the first line and the standard deviations are shown in the second. The highest values for each coreset sizes are highlighted in bold.

Coreset size (%. dataset)	Random	Cal	GraphCut	Glister	GraNd	Forgetting	Herding	Slocurv	LTC
0.2%	3.66 ±0.41	5.24 ±0.41	<b>5.80</b> <b>±0.24</b>	3.43 ±0.32	2.49 ±0.20	3.52 ±0.16	2.57 ±0.52	3.62 ±0.44	5.18 ±0.36
	6.03 ±0.28	<b>7.46</b> <b>±0.28</b>	7.07 ±0.39	4.91 ±0.37	3.67 ±0.29	5.12 ±0.53	3.42 ±0.49	5.46 ±0.36	7.43 ±0.43
0.6%	6.80 ±0.27	<b>9.12</b> <b>±0.27</b>	8.96 ±0.31	7.49 ±0.61	5.61 ±0.14	6.80 ±0.18	4.07 ±0.41	7.44 ±0.20	8.76 ±0.40
	7.96 ±0.79	<b>10.19</b> <b>±0.79</b>	9.39 ±0.81	8.65 ±0.47	7.03 ±0.52	8.42 ±0.38	5.14 ±0.70	9.96 ±0.42	9.88 ±0.48
0.8%	9.38 ±0.41	<b>13.73</b> <b>±0.41</b>	11.85 ±0.47	9.04 ±0.43	9.68 ±0.17	11.53 ±0.43	5.31 ±0.32	12.17 ±0.30	11.56 ±0.36
	12.74 ±0.28	16.45 ±0.28	16.95 ±0.55	14.54 ±0.41	14.63 ±0.17	15.89 ±0.21	8.29 ±0.45	13.34 ±0.42	<b>16.77</b> <b>±0.56</b>
2%	16.58 ±0.95	20.05 ±0.95	19.21 ±0.57	17.47 ±0.32	16.71 ±0.14	18.24 ±0.32	9.23 ±0.52	<b>22.67</b> <b>±0.80</b>	19.75 ±0.87
	19.99 ±1.03	22.97 ±1.03	21.33 ±0.71	<b>23.99</b> <b>±0.58</b>	22.33 ±0.31	23.82 ±0.86	11.52 ±1.15	21.97 ±0.55	22.32 ±0.82
4%	22.41 ±0.54	24.37 ±0.54	26.31 ±0.58	24.82 ±0.72	22.10 ±0.48	26.38 ±0.84	13.66 ±1.35	23.44 ±0.71	<b>26.33</b> <b>±1.40</b>
	23.66 ±0.49	26.93 ±0.49	<b>30.35</b> <b>±0.68</b>	26.57 ±0.69	23.01 ±0.47	28.16 ±0.62	15.49 ±0.91	25.41 ±0.43	30.19 ±0.92
5%	28.36 ±0.85	27.37 ±0.85	<b>31.63</b> <b>±0.71</b>	27.57 ±0.84	25.40 ±0.24	30.95 ±0.66	18.52 ±0.56	27.45 ±0.60	28.82 ±0.92
	30.75 ±1.12	28.32 ±1.12	32.22 ±0.48	28.79 ±0.79	26.47 ±0.47	31.84 ±0.23	18.52 ±0.39	29.17 ±0.47	<b>32.40</b> <b>±0.90</b>
7%	32.12 ±1.48	29.19 ±1.48	33.01 ±0.82	30.21 ±0.32	27.66 ±0.39	32.79 ±0.94	18.52 ±0.98	30.71 ±0.57	<b>33.89</b> <b>±0.80</b>
	32.75 ±1.02	31.02 ±1.02	34.41 ±0.96	31.22 ±1.33	29.45 ±0.51	33.04 ±0.65	19.54 ±0.85	33.17 ±1.16	<b>35.47</b> <b>±0.35</b>

as:

$$\arg \max_{S_j \subseteq S, |S_j| \leq k} \text{LL}_Q \left( \arg \max_{\theta^T} \text{LL}_S(\theta_{S_j}^T) \right)$$

where  $\text{LL}_S$  and  $\text{LL}_Q$  denote the training and validation log-likelihood functions, respectively.

The computational overhead in **Glister** arises from the subset selection. The outer optimization selects a subset  $S_j$  from the training set using greedy or stochastic greedy algorithms. For a training set of size  $N$ , the naive greedy algorithm has a worst-case complexity of  $O(N \cdot k \cdot Q \cdot f \cdot T/\gamma)$ , where  $k$  is the subset size and  $\gamma$  is the frequency of subset selection. Stochastic greedy methods improve this to  $O(N \cdot Q \cdot f \cdot T/\gamma \cdot \log(1/\epsilon))$ , where  $\epsilon$  is the tolerance level. Thus, the total computational overhead (in the best case with stochastic greedy optimization) for subset selection can be

expressed as :

$$\text{Computational Overhead}_{\text{Glister}} = O\left(\frac{NQTf}{\gamma} \log(1/\epsilon)\right)$$

The storage overhead for **Glister** includes storing loss values of the  $Q$  validation set samples in addition to the regular training, resulting in,

$$\text{Storage Overhead}_{\text{Glister}} = O(Q)$$

#### Example Forgetting (Forgetting) (Toneva et al., 2018)

This metric measures the number of times individual training samples transition from being correctly classified to incorrectly classified during training. This provides a way to rank examples based on their learning dynamics, with

Table 3. Performance summary of various coresnet generation techniques on ImageNet-1k with ResNet-18. The mean values are shown in the first line and the standard deviations are shown in the second. The highest values for each coresnet sizes are highlighted in bold.

Coreset size (%. dataset)	Random	Cal	GraphCut	Glistner	GrNd	Forgetting	Herding	Slocurv	Ours LTC
0.1%	0.70 ±0.03	1.13 ±0.12	1.09 ±0.09	0.86 ±0.01	0.97 ±0.01	0.64 ±0.01	0.31 ±0.01	1.23 ±0.06	<b>1.95</b> ±0.90
0.5%	3.98 ±0.19	6.84 ±0.13	<b>7.27</b> ±0.03	5.55 ±0.05	6.81 ±0.20	4.78 ±1.01	1.39 ±0.17	5.90 ±0.07	6.91 ±0.56
1%	7.86 ±0.43	13.17 ±0.22	14.27 ±0.31	12.45 ±0.01	16.07 ±0.23	12.67 ±0.51	4.32 ±0.62	14.17 ±0.02	<b>15.89</b> ±1.21
5%	39.78 ±0.23	37.65 ±1.30	39.80 ±0.60	42.19 ±0.03	41.41 ±0.14	44.85 ±0.74	15.36 ±0.18	40.10 ±0.14	<b>45.15</b> ±0.01
10%	51.24 ±0.04	44.16 ±0.78	48.27 ±1.02	50.10 ±0.01	47.28 ±0.56	53.19 ±0.06	26.84 ±0.05	46.39 ±0.50	<b>53.78</b> ±0.30
30%	60.87 ±0.13	54.41 ±0.45	<b>61.24</b> ±0.01	58.52 ±0.05	55.76 ±0.70	60.90 ±0.05	46.61 ±0.87	57.19 ±0.01	61.11 ±0.41

frequently forgotten examples considered more critical to the model’s learning process.

The computational overhead of Forgetting arises from the need to track classification performance for every training sample throughout the training process. Specifically, for a dataset  $S \sim \mathbf{Z}^N$  with  $N$  training samples, Forgetting events are tracked over  $T$  epochs of training. Since the model’s prediction of all samples are readily available during training, no additional compute overhead is added.

$$\text{Computational Overhead}_{\text{Forgetting}} = 0$$

The storage overhead of Forgetting includes storing the classification results or loss values for each training sample across all epochs resulting in

$$\text{Storage Overhead}_{\text{Forgetting}} = O(NT)$$

**GraphCut-based Data Subset Selection (GraphCut) (Iyer et al., 2021)** The GraphCut method selects representative subsets of data by leveraging a submodular objective based on a generalized graph cut function. The objective is defined as:

$$f(S_j) = \lambda \sum_{m \in S} \sum_{j \in S_j} s(m, j) - \sum_{j_1, j_2 \in S_j} s(j_1, j_2)$$

where  $s(a_1, a_2)$  is the similarity kernel between data points,  $\lambda \geq 2$  ensures monotonicity and submodularity, and  $S$  is the ground set of all data points. The subset  $S_j$  is selected to maximize the mutual information between  $S_j$  and its complement  $S \setminus S_j$  under a cardinality constraint.

The computational overhead of GraphCut with a naive greedy approach arises from evaluating the marginal gain

$f(j \mid S_j)$  for each element  $j$  in the candidate set  $S \setminus S_j$  at every iteration. For a dataset of size  $N$ , the marginal gain computation requires evaluating the similarity kernel  $s(m, j) \forall m \in S, j \in S_j$  as well as  $s(j_1, j_2) \forall j_1, j_2 \in S_j$ . Since the subset  $S_j$  has at most  $k$  elements, the cost per gain evaluation is approximately  $O(N \cdot k + k^2)$ . Over  $(N - k)$  iterations of subset selection, the total computational overhead is  $O((N \cdot k + k^2) \cdot (N - k)) = N^2k - k^3$ . Since  $N > k$ , we can express the total computational overhead in Big- $O$  notation as,

$$\text{Computational Overhead}_{\text{GraphCut}} = O(N^2k)$$

The storage overhead includes storing the similarity kernel  $s(i, j)$  for  $N$  data points, which requires  $O(N^2)$  memory. In addition, the subset  $S_j$  being constructed requires storage for  $k$  elements, and marginal gains for  $N$  elements are stored at each iteration, resulting in an additional storage cost of  $O(k + N)$ . Thus, the total storage overhead is dominated by the similarity kernel and is given by,

$$\text{Storage Overhead}_{\text{GraphCut}} = O(N^2)$$

**Contrastive Active Learning (Cal) (Margatina et al., 2021)** The Cal method selects unlabeled examples for annotation by identifying contrastive examples in the pool of unlabeled data. These are data points that are similar to labeled examples in the feature space but have maximally different predictive likelihoods. The contrastive acquisition function is defined by a two-step process: (1) finding neighbors in the feature space using a similarity measure, and (2) ranking candidates by their divergence in predictive probabilities.

Table 4. Cross-architecture performance of coresets of different sizes of ImageNet-1k identified by LTC. The mean values are shown in the first line of each cell and the standard deviations are shown in the second. The results demonstrate the transferability of coresets identified using a smaller source model (ResNet-18) as there is a minimal accuracy drop (< 2%) compared to coresets identified using source models having the same architecture as the target model.

Target Model	Source Model	Coreset size (%. of dataset)					
		0.1%	0.5%	1%	5%	10%	30%
ResNet-34	ResNet-18	1.81 ± 0.05	6.91 ± 0.51	16.98 ± 0.43	46.21 ± 0.57	54.13 ± 0.03	63.81 ± 0.91
	ResNet-34	1.90 ± 0.10	7.03 ± 0.28	17.56 ± 0.10	47.08 ± 0.41	55.31 ± 0.14	63.78 ± 0.02
ResNet-50	ResNet-18	4.12 ± 0.01	9.31 ± 0.45	18.92 ± 1.23	50.04 ± 0.01	57.91 ± 0.13	67.02 ± 0.87
	ResNet-50	4.38 ± 0.50	9.67 ± 0.13	19.14 ± 0.10	50.14 ± 0.27	58.64 ± 0.14	66.37 ± 0.01
DenseNet-121	ResNet-18	3.89 ± 0.12	13.02 ± 0.06	24.50 ± 0.51	55.92 ± 0.78	62.12 ± 1.23	69.97 ± 0.40
	DenseNet-121	4.21 ± 0.17	15.02 ± 0.31	25.07 ± 0.02	57.03 ± 0.10	64.21 ± 0.05	71.19 ± 1.03
VGG-19(bn)	ResNet-18	3.76 ± 0.14	14.41 ± 0.23	22.50 ± 0.12	54.03 ± 0.67	63.12 ± 1.05	71.68 ± 0.84
	VGG-19(bn)	3.81 ± 0.14	15.68 ± 0.04	23.71 ± 0.13	55.89 ± 0.56	62.68 ± 0.43	72.01 ± 0.52
Swin-T	ResNet-18	3.01 ± 0.02	17.75 ± 0.31	26.85 ± 0.06	58.29 ± 0.51	62.18 ± 0.17	73.12 ± 0.65
	Swin-T	2.12 ± 0.92	18.21 ± 0.31	27.61 ± 0.76	59.81 ± 0.53	64.96 ± 0.83	75.43 ± 0.59

The computational overhead of  $\text{Ca1}$  arises from two main components: nearest-neighbor searches and divergence computations. Given a labeled set  $S$  of size  $N$  and an query pool of size  $Q$ , nearest-neighbor searches involve computing pairwise distances between the feature representations of all  $Q$  unlabeled data points and all  $N$  labeled data points. Using a feature dimensionality  $d$ , this requires  $O(N \cdot Q \cdot d)$  operations for each acquisition iteration.

The second step computes the Kullback-Leibler (KL) divergence (Kullback & Leibler, 1951) between the model’s predictive probability distributions for each unlabeled data point and its neighbors. Assuming  $k$  nearest neighbors are considered, this step requires  $O(Q \cdot k \cdot c)$ , where  $c$  is the number of classes. Since  $NQd \gg Qkc$ , in big- $O$  notation,

$$\text{Computational Overhead}_{\text{Ca1}} = O(NQd)$$

The storage overhead for  $\text{Ca1}$  includes storing the feature representations for all labeled and unlabeled data points,

$(N+Q) \cdot d$ . Additionally, the KL divergence scores for all  $Q$  unlabeled candidates are stored during ranking, contributing an additional  $Q$ . Since  $Nd \gg Qd + Q$ , in big- $O$  notation,

$$\text{Storage Overhead}_{\text{Ca1}} = O(Nd)$$

**Gradient Norm-based Data Pruning (GraNd) (Paul et al., 2021)** The GraNd method identifies important examples early in training by ranking them based on their gradient norms. The score for a training sample  $\tilde{z}_m$  at time  $t$  is defined as the expected norm of the gradient of the loss function with respect to the model parameters, denoted as  $\mathbb{E}_{\theta_t} \|\nabla_{\theta_t} \ell(\theta_S^t, \tilde{z}_m)\|^2$ . GraNd uses these scores to rank and prune unimportant training examples early in training, allowing models to train efficiently on smaller subsets of data.

The computational overhead of GraNd arises from two main

components: gradient computation and score averaging. For a dataset  $S$  with  $N$  training samples and  $p$  model parameters, the gradient norm for each sample must be computed at each iteration. Assuming  $f$  represents the cost of a single forward pass and  $2f$  represents the cost of a backward pass, the cost of computing the gradient norm for all samples in one epoch is approximately  $N \cdot 3f$ . To improve robustness, GraNd scores are averaged over multiple model initializations, denoted by  $R$ . Thus, for computing the scores over  $T$  epochs and  $R$  initialization,

$$\text{Computational Overhead}_{\text{GraNd}} = O(3NTRf)$$

The storage overhead of GraNd includes storing gradient norms (of  $p$  parameters) for all  $N$  training samples across  $T$  epochs and  $R$  initializations. Thus,

$$\text{Storage Overhead}_{\text{GraNd}} = O(NTRp)$$

**Kernel Herding (Herding) (Chen et al., 2010)** selects representative samples by iteratively constructing a subset that approximates the true distribution of a dataset. Using a kernel-based approach, Herding generates a sequence of samples that greedily minimize the error in approximating the distribution in a *Reproducing Kernel Hilbert Space* (RKHS). At each iteration  $t$ , the sample  $\vec{z}_m$  is selected to maximize:

$$\vec{z}_m^{*t} = \arg \max_{\vec{z}_m \in S} \langle w_{t-1}, \phi(\vec{z}_m) \rangle$$

where  $\phi(\vec{z}_m)$  is the feature map for the kernel,  $w_{t-1}$  is the weight vector at iteration  $t - 1$ , and  $S$  is the dataset.

The computational overhead of Herding arises primarily from evaluating the kernel similarity between the current candidate  $\vec{z}_m \in S \setminus S_j$  and all previously selected samples  $S_j = \{\vec{z}_m^{*t'}, \forall t' < t\}$ . For a dataset  $S$  with  $N$  samples, the cost of computing the similarity kernel for all  $N$  data points at each of the  $T$  iterations is  $O(N \cdot T \cdot d)$ , where  $d$  is the dimensionality of the feature space. Additionally, updating the weight vector  $w_t$  at each step requires  $O(T \cdot d)$  operations. Thus,

$$\text{Computational Overhead}_{\text{Herding}} = O(NTd)$$

The storage overhead for Herding includes storing the similarity kernel for all pairs of selected samples, which scales as  $O(T^2)$ . Additionally, the feature map for all samples in the dataset must be stored, requiring  $O(N \cdot d)$  memory. Since  $Nd >> T^2$  in big- $O$  notation,

$$\text{Storage Overhead}_{\text{Herding}} = O(Nd)$$

**Using Curvature (Slocurv) (Garg & Roy, 2023)** It computes the input curvature Curv, a proxy TDA for each training sample at the end of training as

$$\frac{1}{R} \sum_R \left\| \frac{\partial (\ell(\theta_S^T, [\vec{z}_m + hv]) - \ell(\theta_S^T, [\vec{z}_m]))}{\partial \vec{z}_m} \right\|_2^2$$

Here, the hyperparameters  $R$  represent the number of “repeats” (*not model retrainings*) done to get an empirical expectation of the randomly generated  $hv$  which represents the Rademacher random variables used in Hutchinson’s trace estimator (Hutchinson, 1989). Then the samples with the lowest Curv are chosen to form the coresnet.

For each training sample ( $N$ ), and each repeat  $R$ , 2 forward passes and 1 backward pass are required. Assuming that  $\ell(\theta_S^T, [\vec{z}_m])$  can be estimated only once, this requires a total of  $R+1$  forward and  $R+1$ backward passes for each sample per epoch. Hence,

$$\text{Computational Overhead}_{\text{Slocurv}} = O(3NRf)$$

The storage overhead for Slocurv arises primarily from storing gradients and random perturbations used during the computation of the curvature for each training sample. The main memory requirement comes from storing the gradients  $\frac{\partial \ell}{\partial \vec{z}_m}$  for all samples  $\vec{z}_m \in S$ . For a dataset of size  $N$  and input dimensionality  $d$ , the gradients must be stored across  $R$  repeats, leading to a storage cost of  $O(R \cdot N \cdot d)$ . Additionally, the Rademacher random perturbations  $hv$  used in Hutchinson’s trace estimator must be stored for all  $R$  repeats, adding an additional  $O(R \cdot d)$ . Intermediate loss values  $\ell(\theta_S^T, [\vec{z}_m])$  and their perturbed counterparts  $\ell(\theta_S^T, [\vec{z}_m + hv])$  are computed and stored temporarily during forward and backward passes, but these contribute negligible additional storage compared to the gradient and perturbation storage. Since  $NRd >> Rd$ ,

$$\text{Storage Overhead}_{\text{Slocurv}} = O(NRd)$$

**Loss Trajectory Correlations (LTC)** Unlike previous core-set methodologies, we only need access to the loss values of the training ( $N$ ) and query samples ( $Q$ ) throughout training. Since loss values of the training samples are directly available during training, we only need to perform forward passes (with  $f$  FLOPs) for each epoch  $T$ , for all the query samples  $Q$ , resulting in

$$\text{Computational Overhead}_{\text{LTC}} = O(QTf)$$

In terms of storage, we need to store the loss values of all the training and query samples across the epoch, since  $N >> Q$ ,

$$\text{Storage Overhead}_{\text{LTC}} = O(NT)$$

### B.1. Summary and Example Setting

To further illustrate the computational efficiency of LTC, we provide approximate overhead values by substituting the values of the parameters for an example using the ImageNet-1k dataset ( $N = 1,281,167$ ,  $Q = 50,000$ ,  $d = 224 \times 224 \times 3$ ,  $c = 1000$ ) and a ResNet-18 model with  $p = 11,689$ , 128 parameters and  $f = 1,818,228,160$  FLOPS per forward pass. We considered the training recipe provided by the popular Bearpaw library (Yang, 2017), where  $T = 90$ . For the Slocurv, GraNd methods we used  $R = 10$ , and for Glister we used  $\gamma = 1$ ,  $\epsilon = 0.01$  as recommended by the authors. The results are summarized in Table 5. We assume each parameter requires 4B of disk space and express the computational overhead in terms of *peta* ( $10^{15}$ ) floating-point operations per second (PFLOPs).

## C. Training Data Attribution (TDA) Methods

The earliest TDA methods utilized *Leave-One-Out* (LOO) training, which involves retraining the model after removing specific data points and observing the changes in performance. While straightforward, LOO retraining is computationally prohibitive for modern deep learning models due to the need for multiple retraining cycles (Koh & Liang, 2017). Recent TDA metrics, such as FZ-Influence (Inf1) (Feldman & Zhang, 2020) and Datamodels (Ilyas et al., 2022), have gained popularity owing to precomputed scores for widely-used datasets in computer vision. These methods, however, face scalability challenges.

A prominent alternative that arose was *Influence Functions*, which estimated the effect of downweighting individual samples using first-order (gradient) and second-order (Hessian) computations (Koh & Liang, 2017; Basu et al., 2021) performed at the end of training. Methods like RandSelect (Wojnowicz et al., 2016) and Arnoldi iterations (Schioppa et al., 2022) improved computational efficiency by approximating the Hessian. Similarly, TRAK (Park et al., 2023) combined random projections, gradient-based methods, and ensembling to estimate the influence of training samples. However, these approaches often rely on strong assumptions, such as convergence to a unique optimal solution, which limits their applicability to neural networks. Additionally, Hessian computations introduce significant computational overhead. To address these challenges, *unrolling-based* methods that observe the learning process across training iterations have been proposed. These techniques approximate the impact of samples by differentiating through the optimization trajectory (Hara et al., 2019). Among these, TracIn (Pruthi et al., 2020) is a highly efficient method that estimates influence using gradients tracked throughout training. Its practical implementation, TracInCP, uses intermediate checkpoints to alleviate computational burdens. While effective, unrolling methods require storing intermedi-

ate training states, leading to high storage and computational costs.

### C.1. Computational and Storage Overheads

LTC also has lower storage and computational overhead compared to TDA metrics that measure influence. Utilizing the same notation as used in Section 4.5, we also compare the overheads of TDA metrics with LTC. This is shown in Table 6. Here  $R$  is the number of model retrainings required and  $p'$  is a lower dimension ( $p' \ll p$ ) used to alleviate computation.  $\alpha$  is a fractional value  $\in (0, 1]$  and  $b$  is the batch size used to increase computation speed. LTC exhibits significantly lower computational complexity compared to these metrics. The details are outlined below.

#### C.1.1. LOO METHODS

**FZ-Influence (Inf1)** (Feldman & Zhang, 2020) The influence score Inf1 of a training sample  $\vec{z}_m$  on a query sample  $\vec{z}_q$  is given by:

$$\text{Inf1}(\vec{z}_q, \vec{z}_m, S; \lambda) := \mathbb{E}_{\xi} \left[ \Pr(\theta_S^T(\vec{x}_q) = \vec{y}_q) - \Pr(\theta_{S \setminus m}^T(\vec{x}_q) = \vec{y}_q) \right]$$

Here,  $\Pr(\theta_S^T(\vec{x}_q) = \vec{y}_q)$  represents the confidence in the prediction for  $\vec{z}_q$  using a model trained on  $S$ , and  $S \setminus m$  is the training set with sample  $\vec{z}_m$  removed. This expression measures the change in the prediction probability for the query sample  $\vec{z}_q$  when the training sample  $\vec{z}_m$  is included in the training set compared to when it is excluded. The underlying intuition is that if sample  $\vec{z}_m$  significantly influences the prediction for  $\vec{z}_q$ , its exclusion will lead to a notable reduction in prediction accuracy for  $\vec{z}_q$ , resulting in a higher influence score. This metric ideally require re-training  $N + 1$  models to calculate the influence scores of all pairs of samples. To reduce the retraining costs, the authors of the metrics suggest training a smaller number ( $R$ ) of models with a fraction ( $\alpha$ ) of the training dataset ( $|S_j| = \lceil \alpha |S| \rceil$ ). Each of these subsets is selected randomly. They then estimate  $\Pr(\theta_S^T(\vec{x}_q) = \vec{y}_q)$  to be the average prediction probability of  $\vec{z}_q$  of all models that were trained with a subset  $S_j$  that included the sample  $m$  ( $m \in S_j$ ), and  $\Pr(\theta_{S \setminus m}^T(\vec{x}_q) = \vec{y}_q)$  to be the average prediction probability of  $\vec{z}_q$  of all models that were trained with a subset  $S_{j'}$  that excluded the sample  $m$  ( $m \notin S_{j'}$ ).

Hence the total computational overhead would be to train  $R$  models for  $T$  epochs (run forward and backward passes for each sample  $RT$  times), and for each model, we need to calculate the prediction probability (or run a forward pass) for  $\lceil \alpha N \rceil$  train samples and  $Q$  test samples. Thus,

$$\text{Computational Overhead}_{\text{Inf1}} = O(3 \lceil \alpha N \rceil QRTf)$$

Table 5. Summarized overheads of coresnet methodologies along with a quantitative illustration to demonstrate the orders of magnitude of overhead of each technique.

Method	Computational Overhead	Storage Overhead	ImageNet-1k, ResNet-18	
			Comp. Overhead	Sto. Overhead
Glister	$O(\frac{NQTF}{\gamma} \log(1/\epsilon))$	$O(Q)$	$\approx 2.10 \times 10^7$ PFLOPs	$\approx 2 \times 10^{-4}$ GB
Forgetting	-	$O(NT)$	-	$\approx 0.4$ GB
GraphCut	$O(N^2k)$	$O(N^2)$	$\approx 2.10 \times 10^2$ PFLOPs	$\approx 6.56 \times 10^3$ GB
Cal	$O(NQd)$	$O(Nd)$	$\approx 9.64$ PFLOPs	$\approx 7.71 \times 10^2$ GB
GraNd	$O(3NTRf)$	$O(NTRp)$	$\approx 6.29 \times 10^3$ PFLOPs	$\approx 5.39 \times 10^7$ GB
Herding	$O(NTd)$	$O(Nd)$	$\approx 1.74 \times 10^{-2}$ PFLOPs	$\approx 7.71 \times 10^2$ GB
Slocurv	$O(3NRf)$	$O(NRd)$	$\approx 69.9$ PFLOPs	$\approx 7.71 \times 10^3$ GB
LTC (Ours)	$O(QTf)$	$O(NT)$	$\approx 8.18$ PFLOPs	$\approx 0.4$ GB

Table 6. Comparison of computational and storage overheads of LTC with existing TDA methods. The details of the notation are outlined in Section 4.5.

Method	Computational Overhead	Storage Overhead
Inf1	$O(3\lceil\alpha N\rceil QRTf)$	$O(Rp)$
Datamodels	$O(3\lceil\alpha N\rceil RTf)$	$O(RN)$
TRAK	$O(2NRp \cdot p')$	$O(Np')$
Arnoldi	$O(2NQp \cdot p')$	$O(NQp)$
TracIn	$O(3NTf)$	$O(NTp)$
LTC (Ours)	$O(QTf)$	$O(NT)$

In addition, there would be a need to store the  $R$  model parameters for influence score calculation, resulting

$$\text{Storage Overhead}_{\text{Inf1}} = O(Rp)$$

**Datamodel Metric (Ilyas et al., 2022)** For a fixed query sample  $\vec{z}_q$ , Datamodels is a parametric function  $g_\theta$  trained to predict the outcome of a model  $\theta_{S_j}^T$  trained on a subset  $S_j$  of the training data  $S$  on  $\vec{z}_q$ . The Datamodels function  $g_\theta$  is typically instantiated as a linear function of the characteristic vector of  $S_j$ , allowing for more efficient computation and analysis.

The computational overhead of constructing Datamodels is determined by three steps: sampling subsets, training models on these subsets, and solving the regression problem to estimate the Datamodels parameters.

To construct the Datamodels,  $R$  subsets  $S_j$  of size  $\lceil\alpha N\rceil$  are sampled from the training data. A model is then trained

on each subset, and the output  $\theta_{S_j}^T(\vec{z}_q)$  is computed. This incurs a cost of  $(R \cdot T \cdot \alpha N \cdot 3f)$  where  $3f$  accounts for the forward and backward passes required during training.

The regression step fits the Datamodels function  $g_\theta$  by solving an  $l_1$ -regularized least squares problem over  $R$  samples, where the feature vector dimension corresponds to the training set size  $N$ . Using efficient solvers optimized for sparse regression, the computational cost of this step scales as  $O(NR)$ . Since  $3\lceil\alpha N\rceil RTf > NR$ , using big- $O$  notation,

$$\text{Computational Overhead}_{\text{Datamodels}} = O(3\lceil\alpha N\rceil RTf)$$

The storage overhead includes storing the  $NQ$  dimensional Datamodels training matrix, which consists of  $R$  characteristic vectors of size  $Q$ , and storing the outputs  $\theta_{S_j}^T$  for each subset. Since  $N \gg Q$ ,

$$\text{Storage Overhead}_{\text{Datamodels}} = O(RN)$$

### C.1.2. IMPLICIT DIFFERENTIATION TDA TECHNIQUES

**Tracing with the Randomly-projected After Kernel TRAK Score (Park et al., 2023)** It is a scalable data attribution method that leverages linearization via the empirical neural tangent kernel (eNTK) and random projections to approximate model behavior. It achieves data attribution by training  $R$  models, each on a subset of the training data, and computing influence scores using gradient features projected to a lower-dimensional space.

The total computational overhead for TRAK is composed of three main components: model training, gradient computation, and random projections. For model training,  $R$  models are trained on subsets of the training set of size  $\lceil \alpha N \rceil$ , where  $\alpha$  is the fraction of the training set used per subset. Each model is trained for  $T$  epochs, with the cost of a forward and backward pass for each sample being  $3f$  FLOPs. Since  $R - 1$  additional models need to be trained, the additional training cost is  $3\lceil \alpha N \rceil(R - 1)Tf$ . Next, for each of the  $R$  trained models, gradients are computed for all  $N$  training samples and  $Q$  test samples. Each gradient computation involves one forward and one backward pass, resulting in a cost of  $3R(N + Q)3f$ . The gradient features are then projected to a lower dimension  $p'$  using a random projection matrix. Each projection operation for a single sample requires the computation of a matrix-vector product of size  $p \times p'$ , which results in  $2p \cdot p'$  FLOPs (accounting for one multiplication and one addition), leading to an additional cost of  $2R(N + Q)p \cdot p'$ . Thus, in big- $O$  notation,

$$\text{Computational Overhead}_{\text{TRAK}} = 2NRp \cdot p'$$

The storage overhead for TRAK consists of two main components. First, the  $R$  trained models need to be stored, resulting in a storage cost of  $R \cdot p$ . Second, the gradients of  $N$  training samples and  $Q$  test samples are projected into a space of dimension  $p'$ . This adds a storage requirement of  $(N + Q) \cdot p'$ . Thus,

$$\text{Storage Overhead}_{\text{TRAK}} = O(Np')$$

**Arnoldi Iterations (Schioppa et al., 2022)** This method computes influence scores by approximating the inverse Hessian using Arnoldi iteration. By diagonalizing the Hessian  $H$  in a low-dimensional subspace spanned by the top eigenvectors, it aims to alleviate storage and computational overhead. The influence score between a training example  $\vec{z}_m$  and a query example  $\vec{z}_q$  is calculated as:

$$\text{Arnoldi}(\vec{z}_q, \vec{z}_m, S, \lambda) = \langle \nabla_{\theta} \ell(\theta_S^T, \vec{z}_q), H^{-1} \nabla_{\theta} \ell(\theta_S^T, \vec{z}_m) \rangle$$

where  $H = \nabla_{\theta}^2 \ell(\theta_S^T, \vec{z})$  is the Hessian of the loss  $\ell$  with respect to the model parameters  $\theta$ .

The computational overhead is primarily composed of three steps: *Hessian-vector products* (HVPs), Arnoldi iteration to diagonalize  $H$ , and projections to compute influence scores.

Each HVP computes the product  $H \cdot \vec{v}$  for a vector  $\vec{v}$  and can be efficiently implemented as a combination of reverse- and forward-mode differentiation. For a model with  $p$  parameters, the HVP computation involves backpropagating through the network for a batch of size  $b$ . The cost for a single HVP is therefore  $3bpf$ . Here, the factor of  $3f$  accounts for one forward and one backward pass per HVP. Arnoldi iteration computes the dominant eigenvectors of  $H$  over  $R$  iterations. The total cost of diagonalization, accounting for  $R$  HVPs, is  $(3Rbpf)$ .

Once  $H$  is diagonalized in a subspace of dimension  $p' \ll p$ , influence scores are computed using projections. Each projection involves a matrix-vector multiplication requiring  $2p \cdot p'$  FLOPs (accounting for one multiplication and one addition). For  $NQ$  training-query pairs, the total cost of projections is  $2NQp \cdot p'$ . Since  $2NQp \cdot p' \gg 3Rbpf$ ,

$$\text{Computational Overhead}_{\text{Arnoldi}} = O(2NQp \cdot p')$$

The storage overhead consists of storing the reduced subspace and gradients. The subspace requires  $p' \cdot p$  storage, while gradients for  $NQ$  samples require  $NQp$  storage. Since the gradients are larger, in big- $O$  notation,

$$\text{Storage Overhead}_{\text{Arnoldi}} = O(NQp)$$

### C.1.3. UNROLLED TDA TECHNIQUES

**TracIn Score (Pruthi et al., 2020)** It computes the influence as:

$$\text{TracIn}(\vec{z}_q, \vec{z}_m, S; \lambda) := \sum_{t=0}^T \nabla_{\theta} \ell(\theta_S^t, \vec{z}_m) \cdot \nabla_{\theta} \ell(\theta_S^t, \vec{z}_q)$$

This metric requires computing the gradient (one forward and one backward pass) for each sample in the train set of size  $N$ , and the query set of size  $Q$  for each epoch ( $T$  times). Since  $N \gg Q$ ,

$$\text{Computational Overhead}_{\text{TracIn}} = O(3NTf)$$

The gradients of each of the samples need to be trained for each iteration, resulting in

$$\text{Storage Overhead}_{\text{TracIn}} = O(NTp)$$