Compute-Efficient Deep Learning: Algorithmic Trends and Opportunities

Brian R. Bartoldson*   Bartoldson@llnl.gov
Lawrence Livermore National Laboratory, USA

Bhavya Kailkhura*   Kailkhura1@llnl.gov
Lawrence Livermore National Laboratory, USA

Davis Blalock*   Davis@MosaicML.com
MosaicML, USA

Editor: Samy Bengio

Abstract

Although deep learning has made great progress in recent years, the exploding economic and environmental costs of training neural networks are becoming unsustainable. To address this problem, there has been a great deal of research on *algorithmically-efficient deep learning*, which seeks to reduce training costs not at the hardware or implementation level, but through changes in the semantics of the training program. In this paper, we present a structured and comprehensive overview of the research in this field. First, we formalize the *algorithmic speedup* problem, then we use fundamental building blocks of algorithmically efficient training to develop a taxonomy. Our taxonomy highlights commonalities of seemingly disparate methods and reveals current research gaps. Next, we present evaluation best practices to enable comprehensive, fair, and reliable comparisons of speedup techniques. To further aid research and applications, we discuss common bottlenecks in the training pipeline (illustrated via experiments) and offer taxonomic mitigation strategies for them. Finally, we highlight some unsolved research challenges and present promising future directions.

Keywords: deep learning, training speedup, computational efficiency, carbon emission

1. Introduction

> "*Science is a way of thinking much more than it is a body of knowledge.*"
>
> — Carl Sagan

In the last few years, deep learning (DL) has made significant progress on a wide range of applications, such as protein structure prediction (AlphaFold, Jumper et al., 2021), text-to-image synthesis (DALL-E, Ramesh et al., 2021), text generation (GPT-3, Brown et al., 2020a), etc. The key strategy behind achieving these performance gains is scaling up DL

---

*. All authors contributed equally to this research. Bhavya led the study conceptualization and taxonomy design. Brian led the written survey of the literature. Davis conducted all experiments and led the creation of a guide to achieving speedups in practice.

models to extremely large sizes and training them on massive amounts of data. For most applications, the number of trainable parameters is doubling at least every 18 to 24 months—language models are leading with a 4- to 8-month doubling time (Sevilla and Villalobos, 2021). Notable examples of massive AI models include the following: Swin Transformer-V2 (Liu et al., 2022b) with 3 billion parameters for vision applications, PaLM (Chowdhery et al., 2022) with 540 billion parameters for language modeling, and Persia (Lian et al., 2021) with 100 trillion parameters for content recommendations.

Although scaling up DL models is enabling unprecedented advances, training large models has become extremely expensive. For example, GPT-3 training was estimated to cost \$1.65 million with Google v3 TPUs (Lohn and Musser, 2022) and inefficient/naive development of a transformer model would emit carbon dioxide (CO_2) equivalent to the lifetime carbon footprint of five cars (Strubell et al., 2019). Concerningly, DL has still not reached the performance level required by many of its applications: e.g., human-level performance is required for deploying fully autonomous vehicles in the real world but hasn't yet been reached. Growing model and data sizes to reach such required performances will make current training strategies unsustainable on financial, environmental, and other fronts. Indeed, extrapolating current trends, the training cost of the largest AI model in 2026 would be more than the total U.S. GDP (Lohn and Musser, 2022). Moreover, the heavy compute reliance of DL raises concerns around the marginalization of users with limited financial resources like academics, students, and researchers (particularly those from emerging economies) (Ahmed and Wahed, 2020). We discuss these critical issues in more detail in Appendix A.

Given the unsustainable growth of its computational burden, progress with DL demands more compute-efficient training methods. A natural direction is to eliminate algorithmic inefficiencies in the learning process to reduce the time, cost, energy, and carbon footprint of DL training. Such *Algorithmically-Efficient Deep Learning* methods could change the training process in a variety of ways that include: altering the data or the order in which samples are presented to the model; tweaking the structure of the model; and changing the optimization algorithm. These algorithmic improvements are critical to moving towards estimated lower bounds on the required computational burden of effective DL training, which are greatly exceeded by the burden induced by current practices (Thompson et al., 2020). Further, these algorithmic gains compound with software and hardware acceleration techniques (Hernandez and Brown, 2020). Thus, we believe algorithmically-efficient DL presents an enormous opportunity to increase the benefits of DL and reduce its costs.

While this view is supported by the recent surge in algorithmic efficiency papers, these papers also suggest that research and application of algorithmic efficiency methods are hindered by fragmentation. Disparate metrics are used to quantify efficiency, which produces inconsistent rankings of speedup methods. Evaluations are performed on narrow or poorly characterized environments, which results in incorrect or overly-broad conclusions. Algorithmic efficiency methods are discussed in the absence of a taxonomy that reflects their breadth and relationships, which makes it hard to understand how to traverse the speedup landscape to combine different methods and develop new ones.

To address these fragmentation issues, we eschew a more traditional survey approach that focuses on just a single component (e.g., the model) or single action (e.g., reducing model size) in the training pipeline. Instead, we adopt a wholistic view of the speedup problem and emphasize that one needs to carefully select a combination of techniques, which

we survey in Section 3, to overcome various compute-platform bottlenecks. We use experiments to illustrate the importance of such a wholistic view to achieving speedup in practice, and we provide guidance informed by the relationships between different bottlenecks and components of training. While this leads to our consideration of a broad set of topics, we limit our scope to training (not inference), algorithms (not efficient hardware or compute-kernels), trends (not every possible method), and techniques applicable to language and computer vision tasks (rather than techniques specific to RL, graphs, etc.).

Accordingly, our central contributions are an organization of the algorithmic-efficiency literature (via a taxonomy and survey inspired by Von Rueden et al., 2019) and technical characterization of the practical issues affecting the reporting and achievement of speedups (via guides for evaluation and practice). Throughout, our discussion emphasizes the critical intersection of these two thrusts: e.g., whether an algorithmic efficiency method leads to an actual speedup indeed depends on the interaction of the method (understandable via our taxonomy) and the compute platform (understandable via our practitioner's guide). Our contributions are summarized as follows:

- **Formalizing Speedup:** We review DNN efficiency metrics, then formalize the algorithmic speedup problem.

- **Taxonomy and Survey:** We classify over 200 papers via 5 speedup actions (the 5Rs) that apply to 3 training-pipeline components (see Tables 1 and 3). The taxonomy facilitates selection of methods for practitioners, digestion of the literature for readers, and identification of opportunities for researchers.

- **Best Evaluation Practices:** We identify evaluation pitfalls common in the literature and accordingly present best evaluation practices to enable comprehensive, fair, and reliable comparisons of various speedup techniques.

- **Practitioner's Guide:** We discuss compute-platform bottlenecks that affect speedup-method effectiveness. Connecting our survey and practice, we suggest appropriate methods and mitigations based on the location of the bottlenecks in the training pipeline.

With these contributions, we hope to improve the research and application of algorithmic efficiency, a critical piece of the compute-efficient deep learning needed to overcome the economic, environmental, and inclusion-related roadblocks faced by existing research. This paper is organized mainly into four parts: Section 2 provides an overview of DNN training and efficiency metrics along with a formalization of the algorithmic speedup problem. Section 3 uses broadly applicable building blocks of speedup methods and the training pipeline components they affect to develop our taxonomy. Section 4 presents a comprehensive categorization of the speedup literature based on our taxonomy and discusses research opportunities and challenges. Sections 5 and 6 discuss best evaluation practices for comparing different approaches and our practical recommendations for choosing suitable speedup methods, respectively. Finally, Section 7 concludes and presents open questions in the algorithmic-efficiency area.

Popular Speedup Approaches				Example Details	
Component	Subcomponent	Action	Example	Key Challenge	Future Direction
Function	Model parameters	Restrict	Structured parameterization (e.g., factorization)	Identifying hardware-efficient patterns	Dynamic approaches, Hardware-aware learnable parameterization
	Architecture	Remove	Remove expensive layers (e.g., NFNets)	Maintaining the performance and stability of the original model	Learning-based and dynamic approaches to identify architectural redundancies
Data	Training Data	Reorder	Curriculum learning (e.g., teaching with commentaries)	Automatic design of curriculum for new tasks/problems	Meta-learning curriculum for larger-scale datasets and new applications
	Derived Data	Remove	Gradient pruning (e.g., Selective Backprop)	Maintaining generalization performance on challenging datasets, e.g., ImageNet	Accurate selection strategies for complex datasets (e.g., ImageNet) and challenging problems (e.g., object detection)
Optimization	Training Objective	Retrofit	Adding curvature information (e.g., SAM)	Reducing the costliness of computing new update	Function approximation for reducing cost while maintaining the accuracy of new updates
	Training Algorithm	Replace	Using a better optimizer (e.g., learned optimizer)	Reducing instability and computational inefficiency	Scalable and stable training methods for bigger and more complicated models

Table 1: **Popular Approaches to Speedup.** Speedup techniques can be categorized in terms of "5R" actions applicable to three components of the training pipeline. For each component of the DNN training pipeline, we report an example speedup approach found in our literature review and corresponding details.

2. Compute-Efficient Training: Overview, Metrics, and Definition

In this section, we first provide a brief overview of the Deep Neural Network (DNN) training process. Next, we mention various metrics that quantify training efficiency and discuss their pros and cons. Finally, we formally define algorithmic speedup for DNN training.

2.1 Overview of DNN Training Process

At a high level, the goal of DL is to learn a function that can map inputs to outputs to accomplish a certain task. This function, referred to as the model, is chosen from a parametric family called the *architecture* during training. Model training is formulated as an optimization problem, where the goal of the optimizer is to find model parameters that optimize the training objective function. On each iteration, a DNN training algorithm typically performs the following steps (see Figure 1):

- Load a subset of the training dataset (referred to as a batch) from the storage device into the main memory. Apply necessary preprocessing on the CPU—e.g., JPEG decompression, data transformation, and data augmentation.

- Move the preprocessed and batched data samples to the AI accelerator (e.g., a GPU) to perform forward and backward propagation, thereby obtaining derived data such as activations and gradients:
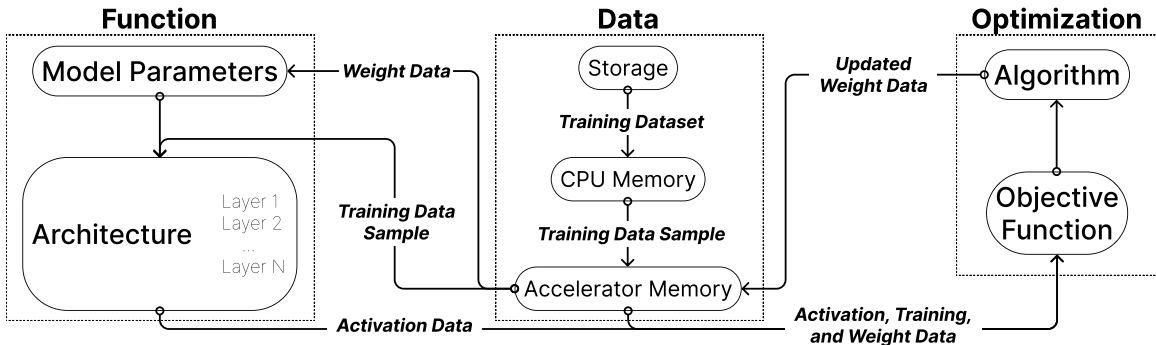
Figure 1: Illustration of data movement and computational steps needed to complete a typical iteration of neural network training.

- *Forward Pass:* traverse the computational graph in the direction of dependencies defined by the DNN. To enable the backward pass, intermediate outputs (i.e., activations) preceding trainable or nonlinear operations must be stored.
- *Backward Pass:* traverse the computational graph in reverse order, calculating (using the chain rule) and storing the gradients for intermediate variables and parameters.

- Compute the descent direction using the gradients from the backpropagation step and any state variables kept by the optimization algorithm. Combine this direction with the learning rate (step size) to update the model parameters and adjust the optimizer state as needed.

This iteration is repeated until the desired model quality or a predefined number of iterations is reached. Thus, total training time T_{total} can be calculated as:

$$T_{\text{total}} = \sum_{i=1}^{n_{\text{iteration}}} T_i,$$

where T_i is the time taken to finish iteration i and $n_{\text{iteration}}$ is the total number of training iterations required.

Recent increases in T_{total} within modern training pipelines are outpacing advancements in compute platforms, accenting the need for algorithmically-efficient solutions (Patterson et al., 2022; Menghani, 2021). Devising such efficient training procedures—e.g., to minimize redundant computations in areas that are bottlenecks on the compute platform —is thus a critical direction to explore.

2.2 Metrics to Quantify Training Efficiency

In order to devise efficient training methods, we first need a proper measure of efficiency. In this section, we briefly summarize some commonly used efficiency metrics along with their benefits and drawbacks. For a more detailed discussion on this topic, we refer readers to Schwartz et al. (2020) and Dehghani et al. (2021).

Training Time. A direct measure for evaluating efficiency is training time. This metric has the advantage of being simple to measure and coupled to real-world utility. It is also proportional to monetary cost when training on rented hardware and related to hardware depreciation when training on hardware one owns. The downside of training time is that it depends on one's precise hardware and software configuration, which makes comparison across papers dubious at best.

FLOPs. Another popular metric is the total number of floating-point operations (FLOPs) involved in a model's forward pass or the entire training process. FLOPs do not have a formal, universal definition. However, they typically reflect the number of elementary arithmetic operations present in a computation, including multiplications, additions, divisions, subtractions, and transcendental functions. An appealing property of FLOPs is that they can be computed without taking into account the hardware. This allows comparison across different hardware and software stacks. Unfortunately, FLOPs fail to capture factors such as memory access time, communication overhead, the instructions available on particular hardware, compute utilization, and more.

Number of Model Parameters. Another common measure of computational cost is the number of model parameters. Similar to FLOPs, this metric is agnostic to the hardware on which the model is being trained. Moreover, it also determines the amount of memory consumed by the model and optimizer state. However, like FLOPs, it does not account for most factors affecting cost and runtime. Further, different models with similar parameter counts may need different amounts of work to achieve a certain performance level; e.g., increasing the input's resolution or sequence length increases the compute requirements but does not change the number of parameters.

Electricity Usage. A time- and location-agnostic way to quantify training efficiency is the electricity used during DNN training. GPUs and CPUs can report their current power usage, which can be used to estimate the total electricity consumed during training. Unfortunately, electricity usage is dependent on the hardware used for model training, which makes it difficult to perform a fair comparison across methods implemented by different researchers. Moreover, even for fixed hardware, it is possible to trade off power consumption and runtime (You et al., 2022).

Carbon Emission. The carbon footprint of DNN training is another useful metric. However, accurately measuring carbon emissions can be challenging, requiring inclusion of emissions associated not just with the hardware's use but with its life cycle too (Gupta et al., 2022; Luccioni et al., 2022). Further, running the same training routine twice, consuming the same amount of electricity each time, does not imply the training runs will have the same carbon emissions due to variation in the carbon-intensity of electricity generation. Indeed, this metric depends highly on the local electricity infrastructure and current demand; therefore, one cannot easily compare results when experiments are performed in different locations or even in the same location at different times (Schwartz et al., 2020).

Operand sizes. The total number of activations in a model's forward pass is a proxy for memory bandwidth consumption and can be a useful proxy for runtime (Radosavovic et al., 2020; Dollár et al., 2021). This metric can be defined rigorously and independent of hardware

for a given compute graph. It also decreases when operators are fused, which may or may not be desirable.

2.3 Pitfalls of Different Metrics

To illustrate how some popular metrics can be misleading, we perform two microbenchmarking experiments. In the first, we profile individual PyTorch (Paszke et al., 2017) operations on a 40GB A100, including matrix products, factorized matrix products similar to those of Zhang et al. (2015), and normalization ops. In the second, we profile the time required to perform a forward pass on a batch of 128 224×224 images with a ResNet-50. Experimental details and additional experiments can be found in Appendices B and C, respectively.

In Figure 2 (left), we see the relationship between FLOP count and time for square matrix multiplies of various sizes. At each size, we also show the results of factorizing the right matrix into two low-rank matrices; this reduces the FLOP count when the rank reduction is greater than two. For matrices of size ~ 4096 or larger, FLOPs and time are linearly related. But for small matrices, they can be unrelated—the time is more a function of kernel launch overhead, memory bandwidth, and implementation details (the natural explanation for strictly smaller operations taking more time).

In Figure 2 (middle), we similarly see that the relationship between FLOPs and time can vary greatly across operations. Normalization operations, which consume a great deal of memory bandwidth per FLOP, also take much more time per FLOP—as one would expect from an elementary roofline analysis (Williams et al., 2009). The operator shapes for the normalization ops are taken from ResNet-50, augmented with channel counts increased and decreased by factors of two for extra coverage. The layer normalization and batch normalization curves coincide because the latter is implemented using the former in PyTorch. Shaded areas indicate standard deviations; these are always present but sometimes tiny due to the log scale.

In Figure 2 (right), we see the same measurements as Figure 2 (middle), but with the total size of all input and output operands used as the x-axis rather than FLOPs. This metric is a more comprehensive alternative to the number of activations that also includes the sizes of parameter tensors. This metric is still far from perfect but eliminates much of the difference across ops.

In Figure 3, we replicate the results of Figure 2 with an entire ResNet-50, rather than individual operations. Note that accuracy is not held constant in these experiments, as our focus is showing the conflict among efficiency levels suggested by different metrics. As shown in the left subplot, factorizing all of the convolutions and linear layers decreases the FLOP count significantly, but *increases* the runtime thanks to increased memory bandwidth usage and kernel launch overhead. However, removing the batch normalization ops (as is commonly done during inference) reduces time significantly—despite having almost no impact on FLOP count. A similar pattern holds for parameter count in the middle plot.

However, in Figure 3 (right), we see that measuring the size of input and output operands correctly orders the different ResNet-50 variants—though it is still not a reliable predictor of runtime. Taken together, these results indicate that the network's operations are largely bottlenecked by memory bandwidth. Removing batch normalization ops reduces data movement, while factorizing linear transforms does not, allowing differences in their speedups.
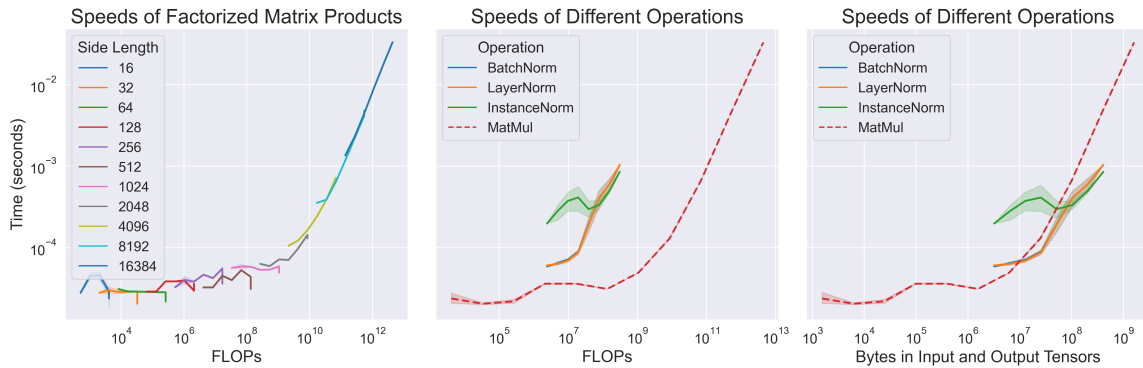
Figure 2: Some FLOPs require orders of magnitude more time than others. *(left)* Multiplying matrices of different sizes with different degrees of factorization in the right matrix. Each curve is one size and each point in the curve is one level of rank reduction, with the rightmost point corresponding to a single, full-rank matrix. Small matrix products run in time nearly independent of the FLOP count. Indeed, time and FLOPs are not even monotonically related. *(middle)* For a fixed number of FLOPs, different operations can vary in time by an order of magnitude. Normalization ops, which tax the memory bandwidth far more than the compute units, are much more expensive. *(right)* Total size of input and output operands, a proxy for memory bandwidth consumption, is a more consistent, though far from perfect, predictor of runtime.
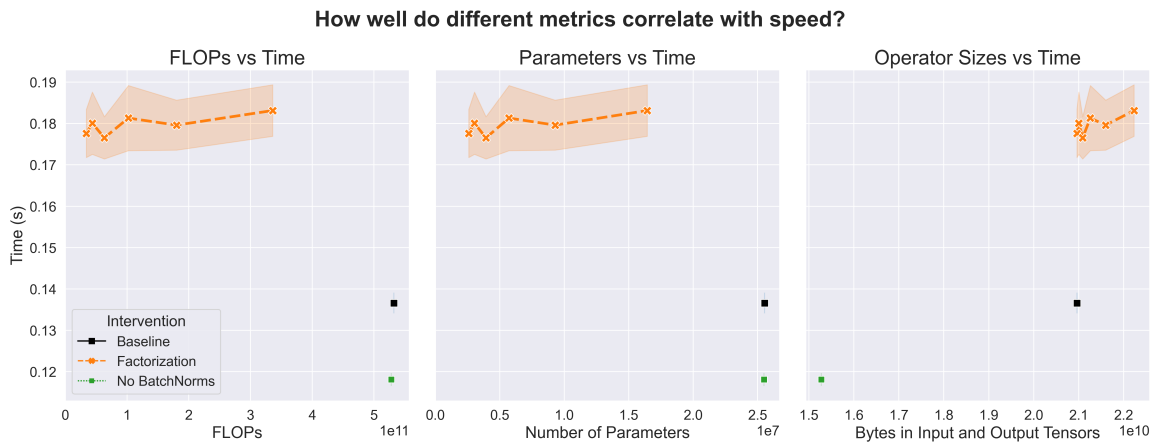

Figure 3: FLOP count *(left)* and parameter count *(middle)* are not reliable proxies for wall time when training ResNet-50. They are not even monotonically related to wall time. *(right)* The total size of all operator inputs and outputs is a somewhat better proxy, at least yielding a roughly monotonic relationship. None of these metrics are accurate enough predictors to substitute for measuring time directly.

This memory bandwidth bottleneck often arises on other hardware and in other networks as well. Illustrating this, we build on Figure 3 by considering two further variables: hardware and model architecture. Regarding hardware, we measure how various metrics correlate with runtime on a 2-socket server with AMD EPYC 7513 32-core processors to complement our A100 GPU results (see Figure 4). Regarding architecture, we collect both CPU and GPU measurements using ConvNeXt-B (Liu et al., 2022c) and Swin-B (Liu et al., 2021b) to complement our ResNet-50 results (CPU results for Swin and ConvNeXt are shown in Appendix C). The setup of Figures 5 and 6 is identical to that of Figure 3 with the exception that we omit removal of Batch Normalization as an intervention, since it is not applicable. Just as in Figure 3, these results show that a) FLOP count can be a poor proxy for wall time, and b) memory bandwidth consumption can be a better, though still imperfect, proxy.
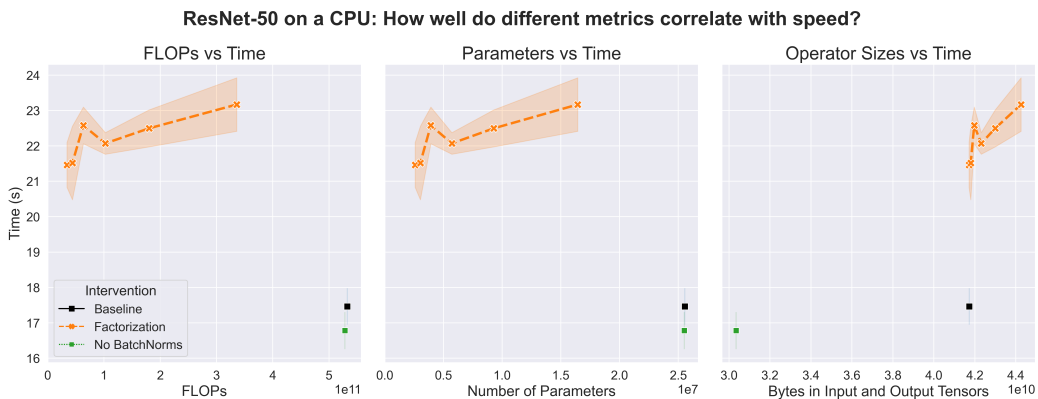

Figure 4: Runtime vs various metrics for ResNet-50 on 64 CPU cores.
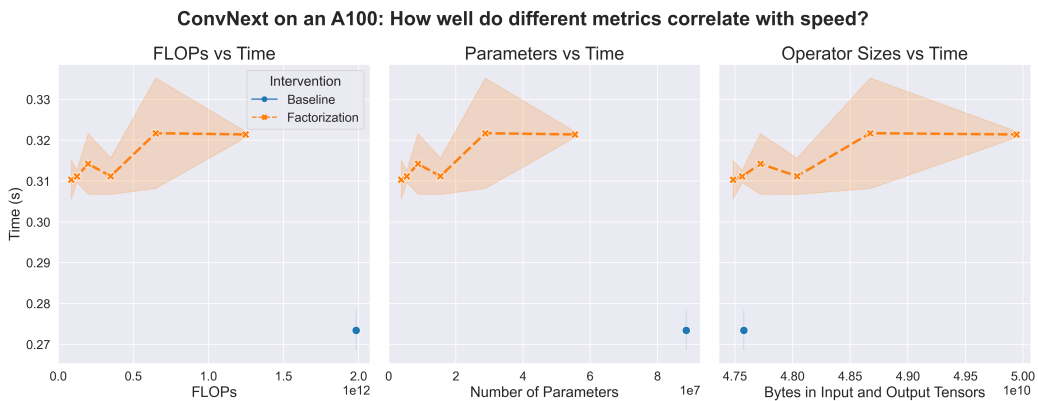

Figure 5: Runtime vs various metrics for ConvNeXt-B on an A100 GPU.

To summarize, all of the above efficiency metrics have limitations. Either these metrics are a step removed from costs of direct interest or they are dependent on confounding factors such as hardware, location, time, etc. However, by reporting multiple metrics—ideally via a set of accuracy-efficiency curves (Portes et al., 2022)—and holding as many confounding
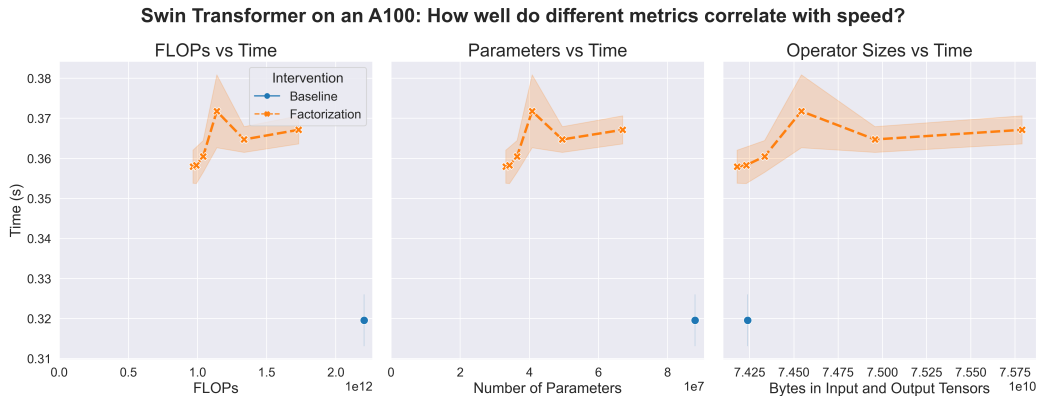
Figure 6: Runtime vs various metrics for Swin-B on an A100 GPU.

factors as possible constant, one can account for the fact that different metrics may lead to different or even opposite efficiency conclusions and adequately characterize efficiency (as we discuss further in Section 5).

Next, we define algorithmic speedup, adopting training time as our metric of interest. Our definition can be extended to perform training efficiency comparisons in terms of other metrics.

2.4 Algorithmic Speedup: Motivation and Definition

The goal of speedup research is to make training efficient by minimizing the time required to achieve the desired model quality. The most prevalent way to achieve a speedup is to improve software- or hardware-engineering aspects of the training pipeline; e.g., using optimized compute kernels or fusing together adjacent memory-bandwidth-bound operations. However, in this paper, we are interested in a complementary direction that we refer to as *Algorithmic Speedup*—modifying the semantics of the training process to achieve a speedup. Algorithmic Speedup is formalized as follows.

Definition 1 ((ϵ, δ)-Speedup) *Let Q denote the model quality achieved by the baseline training recipe $\mathcal{R}$ in time T. Algorithmic Speedup is an ϵ reduction in the time required to achieve model quality Q' by algorithmically changing the base recipe to $\mathcal{R}'$, where $\epsilon = \frac{T}{T'} > 1$ and $\delta = \frac{Q'-Q}{Q}$.*

Algorithmic Speedup aims to reduce the resources required to reach a particular model quality via one or both of the following effects: 1) a reduction in the number of iterations required to reach a specific performance level ($n_{\text{iteration}}$); and 2) a reduction in the time per iteration (T_i). If addition of a speedup method to a training recipe improves training time at the same or better model quality, i.e., ($\epsilon > 1, \delta \geq 0$), then the new recipe is strictly better than the baseline. In cases where model quality and training time conflict with each other (i.e., $\epsilon > 1, \delta < 0$), the goal is to find a Pareto optimal solution with a good tradeoff between both objectives. Ranking speedup methods under such scenarios is a challenging task. We discuss this in more detail in Section 5.
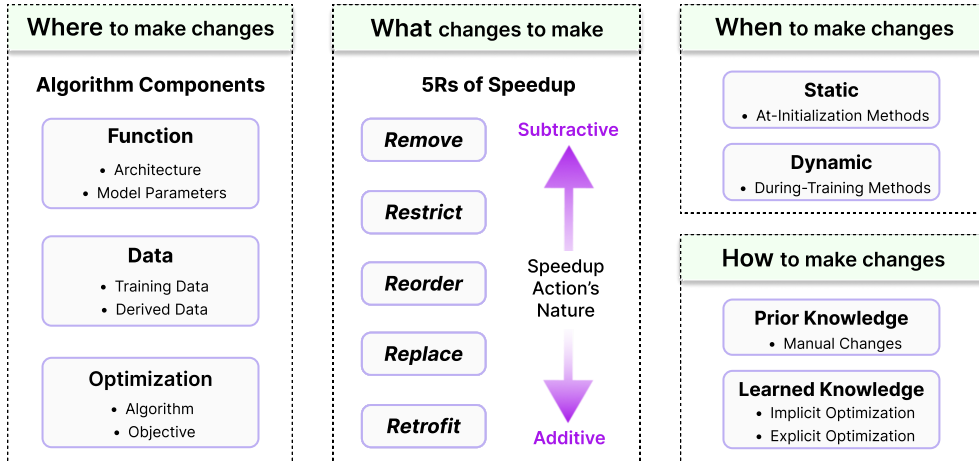
Figure 7: **A Taxonomy of Algorithmic Speedup Techniques.** Our taxonomy classifies algorithmic speedup approaches according to the four analysis questions above. We identify three fundamental building blocks (i.e., component, action, and mechanism) and present the subcomponents that the components comprise. Any speedup technique can be viewed as a path connecting different elements across these blocks. Note that multiple elements can be chosen from component, action, and mechanism blocks. New approaches can be derived by changing the elements and/or connecting a different path.

3. A Unifying Perspective for Speedup Methods: Taxonomy

In this section, we introduce a taxonomy that categorizes algorithmic speedup approaches based on three fundamental building blocks of speedup methods (Figure 7):

1. **Components:** Where to make changes?

2. **Actions:** What changes to make?

3. **Mechanisms:** When and how to make changes?

A detailed categorization of the literature according to this taxonomy with concrete examples will be presented in the next section (Section 4). Here we describe the taxonomy on a more conceptual level.

3.1 Component Categorization

The *component* building block indicates the aspect of the training pipeline to be modified (see Figure 1). Namely, the function, the data, or the optimization. Each component includes subcomponents, such as architecture and model parameters, which we list below.

3.1.1 Function Component

- **Architecture:** A neural network architecture defines the structure of the model and encodes inductive biases about the problem at hand. Two examples of neural network architectures are ResNet-50 (He et al., 2016) and DenseNet (Huang et al., 2017). An architecture is composed of many layers, which we regard as elements of the architecture subcomponent.

- **Model Parameters:** Model parameters are the free variables of a given architecture, which are trained to maximize its predictive performance.

3.1.2 Data Component

- **Training Data:** A training dataset is a collection of data samples used during the learning process to fit the model parameters. A data sample is usually a pairing of a raw data instance (e.g. image, text, or audio) and a corresponding annotation (e.g. a label or bounding boxes). Factors such as the number of samples and the size of each sample (e.g. resolution or sequence length) can significantly impact the training time.

- **Derived Data:** Derived data is data that is computed from training data during the learning process. Derived data appears during the forward and backward passes of the DNN. Some examples are activation values in the forward pass and gradients in the backward pass.

3.1.3 Optimization Component

- **Training Objective:** Neural networks are trained by optimizing an objective function computed using training data. An objective function is often divided into a loss function (e.g., cross-entropy or hinge loss) and a regularization penalty (e.g., weight norm or sharpness). It is usually desirable for objective functions to be easy to optimize, meaning that they are ideally smooth, as convex as possible, and well-conditioned.

- **Training Algorithm:** Training algorithms modify the model parameters to optimize the training objective. The training algorithm is composed of the initializer (e.g., Xavier or Kaiming initialization) and the optimizer (e.g., SGD or Adam). Initialization sets the model parameters to certain initial values that define the starting point for the optimizer. Optimizers are methods of changing the parameters to optimize the objective function, usually based on the gradient of the objective with respect to the parameters. Arguably more sensitive than other subcomponents to hyperparameter settings, the training algorithm can cause training to fail entirely if inappropriately configured.

3.2 Action Categorization

The *action* building block indicates the type of operation applied to a given component to achieve a speedup. Speedup methods tend to use one of the following five actions, which we call the ***5Rs of algorithmic speedup***.
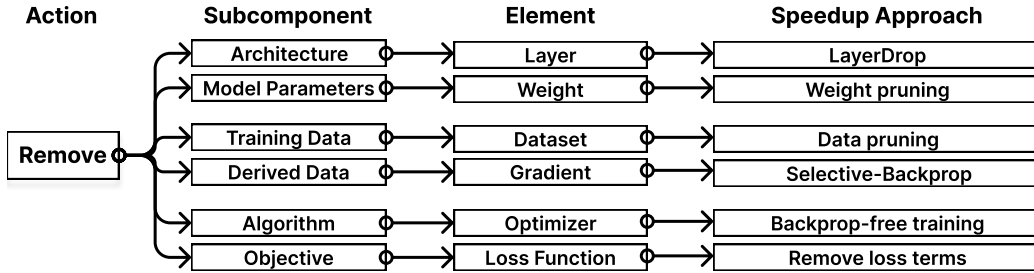
Action	Subcomponent	Element	Speedup Approach
	Architecture	Layer	LayerDrop
	Model Parameters	Weight	Weight pruning
	Training Data	Dataset	Data pruning
Remove	Derived Data	Gradient	Selective-Backprop
	Algorithm	Optimizer	Backprop-free training
	Objective	Loss Function	Remove loss terms

Figure 8: Illustration of example "remove" speedup techniques inside our taxonomy.

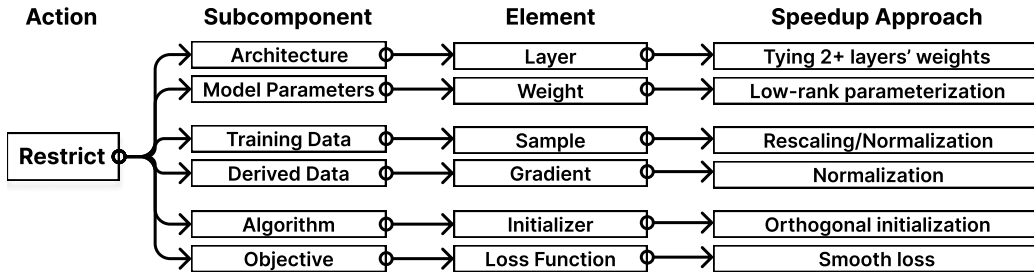| Action	Subcomponent	Element	Speedup Approach
	Architecture	Layer	Tying 2+ layers' weights
	Model Parameters	Weight	Low-rank parameterization
	Training Data	Sample	Rescaling/Normalization
Restrict	Derived Data	Gradient	Normalization
	Algorithm	Initializer	Orthogonal initialization
	Objective	Loss Function	Smooth loss

Figure 9: Illustration of example "restrict" speedup techniques inside our taxonomy.

3.2.1 Remove

This category of actions is subtractive, removing elements of the components they target. The remove operation typically aims to reduce the T_{total} needed to attain the desired accuracy by reducing T_i. Representative "remove" techniques for each subcomponent are illustrated in Figure 8.

3.2.2 Restrict

Components can often take on a range of possible values—e.g., all of $\mathcal{R}^D$ for a D-element parameter tensor. The restrict action shrinks the space of possible values in some way. Actions in this category typically aim to reduce the T_{total} needed to attain the desired accuracy by reducing T_i but may also reduce $n_{\text{iteration}}$. Representative "restrict" techniques for each subcomponent are illustrated in Figure 9.

3.2.3 Reorder

Without adding to or subtracting from its elements, training can be sped up by altering the places and phases in which its elements are introduced and used. Reorder techniques can reduce T_i through progressively increasing problem complexity. They can also reduce $n_{\text{iteration}}$ by, e.g., reordering samples to improve optimization. Representative "reorder" techniques for each subcomponent are illustrated in Figure 10.
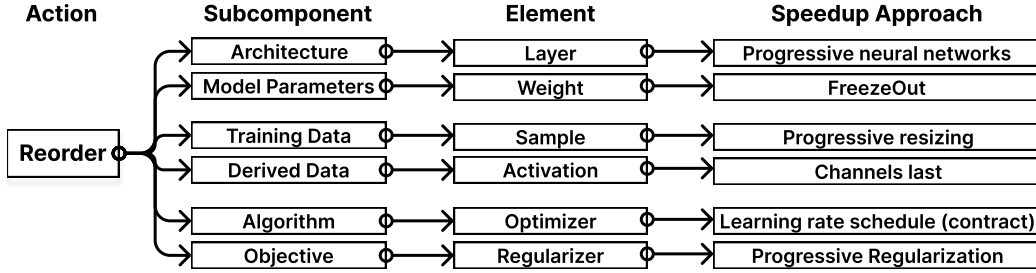
Action	Subcomponent	Element	Speedup Approach
	Architecture	Layer	Progressive neural networks
	Model Parameters	Weight	FreezeOut
	Training Data	Sample	Progressive resizing
Reorder	Derived Data	Activation	Channels last
	Algorithm	Optimizer	Learning rate schedule (contract)
	Objective	Regularizer	Progressive Regularization

Figure 10: Illustration of example "reorder" speedup techniques inside our taxonomy.

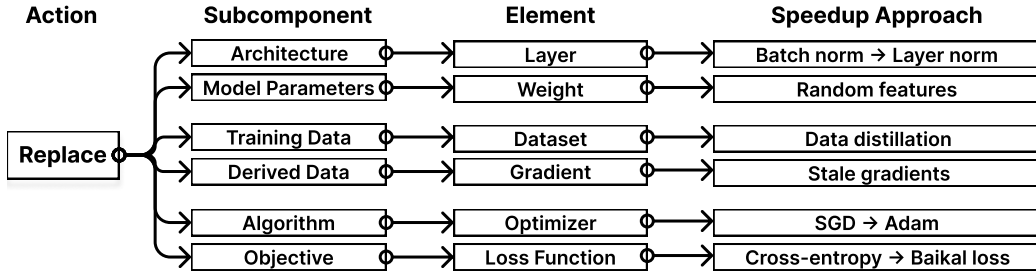| Action	Subcomponent	Element	Speedup Approach
	Architecture	Layer	Batch norm → Layer norm
	Model Parameters	Weight	Random features
	Training Data	Dataset	Data distillation
Replace	Derived Data	Gradient	Stale gradients
	Algorithm	Optimizer	SGD → Adam
	Objective	Loss Function	Cross-entropy → Baikal loss

Figure 11: Illustration of example "replace" speedup techniques inside our taxonomy.

3.2.4 Replace

Sometimes one can obtain a speedup by completely replacing one subcomponent/element with another. This can reduce both T_i and $n_{\text{iteration}}$. Representative "replace" techniques for each subcomponent are illustrated in Figure 11.

3.2.5 Retrofit

Retrofitting adds to components and is the opposite of the remove action. Because it typically does not reduce work done per iteration, retrofitting aims to reduce $n_{\text{iteration}}$ rather than T_i. Representative "retrofit" techniques for each subcomponent are illustrated in Figure 12.

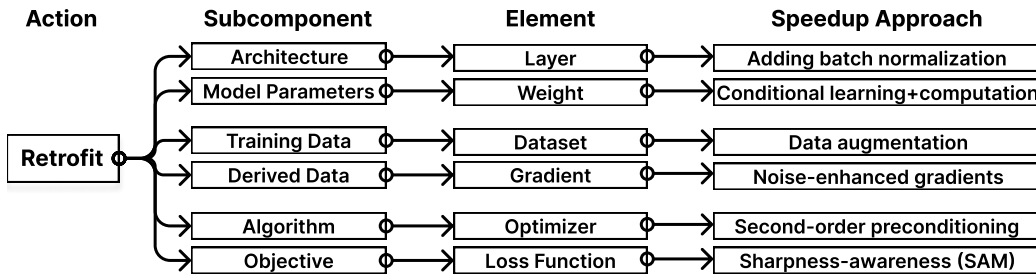| Action	Subcomponent	Element	Speedup Approach
	Architecture	Layer	Adding batch normalization
	Model Parameters	Weight	Conditional learning+computation
	Training Data	Dataset	Data augmentation
Retrofit	Derived Data	Gradient	Noise-enhanced gradients
	Algorithm	Optimizer	Second-order preconditioning
	Objective	Loss Function	Sharpness-awareness (SAM)

Figure 12: Illustration of example "retrofit" speedup techniques inside our taxonomy.

Subcomponent	Action	Timing of Action	Domain-Knowledge Based	Learning Based
Model parameters	Remove	Static	Sparsification at initialization using magnitude pruning	Sparsification at initialization using meta-learning
		Dynamic	Dynamic sparsification using gradual magnitude pruning	Dynamic sparsification using score-based optimization
Training data	Remove	Static	Coreset selection at initialization using expert knowledge	Coreset selection at initialization using meta-learning
		Dynamic	Minibatch selection using expert-guided sampling	Adaptive subset selection using bilevel optimization
Training algorithm	Replace	Static	Fixed learning rate found by experts (or theory)	Fixed learning rate predicted using regression or range test
		Dynamic	Learning rate schedule found by experts (or theory)	Adaptive learning rates predicted using regression

Table 2: **Illustrative Examples of Speedup Mechanisms.** These examples show how and when speedup actions can be applied.

3.3 Mechanism Categorization

The *mechanism* building block describes when and how to perform an action on a component. For "when," we consider time in relation to the training process. For "how," we focus on what information determines the details of the change.

3.3.1 When to Make the Change?

- **Static Methods:** These methods perform component-action changes only once at initialization, i.e., before the training starts.

- **Dynamic Methods:** These methods perform component-action changes during training, e.g., at various training iterations.

3.3.2 Change Based on What Information?

Component-action changes can be made based on various sources of knowledge. We categorize speedup methods based on:

- **Domain-Knowledge-Based Methods:** These methods perform component-action changes using domain knowledge, i.e., prior experience or expertise related to the training process.

- **Learning-Based Methods:** These methods perform component-action changes by either explicitly or implicitly optimizing for faster training time via leveraging the data corresponding to current (or past) training runs.

Some examples of mechanisms for popular subcomponent-action combinations are provided in Table 2. Finally, we note that some speedup methods can be associated with more than one of the paths through our taxonomy (e.g., a method might be reasonably classified as both a **replace → derived data** and a **replace → training algorithm** method). This stems from the fact that our taxonomy was not designed to be a mutually exclusive and

collectively exhaustive categorization of the literature. Instead, our taxonomy aims to group speedup methods to aid reasoning about their potential effectiveness given a particular bottleneck, as well as reveal themes and opportunities among the broad set of approaches we review.

4. Categorization of Existing Work: Survey

Here, we discuss the representative literature on algorithmic speedup methods.

4.1 Function Speedup Strategies

Function speedup strategies apply the 5Rs to the model parameters and architecture. Function speedup strategies that reduce function latency like **remove → model parameters** (e.g., pruning) can reduce T_i when the function is a bottleneck. Alternatively, more additive speedup strategies like **retrofit → architecture** (e.g., widening layers) tend to help reduce $n_{\text{iteration}}$. We place various function speedup options into our taxonomy and review them below.

4.1.1 MODEL PARAMETERS

Remove   **Remove → model parameters** is also known as model pruning (Hoefler et al., 2021) and has been approached in many ways. We focus our discussion on several examples that explicitly demonstrate training speedups despite the fact that sparse matrix computations do not offer significant benefits on most AI accelerators.[1] In particular, we discuss approaches that attain speedups by pruning structures larger than individual weights (Lym et al., 2019; Chen et al., 2020; Yuan et al., 2021). Structured pruning entails removing a block of parameters, rather than individual parameters. E.g., one might remove an entire filter from a convolutional layer. For some block structures, this can produce speedups without specialized hardware (Yuan et al., 2021).

Pruning-based acceleration should also facilitate accuracy similar to the unpruned baseline model's, which is difficult to reach when training from scratch with a fixed and naively-applied sparsity pattern (Liu et al., 2018c; Frankle et al., 2020a). Lym et al. (2019) address this with their method PruneTrain, which applies group lasso regularization to filters/channels in convolutional networks to encourage the emergence of a sparsity pattern during training. PruneTrain periodically computes the maximum absolute value of the parameters in each filter, then removes the filters with the smallest values to reduce T_i. Lym et al. (2019) shows this approach produces speedups without a significant loss of accuracy. The intuition behind the success of this approach is that once weights are sparsified by group lasso, they rarely grow above the threshold later in training. Thus, these weights can be pruned without degrading accuracy.

A similar regularization-based approach is taken by EarlyBERT (Chen et al., 2020) to accelerate pretraining and fine-tuning of large language models. Specifically, Chen et al. (2020) encourage sparsity by applying ℓ_1 regularization to coefficients on outputs associated with groups of parameters in a manner inspired by Liu et al. (2017). A key difference is that the parameter groups in language models are neurons in fully connected layers and

---

1. Emerging hardware like Hall et al. (2021) may accelerate unstructured sparsity in the future.

attention heads rather than convolutional filters. Chen et al. (2020) also take inspiration from the Early-Bird ticket algorithm (You et al., 2019), which was shown to reduce the energy cost of training by pruning early in the training process without sacrificing accuracy. You et al. (2019) use the scaling factors in batch normalization layers as indicators of the corresponding channels' significances. Early-Bird tickets are identified "early" when the Hamming distance between the pruning masks on successive iterations becomes sufficiently small (You et al., 2019). EarlyBERT (Chen et al., 2020) uses Hamming distance during training to justify its early selection of a sparsity pattern as well, and it enforces sparsity before 10% of the total training iterations are reached. They find that this not only reduces T_i but also $n_{\text{iteration}}$, thanks to an increased accuracy early in training.

Yuan et al. (2021) address the issue of picking a suboptimal initial sparsity mask by using a dynamic sparsity mask throughout training in their MEST pruning framework. During training, MEST periodically runs a prune-and-regrow iteration that removes parameters with both small weight and gradient magnitudes, then reactivates previously removed parameters such that the model is always trained with a constant sparsity level. MEST's dynamic approach facilitates a low memory footprint via sparse gradient and parameter vectors, while not requiring the permanent removal of potentially important connections. Notably, since pruning larger structures is associated with worse accuracy, Yuan et al. (2021) also explore pruning structures of various sizes to find a good balance between accuracy and speedup.

Adelman et al. (2021) use sample-based approximations to tensor operations—i.e. matrix multiplications and convolutions. This involves dynamic row/column pruning of the tensors in these operations. On ImageNet, models trained with this sampling can train $1.37\times$ faster with little impact on the final test accuracy.

Learning-based approaches dominate **remove → model parameters** due to the difficulty of reaching good accuracy with a predefined or random sparsity pattern. The development of domain-knowledge approaches could remove the computational overhead of discovering a good sparsity pattern through experimentation. This may involve deepening our understanding of initializing, regularizing, and/or selecting the connectivity of sparse networks.

Restrict   There are multiple ways to **restrict → model parameters** for speedup, including both weight quantization and low-rank factorization methods. Similar to parameter removal, parameter restriction is a straightforward way to reduce theoretical FLOPs. However, realizing these gains while maintaining accuracy is challenging.

Restricting a weight matrix to be less than full rank by factorizing it into two smaller matrices can reduce T_i when computations associated with the weight matrix can be performed faster with its two small factors. This often requires the weight matrix to be relatively large. Sainath et al. (2013) show that applying this low-rank factorization to the large, densely-connected output layer in acoustic and language models can produce speedups without significantly harming accuracy. While low-rank approaches often focus on such fully connected layers, Ioannou et al. (2015) show that factorizing convolutional filters in VGG-11 and similar networks can also produce speedups without significantly harming accuracy. Ioannou et al. (2015) develop a variety of factorization schemes, but a core idea is to ap-

proximate $w \times h$ filters as a layer of $w \times 1$ and $h \times 1$ basis filters followed by a layer of 1×1 filters that linearly combines the outputs of the basis-filter layer.

The Pixelated Butterfly (Chen et al., 2021a) (or Pixelfly) approach can translate FLOPs reductions from restrictions into speedups when models are dominated by sufficiently large General Matrix Multiply (GEMM) operations. Pixelated Butterfly applies the following reparameterization of the weight matrix W before training begins: $W = \gamma B + (1 - \gamma)UV^T$, where B is a block sparse variant of a butterfly matrix, UV^T is a low-rank component, and γ is learnable. Chen et al. (2021a) prove that this combination of sparse and low-rank matrices is more expressive than using just a sparse or just a low-rank matrix. Moreover, Chen et al. (2021a) find that applying Pixelated Butterfly to various Transformer and MLP-mixer models leads to high accuracies along with speedups. Similarly, Dao et al. (2022a) propose Monarch matrices that are hardware-efficient (i.e., parameterized as products of two block-diagonal matrices for better hardware utilization) and expressive (i.e., can represent many commonly used transforms).

Indeed, the sizes of large transformer models have motivated a variety of training efficiency approaches. Hu et al. (2021) provide another example of restricting transformer parameters via their LoRA algorithm, which facilitates fast and high-quality fine-tuning of pretrained language models by learning low-rank update matrices that are added to the pretrained weight matrices (avoiding fine-tuning of the full weight matrices). Note that we discuss even more efficient transformer approaches, including low-rank-like approaches related to efficient transformer *attention*, in Section 4.2.2.

Quantization is another way to restrict parameters and is commonly used to reduce inference cost—see (Gholami et al., 2021) for a survey of such techniques. However, Micikevicius et al. (2017) show that using a mix of half and single-precision weights and/or activations at training time ("mixed precision training") can produce large speedups, particularly on newer AI accelerators that specifically support fast half-precision computation. Notably, Micikevicius et al. (2017) avoid accuracy reductions associated with training with lower precision weights in part by avoiding zeroing of parameter gradient information, which is accomplished via 1) scaling up the loss so that small-magnitude gradients are pushed into a range that is captured by half-precision weights ("loss scaling"); and 2) representing the half-precision gradient in single precision before multiplying it by the step size. Many modern deep learning frameworks provide automatic mixed precision (AMP) training functionality that addresses these subtleties for users.

In terms of mechanisms, **restrict → model parameters** approaches for fast training are mostly static—e.g., the rank is chosen prior to training. Interestingly, there exist more dynamic approaches that can find an appropriate rank for each layer (Idelbayev and Carreira-Perpiñán, 2020), but our review did not find an example of such a method that also provides a speedup (with Liebenwein et al., 2021, being a potential exception for pipelines involving retraining). This is due to the added cost of dynamically finding the appropriate ranks of the parameter matrices. Reducing this cost could lead to low-rank methods that adapt to training dynamics, which might improve their efficacy.

Reorder Speedups that **reorder → model parameters** mostly exploit the observation that parameters in some layers train faster than those in other layers (Raghu et al., 2017). This creates an opportunity to gradually freeze parameters, rather than train all parameters

all the time. When a parameter is "frozen," it no longer requires gradient or parameter update computations.

Brock et al. (2017) propose FreezeOut to reduce the training time by only training each layer for a set portion of the training schedule, progressively "freezing out" layers and excluding them from the backward pass. Specifically, cosine annealing without restarts is used in a layer-wise manner such that the set of trained model parameters shrinks each time a layer's learning rate is annealed to zero. Similarly, Raghu et al. (2017) introduce "Freeze Training" to successively freeze lower layers during training.

In terms of mechanisms, using progressive training to **reorder → model parameters** is dynamic by definition. Moreover, most of these approaches use knowledge-based mechanisms. An exception is PipeTransformer, proposed by He et al. (2021), which uses a learning-based mechanism. PipeTransformers use a gradient-norm-based objective to identify and freeze layers gradually during training. The resulting scheme is lightweight and achieves up to a 2.83x speedup without losing accuracy. We accordingly suspect that there is great potential for other learning-based approaches in this area.

Replace **Replace → model parameters** is made difficult by the need to match the replacement parameters to the learning task at hand. This matching process is illustrated by Sarwar et al. (2017), who note that Gabor filters are a powerful processing tool in computer vision and replace a fraction of convolutional weight kernels with fixed Gabor filters. These fixed filters do not need to be trained, which facilitates improved training speed. High accuracy is reached due to the filters' suitability for computer vision applications (Sarwar et al., 2017).

Similarly, Shen et al. (2020) build on the observation that transformers can perform well with random attention by introducing "Reservoir Transformers". Specifically, Shen et al. (2020) replace a subset of a transformer's layers with random non-linear transformations. They show that replacing trainable layers with these random reservoir layers improves both speed and generalization.

From a mechanism perspective, it's clear that the approaches we reviewed are more static in that **replace → model parameters** happens only once, prior to training's start. Additionally, the reviewed approaches rely on domain knowledge regarding the suitability of the replacement for the new task. A potential opportunity is applying replacement operations more dynamically based on learning. For example, a technique might find speedups by choosing the most appropriate replacement for a task according to the performance of a variety of candidates on a subset of that task.

Retrofit Adding model parameters to lower $n_{\text{iteration}}$ via increased model capacity is typically achieved via architecture changes, but there are creative ways to **retrofit → model parameters** without requiring significant architectural changes. For example, Fedus et al. (2021) retrofit a transformer's baseline parameter set with additional parameters to create Switch Transformers, which have up to a trillion parameters but the ability to train seven times faster than a T5 transformer. By using subsets of its larger parameter set conditionally, Switch Transformers are able to have the same basic architecture and computations as a baseline transformer model despite having many more parameters. Notably, Switch Transformers employ a switching/routing mechanism to select the specific subset of parameters to use based on the input data (Fedus et al., 2021). The parameter-enhanced model

is more sample efficient, which means it has lower training times due to the lower $n_{\text{iteration}}$ required to reach a given model quality.

Alternatively, Izmailov et al. (2018) add a new set of parameters that are the average of multiple points along the SGD-trained-parameter trajectory in their Stochastic Weight Averaging (SWA) method. They show SWA leads to better model quality per parameter update and has little overhead, making speedups possible through reduced $n_{\text{iteration}}$. Yang et al. (2019) extend SWA to low-precision training. Relatedly, Kaddour (2022) propose recording the state of the model parameters at the end of each training epoch in their LAtest Weight Averaging (LAWA) algorithm. At the start of each epoch, the model parameters are defined as the average of the last k prior parameter states, and Kaddour (2022) show this leads to significant speedups. For parallel training, Li et al. (2022b) introduce the Branch-Train-Merge (BTM) approach, which averages copies of a large language model (each trained on its own intelligently-selected data subset), facilitating faster distributed training and better accuracy.

Using a load-balancing loss term to ensure the conditionally used parameters are well utilized and trained, Fedus et al. (2021) illustrate the importance of learning-based mechanisms to **retrofit → model parameters**. Fedus et al. (2021) also provides (for the first time, to the best of our knowledge) an empirical study of parameter count scaling that doesn't affect FLOPs. This emerging domain knowledge and knowledge regarding benefits of weight averaging (Kaddour et al., 2022) may each be useful for developing new **retrofit → model parameters** approaches.

4.1.2 ARCHITECTURE

Remove **Remove → architecture** often immediately reduces T_i through the removal of entire layers. However, work in this area must take steps to maintain performance and/or adapt other aspects of the network to account for the removal. For instance, batch normalization layers have a large computational cost but are important to model performance, providing both regularizing and stabilizing effects. Accordingly, to remove the batch normalization layers, Brock et al. (2021) take several steps to regularize and stabilize the training in other ways. They regularize by adding Dropout (Srivastava et al., 2014), Stochastic Depth (Huang et al., 2016), and a variety of data augmentations. To improve stability, they scale down residual branch outputs, standardize the weights, and (critically for the large batch size and learning rate regime) adaptively clip the gradients based on the scale of the parameters (Brock et al., 2021).

Huang et al. (2016); Fan et al. (2019) show that networks can be resilient to the removal of entire blocks of computation (sets of layers) when the removal is applied stochastically throughout training. Specifically, these approaches effectively apply Dropout (Srivastava et al., 2014), but the structures that are dropped are ResNet blocks (Huang et al., 2016) or transformer layers (Fan et al., 2019) rather than neurons. Huang et al. (2016) show that good performance is attained by their Stochastic Depth method when the drop probability for earlier ResNet blocks is lower than that of later layers, based on the intuition that the extraction of low-level features should reliably be performed on each forward pass. Alternatively, Fan et al. (2019) show good performance of their LayerDrop method when using a constant drop probability for all transformer layers. Each approach reduces T_i and

can actually reduce the $n_{\text{iteration}}$ required to attain a desired performance level through the regularization effect that random layer dropping provides (Huang et al., 2016; Fan et al., 2019).

Our review of popular **remove $\rightarrow$ architecture** methods suggests that such approaches have a strong dependence on domain knowledge, requiring an understanding of how to re-place the effects of removed layers (for example), and tend to be dynamic. Interestingly, while the removal of batch normalization is a static action that NFNets use to attain speedups, Brock et al. (2021) compensated for the loss of batch normalization by dynami-cally altering gradients/weights and adding regularization throughout training.

Restrict Our review found **restrict $\rightarrow$ architecture** for speedups approached in only one way, which may reflect the difficulty of developing ways to restrict entire layers of architectures.

Press and Wolf (2016) do this by tying the input embedding and output embedding matrices of language models. They show that restricting embedding layers in this way results in better perplexity per training iteration, which means this technique lowers $n_{\text{iteration}}$. Weight tying is also used in modern Transformer-based language models, where it can save even more embedding parameters thanks to the presence of both encoder and decoder embedding matrices (Vaswani et al., 2017).

Reorder While it may seem challenging to find inefficiencies in the ordering of popularly used architectures, our survey revealed multiple ways to achieve speedups with **reorder $\rightarrow$ architecture**. For example, Saharia et al. (2022) design a faster U-Net (Efficient U-Net) for the Imagen text-to-image model by applying U-Net's downsampling (upsampling) *before* (*after*) the convolutional layers in the downsampling (upsampling) blocks. This approach causes the convolutional layers to operate on smaller feature maps than they would if the original order were used, which improves training speed via reduced T_i without quality degradation.

Separately, many progressive learning approaches reorder the layers of smaller, quickly-trained (or pretrained) networks to initialize a large network that can be trained faster than its randomly initialized counterpart. For example, Chen et al. (2015a) proposed Net2Net, which accelerates the training of a large network by building it from the pretrained layers of a network with less depth/width. The strategy for growing the network's depth/width ensures that the grown network's output matches the smaller network's output when train-ing starts. This enables a reduced $n_{\text{iteration}}$ relative to using a randomly-initialized large model. Similarly, BERT pretraining can be accelerated via training progressively deeper BERT models created by the stacking of shallower BERT models' layers (Gong et al., 2019) or growing in multiple network dimensions (Gu et al., 2020).

Notably, these methods build architectures using regular reorderings of fundamental components, such as by duplicating each layer to increase the depth by a factor of two. This regularity might be suboptimal on a broader set of learning tasks, which motivates the automated progressive learning scheme AutoProg (Li et al., 2022a). During training, AutoProg repeats the following three steps to gradually build a ViT with no accuracy drop relative to the large ViT baseline: 1) train the best-performing ViT subnetwork found so far; 2) scale the trained subnetwork into an Elastic Supernet, and 3) train a variety of subnetworks sampled from this Elastic Supernet. Li et al. (2022a) connect this approach to

their "Growing Ticket Hypothesis of ViTs", which states that the performance of a large ViT model can be reached in the same number of training iterations by instead progressively growing a ViT from one subnetwork. Due to the lower T_i associated with training subnetworks instead of the large ViT and the constant $n_{\text{iteration}}$, this approach can speed up ViT ImageNet training by 40-85%.

A related but slightly different direction is training each layer or block of layers individually rather than at the same time, which frees up GPU memory. Huang et al. (2018) provide some evidence that this can improve training time not only through a reduced number of required steps to reach a given performance level but also through faster step time. However, the authors also show this approach can sometimes lead to worse model quality relative to end-to-end optimization.

Mechanistically, the **reorder → architecture** approaches we found are mostly domain knowledge, dynamic approaches. However, there is some evidence (Li et al., 2022a) that using learning rather than domain knowledge could improve performance.

Replace **Replace → architecture** is one of the most common architecture-based speedup approaches in the literature we reviewed.

One theme of **replace → architecture** work is frequency-based reformulations of layers. Computational complexity analysis suggests that replacing dense layers with Fourier-transform-based versions can produce speedups (Yang et al., 2015; Moczulski et al., 2015), but Yang et al. (2015) note that practical realization of speedups would require a large fraction of layers to be dense layers. This condition is met in modern transformer networks, and Lee-Thorp et al. (2021) show that their FNets, which replace transformer attention layers with a 2D discrete Fourier transform (DFT), have lower training times than BERT without heavy accuracy loss. At small model sizes, FNets are on the speed-accuracy Pareto frontier, while at larger model sizes BERTs and FNet-transformer hybrids (using self-attention only in the final two layers) are preferable (Lee-Thorp et al., 2021). Similarly, frequency-related reformulations of convolutional layers have also led to demonstrable speedups (Chen et al., 2019; Dziedzic et al., 2019). Note that FNets are closely related to transformer variants that aim to reduce the quadratic (in sequence length) complexity of attention through approximations of attention.[2] However, FNets are notable in that they do not seek to approximate attention and instead simply mix tokens in the sequence and hidden spaces through a 2D DFT. Other efficient transformer variants use attention but avoid the complexity associated with computing its full set of activations and are thus discussed in the relevant sections of 4.2.2.

Another **replace → architecture** approach is using training-aware neural architecture search (NAS). Tan and Le (2021) developed EfficientNetV2, a convolutional neural network optimized for faster training rather than lower FLOPs. Their architecture search shows that the replacement of MBConv layers with fused MBConv layers in early architecture stages offers faster training. Further, a smaller expansion ratio for the MBConv layers and smaller kernel sizes with more layers were also shown to help. Interestingly, Tan and Le (2021) train with various image sizes to attain additional speedups. They also obtain greater

---

2. We recommend Tay et al. (2020c) for a survey on efficient transformers and Narang et al. (2021) for an analysis of various transformer replacement approaches.

accuracy by reducing model regularization strength when the image sizes are smaller, which we discuss further in Section 4.3.1.

Timmons and Rice (2020) used function approximation techniques to develop faster replacements for the sigmoid and hyperbolic tangent activation functions. They show 10-50% reductions in training time without affecting model quality on CPUs, noting that future work may build on their techniques to speed up training on accelerators.

Our review found many approaches to **replace → architecture** that use static, domain knowledge mechanisms. EfficientNetV2 (Tan and Le, 2021), however, uses a learning-based mechanism in its explicit optimization of training speed with neural architecture search (NAS). Importantly, NAS has emerged as a popular technique in the efficiency literature (Ren et al., 2021b)—it can be tailored to discover fast architectures for specific compute platforms (Liberis et al., 2021) and training-accelerating enhancements for known architectures (So et al., 2021). Notably, since it typically involves training, NAS itself can be accelerated through efficient-training techniques like data pruning (Prasad et al., 2022), which we discuss more in Section 4.2.1. See White et al. (2023) for a recent survey of the NAS literature.

Retrofit **Retrofit → architecture** aims to reduce the $n_{\text{iteration}}$ required to reach the desired accuracy at the cost of increased T_i.

For example, So et al. (2021) show large speedups by retrofitting various transformer backbones with operations discovered by NAS. In particular, they introduce a depth-wise convolution after each query (Q), key (K), and value (V) projection. They also replace the softmax with a squared ReLU activation. Importantly, these speedups are shown to be present on various AI accelerators (GPUs and TPUs) and in various models. This retrofit approach increases T_i but reduces the $n_{\text{iteration}}$ needed to reach a given accuracy.

Another popular approach that involves a similar tradeoff is increasing layer width or number of layers (depth), which can lead to a higher accuracy gain per training step. Li et al. (2020) use this strategy and show that increasing the depth and width of RoBERTa models can produce large speedups.

Our review suggests that **retrofit → architecture** approaches tend to be static in nature; i.e., changes are made only once at initialization. However, we found changes may either be based on domain knowledge or learned via evolutionary search over various layer types (So et al., 2021).

4.2 Data Speedup Strategies

Data-related speedup methods apply the 5Rs to the training data and derived data subcomponents. Approaches in the **remove → derived data** category (e.g., Selective-Backprop) reduce the burden of gradient and activation computations and thus can reduce T_i. On the other hand, if data loading is the bottleneck, **remove → training data** approaches can speed up training through methods that include reducing the sizes of data samples. Regardless of bottleneck location, speedups can be achieved by reducing the $n_{\text{iteration}}$ required to reach the desired accuracy through methods like **reorder → training data** (e.g., curriculum learning). We place various data-related speedup strategies into our taxonomy and review them below.

4.2.1 TRAINING DATA

Remove **Remove → training data** can occur at both the dataset and data sample level. On a sample level, Cutout (DeVries and Taylor, 2017) drops square regions of pixels, and ColOut (Stephenson, 2021) randomly drops rows and columns of an input image for a vision model. If the dropout rate is not too large, the image content is not significantly affected, but the image size is reduced. Both of these methods act as a form of regularization, but only ColOut reduces computation too.

On a dataset level, several data-pruning methods are motivated by the observation that there are significant redundancies in popular benchmark datasets (Birodkar et al., 2019; Paul et al., 2021). The challenge for data pruning research is to identify samples to be excluded from training without compromising model quality. Mirzasoleiman et al. (2020) propose Coresets for Accelerating Incremental Gradient descent (CRAIG). CRAIG formulates data pruning as a weighted subset selection problem that aims to approximate the full gradient using selected data. To efficiently solve this NP-hard problem, they transform the original problem into a submodular set cover problem that can be solved using a greedy algorithm. Killamsetty et al. (2021a) propose GradMatch, which is similar to CRAIG but tries to directly minimize the gradient difference between the subset and the entire training set. The authors show that the objective function is approximately submodular and use orthogonal matching pursuit (OMP) to solve the gradient matching problem. Killamsetty et al. (2021b) introduce GLISTER ("GeneraLIzation based data Subset selecTion for Efficient and Robust learning"). GLISTER solves a mixed discrete continuous bi-level optimization problem to select a subset of the training data by maximizing the log-likelihood on a held-out validation set. The authors use a Taylor-series approximation to make GLISTER efficient. All of these approaches make use of the connection between submodularity and data selection that was explored in earlier work on dataset reduction for speech recognition (Wei et al., 2013, 2014), machine translation (Kirchhoff and Bilmes, 2014), and computer vision (Wei et al., 2015). Further, they have been built upon to accelerate domain adaptation training (Karanam et al., 2022), semi-supervised learning (Killamsetty et al., 2021c), and hyperparameter tuning (Killamsetty et al., 2022).

Relatedly, Raju et al. (2021) speed up training by determining a new subset of data to train on multiple times throughout training via simple heuristics for calculating sample importance. They attribute their data pruning approach's ability to maintain model quality at aggressive pruning rates to its dynamism and the existence of points that are important to the learned decision boundary for only some of the training time. To better exploit these points, they propose two reinforcement-learning-based sample importance scores, use them for dynamic data pruning, and achieve slightly better accuracy than a random (but faster) baseline.

The aforementioned approaches perform data selection every $L \geq 1$ epochs of stochastic gradient descent, and the gradient descent updates are performed on the subsets obtained by the data selection. However, a number of works have also studied the problem of pruning a dataset prior to (or very early in) training. For example, Toneva et al. (2018) show that a large fraction of data can be removed prior to training without accuracy loss by removing the examples associated with the lowest "forgetting event" counts. A forgetting event is counted when a sample transitions from being classified correctly to misclassified during

training, as computed by training the model on the full dataset. Due to the need to train the model on the full dataset before training it on the forgetting-event-based subset, they experiment with computing forgetting events on a faster-to-train model; they show that this leads to effective subsets for the original model. Consistent with this, Coleman et al. (2019) design the "selection via proxy (SVP)" approach that performs subset selection based on properties computed with a faster-to-train proxy model. These proxy models are obtained by removing hidden layers from the target deep model, using smaller architectures, and training for fewer epochs. Despite having lower accuracy, these models can be used to compute characteristics like forgetting events that can be used to select data subsets that are effective for training a high-quality target model more quickly. SVP was able to prune 50% of CIFAR-10 without harming the final accuracy. Chitta et al. (2021) show that data pruning can be performed by leveraging ensemble active learning. In particular, they build implicit ensembles from hundreds of training checkpoints across different experimental runs, then build a data subset using prediction-uncertainty-based data acquisition functions that are parameterized by the ensembles. Sorscher et al. (2022) evaluate many existing data pruning metrics, finding that 1) they perform poorly on ImageNet, even if they performed well on smaller datasets like CIFAR-10; 2) the best metrics are computationally intensive; and 3) all depend on labels (which precludes data pruning for unlabeled dataset pretraining). They address these disadvantages by ranking example difficulty without label access via k-means clustering in the embedding space of an ImageNet model trained with self-supervision. Examples with embeddings far from the nearest cluster centroid are considered the most difficult and kept in the dataset. This approach allows training on ImageNet with 20% fewer data samples, correspondingly lower $n_{\mathrm{iteration}}$, and little accuracy loss. Interestingly, Sorscher et al. (2022) also provide evidence for the idea that the hardest (easiest) examples should be kept when the initial training set is relatively large (small).

As our discussion reflects, the **remove → training data** approaches we found span both static and dynamic mechanisms. While learning-based mechanisms typically produce the best results, cheaper-to-compute domain-knowledge-based mechanisms like random pruning can be effective, particularly when applied dynamically. Significant progress has been made on pruning smaller datasets like CIFAR-10 and CIFAR-100, but we are still far from achieving large amounts of data pruning on ImageNet-scale data.

Restrict   **Restrict → training data** can be done at both the sample and dataset levels. These techniques tend to achieve speedup by reducing $n_{iteration}$ while keeping T_i fixed. The major idea behind this class of speedup techniques is to restrict the statistical properties of the data.

On a data sample level, popular normalization techniques include centering, scaling, decorrelating, standardizing, and whitening. A more detailed review of these techniques is provided in Huang et al. (2020).

On a dataset level, there are several theoretical works that show that significant speedups can be achieved by imposing certain restrictions on the training data generation process. For example, Kailkhura et al. (2020) show that training datasets with optimized spectral properties, e.g., space-filling designs, produce models with superior generalization. Jin et al. (2020) introduce cover complexity (CC) to measure the difficulty of learning a dataset as a function of the richness of the whole training set and the degree of separation between

different labeled subsets. They found that the error increases linearly with the cover complexity both in theory and on MNIST and CIFAR-10. The major obstacle preventing the use of these theoretical findings is that one rarely has control over the data-generating distribution.

Most of the above **restrict → training data** approaches are static and based on domain knowledge.

Reorder **Reorder → training data** changes the order in which examples or aspects of examples are presented to the model during training. The central challenge of this set of approaches is outperforming the random default ordering typically used in training. We discuss ordered learning via curriculum and progression approaches and suggest Wang et al. (2021) for further discussion of ordered learning.

Reordering the dataset using a curriculum can aid learning (Bengio et al., 2009), allowing a larger performance gain per iteration that reduces the $n_{\text{iteration}}$ required to reach a desired performance level. Hacohen and Weinshall (2019) analyzed the effect of key curriculum learning method components: 1) the scoring function used to assign training examples a difficulty level; 2) the pacing function used to determine when new examples are presented to the model; and 3) the ordering that determines whether examples are presented easiest first, hardest first, or randomly. Scoring functions typically reflect example complexity, uniqueness, signal-to-noise ratio, etc., while pacing functions can be linear, step-based, exponential, etc. Interestingly, Hacohen and Weinshall (2019) found that curriculum learning offers a relatively small but statistically significant training speed benefit. Similarly, Wu et al. (2020) systematically study the effect of different scoring, pacing, and ordering choices. On CIFAR-10, CIFAR-100, and FOOD-101(N), they found that scoring and ordering had no effect on model quality at convergence, while the dynamic dataset size induced by the pacing function offers a small benefit. However, Wu et al. (2020) found that speedup benefits (through fewer training steps required to reach a particular accuracy) were notable when there was a limited training iteration budget or when labels were noisy; in both cases, an easy-then-hard ordering was helpful, while other orderings were not.

Ordered learning of data can also be done in a progressive manner, with an important example being progressive resizing (Howard, 2018; Touvron et al., 2019; Hoffer et al., 2019b; Tan and Le, 2021). This method works by initially training on images that have been downsampled to a smaller size, then slowly growing the images to a larger size. Training is accelerated because computation in many models is proportional to image resolution, so T_i is reduced for most of the training. Accuracy is maintained (or even improved) with these approaches due to the model's exposure to various image sizes during training. Interestingly, Tan and Le (2021) find that the performance of this approach is further improved by increasing the regularization strength for higher image resolutions.

Our review found that **reorder → training data** is mostly approached with domain knowledge mechanisms. An exception is "teaching with commentaries" (Raghu et al., 2020), which uses a learning-based mechanism to find meta-information (or commentaries) that will produce speedups when used during training. Specifically, Raghu et al. (2020) show that learned per-iteration example weights can facilitate learning speed improvements. More work along these lines may help the success of **reorder → training data** approaches, as

the marginal benefits seen with domain-knowledge-based curricula may relate to not-yet-understood aspects of neural network training dynamics.

Replace   There are two broad approaches to **replace → training data** in the literature we reviewed: 1) replacing the training dataset with an encoded/compressed version; and 2) replacing the training dataset with a distilled dataset. Both sets of approaches usually reduce training step time T_i by operating on smaller and/or more efficient representations. The former approaches typically require changes to the model but can avoid the expensive JPEG decoding process. For instance, Gueguen et al. (2018) train directly on DCT coefficients, avoiding the JPEG decoding/decompression step. Beyond requiring model changes, a limitation of this method is that usage of traditional data augmentation strategies will require a decoding-augmentation-reencoding step, which may eliminate the speed gains. Dubois et al. (2021) learn an encoder that can perform zero-shot compression of new training datasets, achieving not only substantial bit-rate savings relative to JPEG but also encodings that can be learned from directly. Learning a compressor that aims to preserve predictability (rather than human interpretability) is an apparently new but promising direction: Dubois et al. (2021) provides a minimal script that trains an image encoder, encodes the STL dataset, and trains a linear classifier on the resulting encodings to 98.7% accuracy in under five minutes.

Another way in which data can be replaced is through data distillation (Wang et al., 2018; Zhao et al., 2020; Nguyen et al., 2020, 2021). These approaches replace the full set of training samples with a smaller set. Unlike subset selection, these approaches produce samples that may not appear in the original dataset, and can instead be the output of an arbitrary algorithm. Dataset distillation is often expressed as a bi-level meta-learning process where an "inner loop" trains a model on a distilled dataset, and an "outer loop" learns to compress that data for performance on the original dataset. Compared to training on a non-distilled dataset of equivalent size, higher performance per step of training can be reached when using distilled data. However, our literature review suggested that data distillation approaches may be less helpful for reaching the performance levels attainable by training with a large non-distilled dataset.

Mechanistically, the **replace → training data** approaches we found are primarily static, learning-based methods. Reduced model quality is a major concern when replacing the data, and it's possible that applying **replace → training data** more dynamically (e.g., at different points during training) might offer more control over the speedup-quality tradeoff.

Retrofit   **Retrofit → training data** is approached in many ways in the literature. These techniques mostly aim to improve $n_{\text{iteration}}$.

Data augmentation is a widely used method for retrofitting a dataset with more examples—see Shorten and Khoshgoftaar (2019) and Feng et al. (2021) for image and NLP surveys. For example, Dai et al. (2021); Wightman et al. (2021) show that RandAugment (Cubuk et al., 2020), mixup (Zhang et al., 2017), and CutMix (Yun et al., 2019) can improve accuracy. Further, Hoffer et al. (2019a); Fort et al. (2021); Wightman et al. (2021) accelerate data loading (and sometimes increase accuracy) by using multiple augmentations of a given image in each batch. Fort et al. (2021) provides evidence that this approach speeds up learning by reducing gradient variance arising from the data augmentation process—for a fixed batch size, the reduction in variance achieved by raising the augmentation multiplicity

enables stable training with a larger learning rate and higher performance per step. See Fort et al. (2021) to learn more about the interaction between augmentation multiplicity, learning rate, and performance/speedup. Choi et al. (2019) use the same examples (e.g., the same batch) for multiple training iterations in an epoch when the training process is CPU bound, which accelerates training by ensuring the AI accelerator doesn't idle.

Orthogonally, pre-training on large datasets that may have no or weak labels is an essential (though expensive) retrofit strategy for attaining the highest possible accuracy (Devlin et al., 2018; Dai et al., 2021; Zhai et al., 2021). Indeed, Hoffmann et al. (2022b) show that scaling up the dataset size and model size roughly equally yields the best quality for a fixed compute budget—at least for language modeling with transformers.

Our review found that **retrofit → training data** is usually approached with a domain knowledge mechanism. One exception is Raghu et al. (2020), where the authors learned augmentations to improve generalization. Learning-based **retrofit → training data** methods like AutoAugment (Cubuk et al., 2018) are difficult to develop due to the computational demands of learning augmentation policies on large-scale tasks, combined with the difficulty of transferring policies learned on other (smaller) tasks (Cubuk et al., 2020).

4.2.2 Derived Data

Our survey found that speedups obtained by targeting derived data (i.e., activations and gradients) primarily involve the **remove** and **restrict** actions. Since the targets of the speedup method are created during training (e.g., activations don't exist until the data is fed through the model), these approaches necessarily are applied during the training procedure rather than before training. Their behavior can be learning-based (e.g., learnable attention masks) or based upon domain knowledge (e.g., removing high-frequency components of feature maps).

Remove   **Remove → derived data** aims to reduce the time per training step T_i by removing activation or gradient information. A key challenge is developing a good approximation to or proxy for a training step that uses all activation and gradient information.

For convolutional layers, redundant or less helpful feature map components can be removed using frequency information (Chen et al., 2019; Dziedzic et al., 2019). Chen et al. (2019) downsample a fraction α of a feature tensor's channels by a factor of 2 in each spatial dimension and modify the filters to operate on these lower resolution channels, achieving lower complexity for the computations involving the downsampled portion of feature maps—the parameter α controls the tradeoff between accuracy and speedup for these "Octave Convolutions" that replace regular convolutions. Relatedly, Dziedzic et al. (2019) introduce band-limited convolutional layers, retaining the original feature map size in the spatial domain but performing convolution in the frequency domain via FFT, which allows the computational exclusion of feature map components according to their frequencies. Dziedzic et al. (2019) choose to exclude higher frequencies from the FFT based on the idea that models are biased towards lower frequencies. They show that the resulting models are accurate, with the fraction of frequencies discarded governing the accuracy-speedup tradeoff.

Similarly, there are various techniques that address the well-known quadratic (in sequence length) memory and time complexity of the attention operation by removing entries

from the attention matrix. This is done in either a fixed manner using the domain knowledge that nearby tokens are important or in a learnable/dynamic manner. Examples of the former include the Memory Compressed Transformer (Liu et al., 2018b) and Image Transformer (Parmar et al., 2018), while the Sinkhorn Transformer(Tay et al., 2020a) illustrates the latter. Because the central idea of these methods is to remove the quadratic complexity with respect to sequence length, they are most beneficial in situations that involve long sequences. Accordingly, Tay et al. (2020b) introduce a suite of long-range tasks to compare the various "efficient" attention mechanisms. Tay et al. (2020b) suggests that dynamic/learnable approaches are more accurate but slower than static/domain-knowledge-based approaches (see Figure 3 of Tay et al., 2020b), consistent with what our survey tended to find for dynamic/learnable versions of other speedup approaches. Tay et al. (2020c) provides a survey of methods for improving attention's efficiency, and we describe several more of these methods below in terms of other 5R speedup actions being applied to the derived data (i.e., the full attention matrix activation set, which has quadratic complexity).

Since the backward pass can be twice as expensive as the forward pass, significant speedups can be obtained by only computing gradients for a subset of examples that meet some forward-pass-related criteria. Motivated by importance sampling (Loshchilov and Hutter, 2015; Katharopoulos and Fleuret, 2018), Jiang et al. (2019) propose Selective-Backprop, a simple but effective gradient pruning technique. Given the loss for a sample, Selective-Backprop chooses to either include or omit this sample from the backward pass. Experiments on CIFAR-10, CIFAR-100, and SVHN show that Selective-Backprop can achieve a 3.5x speedup compared to standard SGD in exchange for a decrease in accuracy. Mindermann et al. (2021) similarly compute gradients for only a subset of samples but select them based on more than simply the magnitude of their loss. Indeed, they find that high-loss samples can slow down training when they correspond to noisy labels or rare exceptions. Therefore, they propose Reducible Holdout Loss Selection (RHO-LOSS) to identify the examples that are expected to improve generalization. Specifically, they only compute gradients for samples that produce a high loss for the model being trained but a low loss for a small proxy model trained on a held-out dataset. Similarly, Siddiqui et al. (2022) accelerate training by only computing gradients for samples with both high loss and high probability of being clean (e.g., not mislabeled) according to their Metadata Archaeology via Probe Dynamics (MAP-D) sample-classification approach.

Our review found that **remove → derived data** is mostly approached using dynamic, domain-knowledge-based mechanisms. Interestingly, the types of domain knowledge used in the methods we reviewed suggest that making approximations to learning-based speedup approaches is a fruitful path towards new **remove → derived data** speedup methods. For example, Parmar et al. (2018); Liu et al. (2018b) use fixed sparse attention that is an approximation to learnable sparse attention (Tay et al., 2020a), and Mindermann et al. (2021) develop approximations to avoid solving an intractable holdout-loss minimization problem over the set of possible points to train on.

Restrict Our review found that **restrict → derived data** is applied during both the forward and backward passes. Like weights, gradients and activations can be restricted via quantization or low-rank approximations to reduce T_i. Also, normalizing or otherwise restricting the distribution of derived data can avoid ill-conditioned loss landscapes

and training instability (Huang et al., 2020), accelerating training by reducing $n_{\text{iteration}}$. However, these methods must be careful to not introduce too much overhead or training instability via approximations; for example, the large time cost of standardizing derived data with Batch Normalization is shown in Figure 3.

Training efficiency can be improved by normalizing activations using estimates of their statistics (Wiesler et al., 2014; Desjardins et al., 2015) or using exact statistics computed inside normalization layers (Ioffe and Szegedy, 2015; Ba et al., 2016; Wu and He, 2018). For example, Wiesler et al. (2014) introduce mean-normalized SGD, a preconditioning approach to keeping activations mean-centered that improves model quality and convergence rate. Alternatively, Batch Normalization (Ioffe and Szegedy, 2015) layers standardize activations using mini-batch statistics, ensuring activations seen during training always have zero mean and unit variance. To avoid issues with computing statistics in a way that depends on batch size, Ba et al. (2016) introduce layer normalization, and Wu and He (2018) generalize layer normalization with group normalization. For more discussion of activation normalization methods, please see Huang et al. (2020).

Approaches that normalize gradients also exist and often aim to exploit curvature information to improve the optimization trajectory. Pascanu et al. (2013) propose clipping the gradient's norm to a threshold in order to stabilize training and improve the model quality attained per iteration. Similarly, Yu et al. (2017) propose normalizing gradients in a layer-wise manner (block normalized gradient descent) to avoid vanishing/exploding gradients. In particular, they find better performance per step when scaling the gradients with respect to the weights such that their norms are proportional to the norms of their associated weights. Other gradient normalization approaches have helped training in large batch (You et al., 2017) and mixed batch-size (Hoffer et al., 2019b) contexts. Instead of scaling, Yong et al. (2020) propose gradient centralization, which computes the mean of each gradient tensor along some axis then subtracts these means so that the resulting centralized gradients have zero mean. Yong et al. (2020) show that this approach accelerates training; they attribute this speedup to the stabilization brought by the reduced scale of the gradient and the regularization from ensuring that the sum of weights associated with a neuron/channel is constant during training. For more discussion of gradient normalization methods, please see Huang et al. (2020).

In distributed training contexts, the cost of communicating the full gradient can be the bottleneck; restricting gradients via compression techniques can therefore provide speedups (Xu et al., 2020). For example, Seide et al. (2014) propose 1-bit SGD in which 32-bit gradient elements are approximated by 1-bit representations; this leads to significant speedups with little to no accuracy loss. The accuracy preservation is a result of their "error feedback" mechanism, which adds the cumulative quantization error to the gradient before quantization. Bernstein et al. (2018) also quantize gradients to 1 bit via signSGD, which approximates negative gradient components with -1 and others with 1. 1-bit Adam (Tang et al., 2021) and 0/1 Adam (Lu et al., 2022) make extreme gradient compression compatible with the Adam optimizer. Dettmers (2015) show that 32-bit gradient data can be approximated with a variety of 8-bit representations that do not significantly increase error but greatly reduce memory requirements for large models. Beyond quantization, gradients can be compressed through low-rank decomposition of the gradient via methods like PowerSGD (Vogels et al., 2019) and GradZIP (Cho et al., 2019), as well as through

sparsification/remove methods. For a more thorough discussion of gradient compression techniques, please see (Xu et al., 2020).

Low-rank approaches have also been used to restrict transformer activations to achieve speedups. For instance, Linformer (Wang et al., 2020) removes the quadratic complexity in sequence length N of transformer attention to obtaining faster training on long sequences. They do this by imposing a low-rank restriction on the attention matrix, the $N \times N$ matrix of activations that multiplies the $N \times d$ value (V) matrix. Specifically, Wang et al. (2020) add new $k \times N$ matrices E and F to the attention operation in order to project the key (K) and value (V) matrices down from $N \times d$ to $k \times d$, turning attention from $Softmax(QK^T)V$ into $Softmax(Q(EK)^T)FV$.

Our review found that **restrict $\rightarrow$ derived data** is approached in both static and dynamic ways. However, most approaches depend on domain knowledge to achieve speedups, suggesting that learning-based approaches may be an interesting target for future development.

Reorder   Our review found few ways to **reorder $\rightarrow$ derived data** to produce a speedup, which may reflect the difficulty of developing approaches in this area.

Exemplifying such approaches, Matani and Subramanian (2021) reorder the input and activation axes into a channels-last memory format and show that this can accelerate training. This is partly a product of memory access patterns and partly an artifact of software support, but mostly a consequence of the channels-last format enabling a more efficient reduction of convolution to large, contiguous, implicit GEMM operations in accelerators.

The **reorder $\rightarrow$ derived data** speedup methods we found (Matani and Subramanian, 2021; Dao et al., 2022b) use static, domain-knowledge-based mechanisms. The scarcity of approaches in this category suggests there may be opportunities for more, potentially challenging work in this direction.

Replace   Our review found several ways to **replace $\rightarrow$ derived data**, including activation- and gradient-focused approaches. Most approaches focused on reducing T_i. A major challenge in this area is reaching the model quality attained with the original derived data.

Replacing gradients with their stale counterparts can accelerate training when gradient communication is a bottleneck. Such asynchronous updates are particularly effective when the learning rate is reduced more for gradients that are more stale (Dutta et al., 2018). Relatedly, Jiang et al. (2019) further accelerated training with Selective-Backprop—removal of the backward pass on low loss samples—by using stale losses; this is a clear improvement when a sample's loss does not change across epochs.

Kernel-based approaches to transformer attention such as Linear Transformer (Katharopoulos et al., 2020), Performer (Choromanski et al., 2020), and Scatterbrain (Chen et al., 2021b) make transformers faster on long sequences through replacement of the traditional attention matrix $Softmax(QK^T)$ with the product of the kernel feature representations of the query (Q) and key (K) matrices: $\phi(Q)\phi(K)^T$. This allows the costly softmax attention to be approximated or replaced with a kernel formulation $\phi(Q)\phi(K)^TV$. When reordered as $\phi(Q)(\phi(K)^TV)$, this formulation has linear complexity with respect to the sequence length.

The **replace $\rightarrow$ derived data** methods discussed are based on domain knowledge. Methods that alter the attention matrix, including those reviewed here, could be viewed as static replacements of the architecture. We emphasize their effect on derived activations by

classifying them as dynamic modifications of the attention matrix, perhaps suggesting new efficient transformer approaches that utilize ideas from other derived-data approaches.

Retrofit   There exist a few ways to **retrofit → derived data** to achieve speedup.

Neelakantan et al. (2015) propose a low-overhead and easy-to-implement speedup technique: add time-dependent Gaussian noise to the gradients at every training step. The authors found that adding annealed Gaussian noise (i.e., decaying the variance over time) can improve model quality significantly, reducing the $n_{\text{iteration}}$ required to achieve the desired quality.

Press et al. (2021) propose Attention with Linear Biases (ALiBi), which adds a penalty to attention activations that encourages attention at different length scales for different heads. This enables models trained on shorter sequence lengths, which can be trained more quickly, to perform competitively with baseline models when evaluated on longer sequence lengths at test time.

Our review found that **retrofit → derived data** is mostly approached using dynamic, domain knowledge mechanisms. We expect that more accurate methods could be designed by leveraging learning-based mechanisms.

4.3 Optimization Speedup Strategies

Optimization-related speedup methods apply the 5Rs to the optimization algorithm and training objective subcomponents. Most of them aim to minimize the number of iterations $n_{\text{iteration}}$ required to achieve the desired model quality. For example, a **retrofit → training objective** approach that adds sharpness information to the loss function can improve model quality gained per optimization step (Foret et al., 2020). Often, these methods require increased computation. When this is the case, they are most useful when the accelerators are underutilized due to communication or data loading bottlenecks.

4.3.1 TRAINING OBJECTIVE

Remove   **Remove → training objective** primarily aims to speed up training by reducing T_i through approximation of expensive loss functions. As with many remove/restrict approaches, a key challenge here is developing an approximation that maintains the model quality associated with the original loss function.

Loss functions can be expensive to compute when neural networks have large output spaces. For example, when the vocabulary size is large, the final softmax classifier can become a bottleneck (Chen et al., 2015b).

To combat this bottleneck, Morin and Bengio (2005) introduce the hierarchical softmax approach to compute the probability of a target output word as a product of probabilities associated with nodes along a path of a binary tree having leaves that span the entire vocabulary. This requires work logarithmic in the number of classes, rather than linear. Sampling-based approaches achieve a similar removal through different means; Mikolov et al. (2013) develop negative sampling, which removes from the objective all word probabilities except the target word's and those for several negative sample words drawn from a noise distribution.

Loss functions can also be expensive to compute when they include higher-order information like loss surface sharpness. Accordingly, Liu et al. (2022a) focus on improving the

efficiency of Sharpness-Aware Minimization (SAM) (Foret et al., 2020), which uses perturbed weights at each iteration to bias optimization towards flatter minima. Specifically, Liu et al. (2022a) propose to reuse the same weight perturbation for SAM across several iterations, greatly reducing overhead while preserving accuracy. Similarly, Du et al. (2021) compute the SAM weight perturbation for only a subset of weights and using only a subset of samples. They find that these removals greatly accelerate SAM with no loss of model quality. More recently, Du et al. (2022) introduced Sharpness-Aware training for Free (SAF), which removes the weight perturbation term from the SAM objective and uses changes in the loss over time to recover an analogous sharpness-aware objective; this approach has roughly the same computational cost as vanilla training and accuracy/sharpness-avoidance benefits similar to SAM's, which can speed up training by reducing the number of iterations required.

The **remove → training objective** strategies we reviewed depended on domain-knowledge based mechanisms that offered limited control over the drop in model quality caused by the removal. In future work, it may be fruitful to dynamically choose the fidelity of one's approximation to SAM, softmax, etc.

Restrict The only **Restrict → training objective** method we found is that of Akbari et al. (2021), who restrict the geometry of the loss landscape by altering the loss function. Specifically, they introduce the Generalized Jeffries-Matusita loss, which has a smaller Lipschitz constant than a Kullback-Leibler divergence loss. They show that their loss yields lower generalization error after a fixed number of steps. It also enables tighter generalization guarantees.

Our review suggests that **restrict → training objective** is still in an early stage. The approach we reviewed relies on domain knowledge, and we expect that more accurate methods could be designed by developing a deeper understanding of the interaction between loss landscape and generalization error.

Reorder We found two methods that **reorder → training objective**, both of which involve changing the loss function and/or regularization throughout training.

Wu et al. (2018b) propose Learning to Teach with Dynamic Loss Functions (L2T-DLF), in which the loss function at iteration t is the output of a teacher model. This teacher model's input consists of t and several variables describing the student model's performance. The authors show how to optimize the teacher model such that the series of loss functions it outputs lead to more model quality gained per student model weight update, reducing $n_{\text{iteration}}$ on both vision and language tasks.

Similarly, Tan and Le (2021) show that more model quality is gained per training step when regularization is adjusted as inputs are resized: they lower $n_{\text{iteration}}$ without accuracy loss by jointly increasing image resolution, dropout rate, and data-augmentation magnitude throughout training.

While these methods use both knowledge- and learning-based mechanisms, the knowledge utilized and learned was somewhat specific to the tasks considered. This may mean there are opportunities for more general **reorder → training objective** speedup methods. E.g., perhaps one could generalize the resolution-dependence in the image model of Tan and Le (2021) to sequence-length dependence in language models. There might also be

opportunities to employ information about the current loss landscape or different phases of training (Achille et al., 2017; Frankle et al., 2020b).

Replace **Replace → training objective** is a common technique for achieving speedups through improvements to model quality gained per optimization step. Work in this area is typically faced with the challenge of outperforming common objectives like cross-entropy or mean squared error (MSE).

When using mixup and CutMix data augmentation, Wightman et al. (2021) replace the typical cross-entropy loss with one designed to align better with human perception, speeding up training of ResNet-50 on ImageNet. These specific augmentations are relevant because they produce images and labels that are combinations of multiple images (usually from different classes) and their labels; for example, mixup's images have the form $\tilde{x} = \lambda x_i + (1 - \lambda)x_j$ and mixup's labels have the form $\tilde{y} = \lambda y_i + (1 - \lambda)y_j$, where y_i, y_j are the one-hot labels for images x_i, x_j and $\lambda \in [0, 1]$. Optimizing the cross-entropy loss applied to the predicted label vector and $\tilde{y}$ as is typical encourages the model to predict a λ probability of the presence of class y_i in image $\tilde{x}$, whereas humans can typically recognize the presence of class y_i in $\tilde{x}$ with certainty. Therefore, Wightman et al. (2021) instead assume that all classes used to construct $\tilde{x}$ are discernibly present in $\tilde{x}$ by setting each of their labels to 1. To accommodate this change, they treat the problem as a multi-label classification and optimize the binary cross-entropy (BCE) loss for each class.

Gonzalez and Miikkulainen (2020) apply genetic programming to learn loss functions from primitive operations using MNIST validation dataset performance as a signal. They term their final learned loss function the "Baikal loss." Despite being discovered via MNIST data, Baikal continued to provide improvements relative to cross-entropy when evaluated on CIFAR-10 data. Along the same lines, Bechtle et al. (2021) propose a meta-learning approach to training parameterized loss functions, Meta-Learning via Learned Loss (ML³). On previously unseen tasks, the authors show that the meta-learned loss functions improve convergence speed.

In terms of mechanisms, our review found that **replace → training objective** is approached mostly in a static manner. We suspect that dynamic mechanisms—i.e. altering the loss function used based on the data and state of training—may facilitate further improvements in the emerging learning-based **replace → training objective** literature (Bechtle et al., 2021; Gonzalez and Miikkulainen, 2020).

Retrofit **Retrofit → training objective** is a straightforward way to reduce $n_{\text{iteration}}$. This popular set of approaches features a variety of ways to incorporate domain-specific information or known learning principles as additional objectives.

Von Rueden et al. (2019) survey various ways to inject prior knowledge into loss functions to accelerate training and improve model quality. They present approaches in terms of how the knowledge is represented (e.g., as algebraic equations) and its source (e.g., physical laws). Generally, these approaches accelerate training by penalizing model inconsistencies with user-provided knowledge about the data. Wang and Yu (2021) survey methods for incorporating prior knowledge of physics, many of which entail adding terms to the loss. Kukačka et al. (2017) discuss various application-agnostic ways to regularize deep learning. For example, adding an L_2 penalty is a common and often helpful practice.

Perhaps the most notable recent work that retrofits the training objective is sharpness aware minimization (SAM) (Foret et al., 2020). Based on the observation that flatter loss minima are associated with better generalization (Keskar et al., 2016), Foret et al. (2020) add sharpness information to the training objective to encourage convergence to a flatter minimum. SAM produces better model quality per step, but its steps require much more time to execute due to the use of an additional gradient computation to determine sharpness. However, efficient variants of SAM exist, which we discuss in the **remove** → **training objective** section.

Regarding mechanisms, the **retrofit** → **training objective** methods we reviewed rely heavily on domain knowledge. This suggests that deepening our understanding of learning principles might provide means to further enhance training objectives. There may also be opportunities for speedups through methods that learn a new element of the training objective rather than learning the entire loss function (Bechtle et al., 2021).

4.3.2 ALGORITHM

Remove The high-level algorithm for gradient-based training of neural networks (stochastic gradient descent, or SGD) is extraordinarily simple given its effectiveness. Accordingly, a major challenge for approaches that seek to **remove** → **algorithm** is preservation of model quality.

Removal of all gradient computations is one path toward accelerating training. Indeed, there are various zeroth-order optimization (ZOO) training algorithms (see Liu et al., 2020, for a review). For example, Zeroth-Order Relaxed Backpropagation (ZORB) (Ranganathan and Lewandowski, 2020) starts its backward pass with the true sample labels at the output layer, then successively applies the pseudoinverse operation to each layer's weights to determine what each preceding layer should output in order to predict the labels at the output layer. A single forward pass then applies the pseudoinverse to the data sample input to solve for the first-layer weights that will produce the desired first-layer output determined by the backward pass. The output of this layer then becomes the input used to solve for the next layer's weights, and so on. Using small networks on small datasets, this approach can match the accuracy of gradient-based algorithms like Adam while being 300 times faster. However, this pseudoinverse-based approach is difficult to scale up.

Rather than remove computationally intense steps from each iteration of an algorithm, many methods remove entire iterations through weight prediction. For example, Dogra and Redman (2020) introduce Koopman training, which uses Koopman operator theory to learn and quickly predict optimization trajectories. This allows them to extrapolate future weight updates without gradient evaluations. Notably, though, Koopman training can lead to model accuracy degradation and higher memory costs due to the need to track weight evolution. Sinha et al. (2017) provide a similar approach but use an "introspection network" trained to predict weight changes on an easy task (MNIST) to predict weight changes on general tasks (e.g., ImageNet). Knyazev et al. (2021) remove all optimizer iterations and simply predict performant parameters using a Graph HyperNetwork (GHN) in under one second. While this prediction network can be applied to unseen architectures, it must be trained initially and its predicted weights are not as performant as weights optimized by gradient-based methods.

Remove → algorithm mostly relies on computationally-intense, learning-based mechanisms. However, consistent with the difficulty of removing elements from an already simple training algorithm, these methods often show significant accuracy degradation, especially on moderate to large-scale data such as ImageNet. We nonetheless expect more work in this emerging area as neural network training becomes better understood.

Restrict The **restrict → algorithm** approaches we review reduce $n_{\text{iteration}}$ and thereby produce speedups by ensuring that training stays in an efficient regime (e.g., avoiding exploding or vanishing gradients). Mostly based on domain knowledge, these approaches depend on insights regarding choices like initialization scale and step size.

For example, He et al. (2015) restrict the initialization of weights in deep DNNs with ReLU activations to a symmetric distribution with variance 2/fan-in. Hanin and Rolnick (2018) provide a theoretical framework for understanding the importance of this restriction, showing that either uniform or normal weight-initialization distributions that employ it avoid failure modes that can arise with other distributions.

Vorontsov et al. (2017) restrict the weights at initialization and during optimization to be approximately orthogonal via a spectral margin parameter that controls deviations from orthogonality, with a margin of zero corresponding to exact orthogonality. They find that baseline RNNs without any margin constraint can converge slowly or not at all, while networks trained with margins slightly greater than 0 converged faster than both unconstrained and 0-margin networks. This result suggests a benefit to balancing between a hard constraint and no constraint at all. Hu et al. (2020) provide theoretical evidence supporting the idea that orthogonality restrictions simply at weight initialization (rather than throughout training) can be sufficient to speed up convergence.

McCandlish et al. (2018) theoretically and empirically support the idea that the optimal batch size depends on the interaction between the batch size and the "gradient noise scale," where the latter measures the inter-batch variability of the gradients. They find that when the noise scale is large relative to the batch size, one can increase the batch size to decrease the number of optimization steps required, providing a speedup when parallelism prevents an offsetting increase in T_i. Smith et al. (2017) find that one can also increase the batch size by using this as a substitute for decreasing the learning rate.

When using large learning rates, Smith and Topin (2017) find that restricting the level of weight decay to account for the current learning rate can help avoid excessive regularization.

The **restrict → algorithm** mechanisms in our review were mostly static and domain-knowledge based. We suspect that more dynamic **restrict → algorithm** mechanisms based on domain knowledge could produce further speedups. For example, the analyses of McCandlish et al. (2018) and Wang et al. (2022) suggest that the batch size and momentum parameters (respectively) should be adaptively set due to their dependence on dynamic quantities.

Reorder **Reorder → algorithm** is mostly done to alter the progression through a learning rate (LR) schedule. Providing an effective schedule for a given task and model is the main challenge in this area.

When adjusting $n_{\text{iteration}}$ while using a cosine annealing schedule, Hoffmann et al. (2022b) find that the schedule length must be adjusted such that at least 80% of the schedule is completed during the reduced training time window, or else accuracy suffers. Moreover,

the best performance is attained when the end of the schedule coincides with the end of training. One can also repeat a schedule multiple times, possibly with variations each time. Such cyclic learning rate schedules have empirically and theoretically produced speedups, at least in some circumstances (Smith, 2017; Loshchilov and Hutter, 2016; Oymak, 2021; Goujaud et al., 2021).

The LR scheduling approaches we found are primarily fueled by domain knowledge. However, learning-based solutions for **reorder → algorithm** are emerging. E.g., Baik et al. (2020) accelerate MAML by using a task-and-iteration-specific learning rate produced by a meta-learned hyperparameter network.

Replace **Replace → algorithm** covers some of the most well-known ways to speed up training. These methods generally aim to replace the optimization algorithm with one that reduces $n_{\text{iteration}}$. Fair evaluations are a major concern in this area due to interactions between optimizers and hyperparameters like weight decay that can create misleading performance comparisons (Schmidt et al., 2021).

Perhaps the oldest work in this category is that of Polyak (1964), who introduces momentum to the gradient descent algorithm. Later, Hanson and Pratt (1988) found that adding a decay step to weight updates can improve the performance gained per iteration at negligible compute cost. This decay entails multiplying the current weights by a small constant as part of each update.

More recently, there have been many "adaptive gradient" methods that aim to speed up training by scaling the effective learning rate intelligently throughout training. This scaled learning rate is often specific to a given parameter and is a function of past gradients. Some of the most well-known methods of this type include RMSProp (Tieleman and Hinton, 2012), AdaGrad (Duchi et al., 2011), and Adam (Kingma and Ba, 2014).

While replacing SGD with adaptive gradient methods can speed up achievement of a given performance level (e.g., see Dosovitskiy et al., 2020), combining them with weight decay involves subtleties. In particular, Loshchilov and Hutter (2017) note that weight decay should be differentiated from adding an L_2 penalty to the loss, as the latter can produce undesired effects and worse performance. Loshchilov and Hutter (2017) address this performance issue with the AdamW optimization algorithm, which replaces L_2 regularization with weight decay to improve Adam's performance. AdamW is commonly used to train large language and vision models (Hoffmann et al., 2022b; Dai et al., 2021).

Analogous to the replacement of hand-designed features with neural network learned features, Metz et al. (2019) show that hand-designed optimizers can be replaced by faster-learned optimizers. A limitation of this approach is the need to train a task-specific optimizer; moreover, the generalization of such optimizers across tasks is unclear.

Making subtler changes, Wightman et al. (2021) illustrate that optimization generally requires careful tuning based on the task/architecture. Concretely, one can reduce $n_{\text{iteration}}$ by tuning the learning rate, weight decay, and other optimizer hyperparameters.

Parameter initialization at the start of learning provides another opportunity for replacement. Replacing the random initialization of weights with pretrained weights is a particularly popular approach to achieving speedups in this area (Han et al., 2021). For example, Brown et al. (2020b) train the 175 billion-parameter language model GPT-3 and demonstrate that it is flexible enough to provide near-SOTA accuracy on some downstream

learning tasks without any fine-tuning. Critically, this means transfer learning (i.e., using a pretrained model's weights at initialization) may speed up training by reducing the $n_{\text{iteration}}$ required to reach a desired model quality to *zero*. Bommasani et al. (2021) provide a related transfer-learning report and introduce the term "foundation models" for these massive, flexible, pretrained models. Separately, Finn et al. (2017) introduce model-agnostic meta-learning (MAML) for learning initializations that facilitate rapid learning on new tasks.

Our review found numerous **replace** $\rightarrow$ **algorithm** approaches, using all possible mechanisms except static changes based on domain knowledge. This unexplored mechanism may offer an interesting direction for future work.

Retrofit   Our review found several ways to accelerate training with **retrofit** $\rightarrow$ **algorithm** techniques. Generally, a challenge for these techniques is avoiding increases in T_i that overwhelm the reduction to $n_{\text{iteration}}$ they provide.

Adding second-order/curvature information to weight update rules can significantly improve the performance gained per training step (Martens and Grosse, 2015; Gupta et al., 2018; Anil et al., 2020a; Goldfarb et al., 2020). To address the greatly increased cost of computing these updates, Anil et al. (2020a) approximates the large gradient-preconditioning matrices and also updates them only occasionally. These changes reduce the preconditioning overhead enough to yield a large reduction in overall training time.

The **retrofit** $\rightarrow$ **algorithm** methods in our review use dynamic, domain-knowledge-based mechanisms. Given that the major challenge these approaches face is minimizing the overhead they create, future work could consider learning-based mechanisms as a way to incorporate the cost of the retrofitting (e.g., preconditioning steps) into decisions about whether or how much to apply it.

Table 3 summarizes our taxonomy's classification of the speedup methods we explicitly discussed and related methods.

4.4 Other Directions

Beyond the algorithmic speedup approaches that we reviewed above, there are numerous compute-platform approaches to achieving faster training. Broadly, these approaches include faster implementations/kernels for common operations, better hardware, and parallelism. For instance, without changing hardware, FFCV (Leclerc et al., 2022) speeds up data preprocessing with just-in-time (JIT) compiled data augmentations and other data-pipeline optimizations. Data-pipeline optimizations tailored to distributed training, such as NoPFS, create large speedups as well (Dryden et al., 2021). There are numerous approaches to large-scale parallel training (Shazeer et al., 2017; Ben-Nun and Hoefler, 2019; Rajbhandari et al., 2022; Team and Majumder, 2020; Narayanan et al., 2021), each with various tradeoffs depending on accelerator count, interconnect, accelerator memory, batch and input sizes, model structure, and more.

5. Best Evaluation Practices

Speedup comparison is a challenging problem, and existing papers rarely make satisfactory comparisons (Dehghani et al., 2021; Wu et al., 2020; Blalock et al., 2020). Stemming

Action	Model	Architecture	Data Sample	Dataset	Derived Data	Objective	Algorithm
Remove	**Yuan et al. (2021)**, Zhang et al. (2021), Van Gelder et al. (2020), Hoefler et al. (2021), Mocanu et al. (2018), Price and Tanner (2021), You et al. (2019), Hubsen et al. (2021), Liu et al. (2021a), Chen et al. (2021d), Sung et al. (2021), Dey et al. (2019), **Chen et al. (2020)**, **Lym et al. (2019)**, **Adelman et al. (2021)**, **Chen et al. (2022)**	Martens et al. (2021), **Brock et al. (2021)**, **Huang et al. (2016)**, Fan et al. (2019)	**Stephenson (2021)**, DeVries and Taylor (2017), Liu et al. (2023)	Coleman et al. (2019), Killamsetty et al. (2021a), **Raju et al. (2021)**, Sorscher et al. (2022), Kaushal et al. (2019), Birodkar et al. (2019), Mirzasoleiman et al. (2020), Killamsetty et al. (2021b), **Chitta et al. (2021)**, Toneva et al. (2018), Paul et al. (2021), **Mittal et al. (2022)**, Sivasubramanian et al. (2021), Karanam et al. (2022), Killamsetty et al. (2021c), Wei et al. (2015)	**Mindermann et al. (2021)**, Jiang et al. (2019), Killamsetty et al. (2020), Wimmer et al. (2020), **Dziedzic et al. (2019)**, **Chen et al. (2019)**, Parmar et al. (2018), Liu et al. (2018b), Tay et al. (2020a), Liu et al. (2018a), Siddiqui et al. (2022)	Liu et al. (2022a), **Mikolov et al. (2013)**, Morin and Bengio (2005), **Du et al. (2021)**, Chen et al. (2015b), **Du et al. (2022)**	**Dogra and Redman (2020)**, **Knyazev et al. (2021)**, **Sinha et al. (2017)**, Liu et al. (2020), **Ranganathan and Lewandowski (2020)**
Restrict	**Chen et al. (2021a)**, Idelbayev and Carreira-Perpinán (2020), Blalock and Guttag (2021), **Micikevicius et al. (2017)**, Ioannou et al. (2015), **Sainath et al. (2013)**, **Hu et al. (2021)**, **Dao et al. (2022a)**, Liebenwein et al. (2021), Gholami et al. (2021)	**Press and Wolf (2016)**	Huang et al. (2020)	Kailkhura et al. (2020), Jin et al. (2020)	Xu et al. (2020), **Wang et al. (2020)**, Pascanu et al. (2013), You et al. (2017), Huang et al. (2020), Wiesler et al. (2014), **Desjardins et al. (2015)**, Ioffe and Szegedy (2015), Ba et al. (2016), **Wu and He (2018)**, Yong et al. (2020), Yu et al. (2017), Dettmers (2015), Seide et al. (2014), Bernstein et al. (2018), Vogels et al. (2019), Cho et al. (2019), Lu et al. (2022), Tang et al. (2021)	**Akbari et al. (2021)**	Dettmers et al. (2021), Smith and Topin (2017), **Vorontsov et al. (2017)**, Wang et al. (2022), Hu et al. (2020), **He et al. (2015)**, Hanin and Rolnick (2018), **McCandlish et al. (2018)**, Smith et al. (2017)
Reorder	**He et al. (2021)**, Brock et al. (2017), **Raghu et al. (2017)**	Huang et al. (2018), **Saharia et al. (2022)**, **Gong et al. (2019)**, **Li et al. (2022a)**, **Gu et al. (2020)**, **Chen et al. (2015a)**	**Hoffer et al. (2019b)**, Touvron et al. (2019), Howard (2018)	Weinshall et al. (2018), Wu et al. (2020), Raghu et al. (2020), Hacohen and Weinshall (2019), **Bengio et al. (2009)**, Wang et al. (2021)	**Matani and Subramanian (2021)**, **Dao et al. (2022b)**	**Wu et al. (2018b)**	**Oymak (2021)**, **Smith (2017)**, Goujaud et al. (2021), **Loshchilov and Hutter (2016)**, **Baik et al. (2020)**
Replace	Kusupati et al. (2021), Son et al. (2018), Sarwar et al. (2017), **Shen et al. (2020)**	Han and Hong (2021), **Chowdhury et al. (2021)**, Ding et al. (2021), Moczulski et al. (2015), Yang et al. (2015), Wu et al. (2021), Jeon and Kim (2018), Liberis et al. (2021), **Tan and Le (2021)**, Srinivas et al. (2021), **Dai et al. (2021)**, Bello et al. (2021), **Lee-Thorp et al. (2021)**, **Timmons and Rice (2020)**, Tay et al. (2020c), Tay et al. (2020b), Narang et al. (2021), Prasad et al. (2022), Brown et al. (2022), **Geiping and Goldstein (2022)**	Gueguen et al. (2018), Pistono et al. (2020), Dubois et al. (2021), Le et al. (2021), Fu and Guimaraes (2016)	Sucholutsky and Schonlau (2020), Nguyen et al. (2021), **Nguyen et al. (2020)**, Zhao et al. (2020), **Wang et al. (2018)**	**Chen et al. (2021b)**, **Choromanski et al. (2020)**, Katharopoulos et al. (2020), Dutta et al. (2018)	**Wightman et al. (2021)**, Gonzalez and Miikkulainen (2020), **Bechtle et al. (2021)**, Steiner et al. (2021)	Tessera et al. (2021), Metz et al. (2019), Baydin et al. (2022), Kingma and Ba (2014), Loshchilov and Hutter (2017), **Brown et al. (2020b)**, Nakerst et al. (2020), Nesterov (1983), Tieleman and Hinton (2012), Duchi et al. (2011), Han et al. (2021), Bommasani et al. (2021), **Finn et al. (2017)**, Silver et al. (2021), **Xie et al. (2022)**, Yang et al. (2021)
Retrofit	Fedus et al. (2021), Gontijo-Lopes et al. (2021), **Kaddour (2022)**, **Yang et al. (2019)**, Li et al. (2022b), **Izmailov et al. (2018)**	So et al. (2021), **Li et al. (2020)**		Choi et al. (2019), **Hoffer et al. (2019a)**, **Fort et al. (2021)**, Yun et al. (2019), **Zhang et al. (2017)**, **Cubuk et al. (2020)**, Hoffmann et al. (2022b), **Cubuk et al. (2018)**, Devlin et al. (2018), Zhai et al. (2021), Shorten and Khoshgoftaar (2019), Feng et al. (2021)	**Neelakantan et al. (2015)**, **Press et al. (2021)**	Foret et al. (2020), Von Rueden et al. (2019), Wang and Yu (2021), Kukačka et al. (2017)	**Anil et al. (2020a)**, Goyal et al. (2017), **Polyak (1964)**, Martens and Grosse (2015), Goldfarb et al. (2020), Gupta et al. (2018), **Hanson and Pratt (1988)**

Table 3: **Surveyed Literature in Our Taxonomy.** Bold-font approaches explicitly show training speedups. Plain-font approaches target other efficiency metrics, survey a particular set of speedup methods, and/or have not yet provided clear evidence of training speedups to the best of our knowledge.

largely from a lack of experimental standardization and poor evaluation protocols, this limitation could lead to missed opportunities at best and misleading conclusions at worst. To overcome this limitation, speedup comparisons must achieve the following desiderata: 1) the comparison must be fair; 2) the evaluation must be comprehensive, and 3) the results must be reliable. Next, we discuss some common pitfalls of existing evaluation practices and provide suggestions on how to avoid them.

Clarify the Goal   Before discussing how to properly evaluate a speedup method, it is important to clarify its goal. E.g., is the method designed to accelerate training on natural image classification tasks or for all CNNs? Should it work for any model with an attention layer or only decoder models trained on English text? While wide applicability is desirable, improvements on a well-scoped class of problems are more useful than those on an ill-defined class of problems.

Report Experimental Setup and Hyperparameters   Settings of hyperparameters such as the learning rate and weight decay can affect the speed-quality tradeoff created by a speedup method. Similarly, architecture variations and differences in data preprocessing can greatly impact the end results.

To accurately attribute the performance gain of a speedup method to the proposed component-action change, it is important to hold the hyperparameters constant to the greatest extent feasible. Moreover, to facilitate future comparisons to one's proposed method, it should be clear in either the source code or the paper what hyperparameters were used. In the case that one also wishes to claim robustness to hyperparameter choices, it is also necessary to sweep the relevant hyperparameters and report the induced variation.

Control the Confounding Variables   Training time is also strongly dependent on factors unrelated to algorithmic speedup actions. Such confounding factors can be hardware related (i.e., CPU/GPU/TPU/etc. aspects) and/or software related (i.e., libraries, data loading, and other code). For example, an architecture can have significantly different training speeds on different hardware (Dehghani et al., 2021). Similarly, different libraries (e.g., JAX, PyTorch, and TensorFlow) are known to yield different accuracies for the same architecture and dataset, and variations in software support for certain operations can create a large difference in the train time.

Because it is impractical to perform experiments on all possible hardware and software variations, one needs to ensure that methods being compared use identical hardware and software to the greatest extent possible (Narang et al., 2021).

Perform a Complete Characterization   Achieving speedup is inherently a multi-objective optimization problem that requires optimizing for two conflicting objectives simultaneously—accuracy and train time. The optimization outcome is characterized by the accuracy-efficiency Pareto frontier. This tradeoff curve can be obtained by varying hyperparameters related to a component-action combination (e.g., the degree of sparsity for model, or data, pruning).

Accordingly, we recommend that authors report the tradeoff curve by varying hyperparameters at a sufficient granularity (e.g., using at least 3 operating points) for a given dataset and architecture. The curve for one's proposed method should be shown alongside the curves for competing methods. Further, high-accuracy solutions should be prioritized
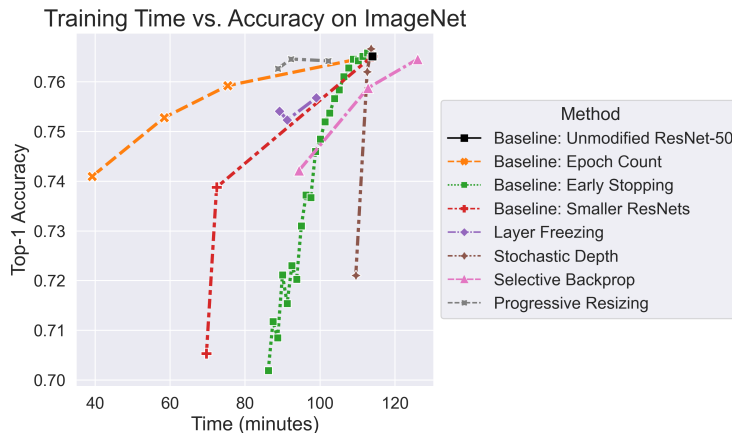
Training Time vs. Accuracy on ImageNet

Figure 13: Training for fewer epochs ("Epoch Count") is a strong baseline, but only if one iterates through the full learning rate schedule (c.f. "Early Stopping"). Methods that speed up training may do so at a greater cost to accuracy than simply reducing epoch count. Some methods, like Selective Backpropagation (Jiang et al., 2019), may slow down training when using certain hyperparameters.

when selecting operating points. Ideally, authors should also use at least three dataset-architecture pairs, including modern, large-scale datasets/models.

Use Simple, Strong Baselines as Sanity Checks   In addition to the baseline training recipe, the most obvious baselines to evaluate against are the existing techniques resembling the method being proposed.

However, we have found that there are several general-purpose baselines that can often outperform proposed speedup methods. In Figure 13, we train ResNet-50 with various speedup methods proposed in the literature and available in Composer (Team, 2021). We chose these methods because all had tested implementations in a common library. Moreover, all but one were included in the winning submission to the 2022 MLPerf Open Division (using these same implementations), suggesting that these speedups have an unusually large degree of empirical support. See Appendix B for more details.

The simple baseline of training for less time outperforms nearly all proposed methods. Crucially, this is only true when the learning rate schedule is adjusted along with the training time. Simply aborting a single training run early—the "Early Stopping" baseline—is far less effective. This discrepancy between a full, short training run and an aborted, long training run is consistent with the findings of Hoffmann et al. (2022a).

Using a smaller model can also be a solid baseline. In this case, we find that training a ResNet-18 or ResNet-34 rather than a ResNet-50 can outperform some speedup methods with certain hyperparameters. For simpler architectures (e.g., most recent NLP architectures), one could also reduce the depth or width algorithmically.

These results do not indicate that the tested speedup methods are necessarily ineffective. This is because changing the hyperparameters, model, dataset, optimizer, etc. could alter

the efficacy of these methods and the baselines. E.g., if we were to make the model $1000\times$ larger and train for a single epoch, it is likely that reducing model size would be a more effective speedup than further reducing training time.

What these results do indicate is that simple approaches like reducing the epoch count and model size can be strong baselines. We, therefore, recommend that authors use these baselines in their speed-accuracy tradeoff curves.

Account for Composition   Along the same lines as holding hyperparameters and libraries constant when comparing methods, one must also hold the set of speedup methods constant. This is because a given method may be more or less effective depending on what other methods are present.

This is illustrated in Figure 14, where we compare results across the fastest known ResNet-50 training "recipes" (Leavitt, 2022) and show the time-accuracy Pareto frontier of the methods from Figure 13. Each recipe obtains a different time vs. accuracy tradeoff, with different points within a curve representing different training runs with varying epoch counts. Details about these recipes are given in Appendix B.

The most obvious pitfall is failing to account for the benefits of other methods. In Figure 14 (left), we see that simply composing multiple methods improves the time vs. accuracy curve greatly compared to the curve made from the best points attainable by using any single method. If a new paper uses more algorithmic speedup methods in its baseline than an older paper, the former might incorrectly attribute its better numbers to its proposed method rather than its algorithmically-enhanced baseline setup.

A subtler source of error is that different methods are effective for different amounts of training time. In Figure 14 (right), the "MosaicML Medium" and "MosaicML Hot" recipes differ only in the presence of regularization methods (see Appendix B for details). These methods harm accuracy at short training times but help it at long training times.

Ensure Reliability and Significance of Results   We recommend that authors perform multiple experimental runs with separate initializations and random seeds. When computationally feasible, results should include clearly defined error bars and a measure of central tendency (e.g., mean) and variation (e.g., standard deviation). As an alternative to reusing the same settings multiple times with different seeds, one can also evaluate more points in the speed vs. quality tradeoff curve (as we have done in the previous subsections).

All the comparisons should include a description of how the comparisons were produced (i.e., whether data was taken from a paper, a paper was reimplemented, or a paper's code was used) and any differences between or uncertainties about the baseline setting and the setting used for experiments.

6. Guidelines for Achieving Speedup in Practice

In order to speed up deep learning training and correctly interpret one's results, it is important to be aware of the various hardware resources involved in training. Without this knowledge, it is easy to draw incorrect conclusions or accelerate operations that are not actually the bottleneck, which provides no benefit.

As a motivating example, consider Figure 15. The curve shows the accuracy and 90-epoch training times for ResNet-{18,34,50,101} on ImageNet. At first glance, it appears
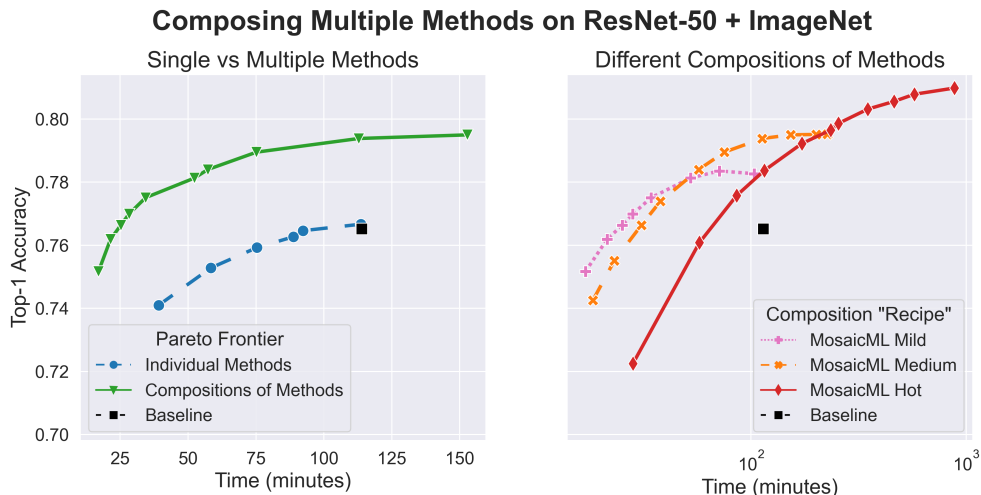
Composing Multiple Methods on ResNet-50 + ImageNet

Figure 14: *(left)* Composing many methods for speeding up training can work far better than using any individual method. *(right)* Different combinations of methods are effective for different amounts of the training budget. No combination is universally the best.

that there is a large accuracy benefit and little speed cost when moving from ResNet-18 to ResNet-34. This could lead one to conclude that, e.g., ResNet-34 is a better architecture than ResNet-18, or that doubling depth is an excellent design choice. However, the ResNet-18 result is handicapped by the data loader. It *appears* not much faster than ResNet-34, but only because the training speed is insensitive to model size in this regime. If we loaded data more quickly, ResNet-18 could be much faster.[3]

Data loading is a common bottleneck in training but by no means the only one. Below are various hardware resources that can become training bottlenecks. This list is not exhaustive—e.g., we omit discussion of caches, register files, execution ports, reorder buffers, read queues, and countless other aspects of various hardware elements. Instead, we focus on a few key resources that all efficiency researchers should be able to reason about for common hardware.

6.1 Possible Bottlenecks and Mitigation

6.1.1 GPU COMPUTE

Overview The term "compute" is often used imprecisely. For our purposes, "compute" refers to operations executed on data, as opposed to the movement of data between devices or levels of the cache hierarchy. Nearly all accelerators have abundant compute capacity. In particular, modern GPUs have much of their area devoted to special hardware for dense matrix multiplications—the most compute-intensive operations in nearly all neural

---

3. This example is not contrived. Because these results use FFCV (Leclerc et al., 2022) on a modern CPU with carefully tuned data loader parameters, it is likely that most reported image classification results with small models are even more bottlenecked by the data loader than this.
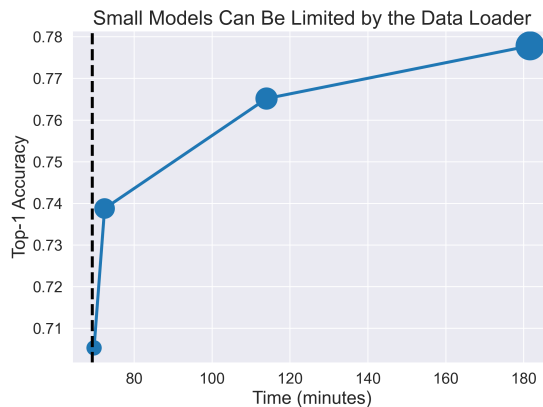
Figure 15: Loading data can often be a bottleneck when dealing with images, especially for small models. Shown are training time and accuracy for ResNet-18, ResNet-34, ResNet-50, and ResNet-101. Marker sizes are proportional to parameter counts. The vertical line indicates the fastest time any model could achieve given the data loader throughput. Failing to control for the data loader bottleneck can make small models appear worse and prevent "speedup" methods from yielding speedups.

networks. As we saw in Figure 2, this makes dense multiply-add operations much cheaper than other computations like normalization, nonlinearities, sparse operations, etc. As a result, compute tends to only be the bottleneck when performing large matrix multiplications or convolutions.

Mitigation The simplest way to eliminate this bottleneck is to reduce the sizes of the tensors involved. This can mean a reduction in feature/channel count or a reduction in resolution/sequence length. One can also impose block sparsity with a large block size, 2:4 sparsity on recent NVIDIA GPUs, or low-rank/structured parameter matrices. See Section 4.1 for specific approaches.

6.1.2 GPU MEMORY

Overview A GPU can only store so much data. During training, memory usage is typically dominated by the optimizer state and activations. The optimizer state is usually a constant factor larger than the model itself, and so dominates when the batch size times input size is small. Here "input size" refers to the resolution, sequence length, or other appropriate measurements. When the batch size times the input size is large, the activation memory typically dominates. E.g., a float32 ResNet-50 model takes around 100MB to store, but its activations can require many gigabytes.

Mitigation The model and optimizer state can be sharded across different accelerators (Rajbhandari et al., 2020) and/or offloaded to CPU RAM (Ren et al., 2021a). One can also use an optimizer with a smaller state (Dettmers et al., 2021; Shazeer and Stern, 2018). Using

a smaller model can also reduce the size of both the model and optimizer state. To reduce activation memory, one can checkpoint/re-materialize activations (ideally intelligently, see Jain et al., 2020), shard saved activations (Rajbhandari et al., 2020), compress activations (Chen et al., 2021c; Evans and Aamodt, 2021), choose activation functions like ReLU that require saving only one bit of state per activation, or use in-place normalization ops (Bulo et al., 2018), among other possibilities (see Sections 4.1.2 and 4.2.2).

6.1.3 GPU MEMORY BANDWIDTH

Overview   Processors, including GPUs, cannot immediately operate on data stored in memory. Instead, values must first be loaded into registers or some other small, fast storage. This loading process takes time. For large blocks of data, such as neural network activations and weights, the loading time is usually limited by the *memory bandwidth*, expressed in bytes per second. A similar cost must also be paid for storing data (i.e., moving it out of fast storage and back into memory).

Mitigation   To reduce memory bandwidth consumption, one must reduce the size of the input and output tensors for one's operations. This means compressing parameters, activations, and optimizer states (see Sections 4.1.2 and 4.2.2). Alternatively, one can *hide* memory accesses by performing more compute per byte. This means *increasing* the sizes of one's inputs and outputs, especially along whichever axis is currently smallest. E.g., if multiplying a 4096×64 and a 64×8192 matrix, increasing the contraction dimension from 64 to 128 would increase the ratio of compute to memory accesses (the *arithmetic intensity*) more than increasing the number of rows or columns. To see this, consider that each element of the left matrix is reused 8192 times and each element of the right matrix is reused 4096 times, while each element of the output matrix has only a 64-element dot product's worth of compute per memory write.

6.1.4 CPU COMPUTE

Overview   For many applications, preprocessing and/or augmenting data happens in the CPU. This is especially likely to be a bottleneck in computer vision applications, where JPEG decoding and complex data augmentation pipelines can be expensive. CPUs may also be taxed by coordinating communication between GPUs, CPU memory, networking cards, and other devices.

Mitigation   If a CPU is bottlenecked by data loading, the surest fixes are to simplify one's data loading pipeline (e.g., reducing the number of augmentations), use a higher-quality data loading library such as NVIDIA DALI or FFCV (Leclerc et al., 2022), or offload some of the computation to accelerators. This offloading is possible with DALI and/or custom code in one's training loop. Separately, this bottleneck can be *hidden* by performing more compute on the GPU per batch loaded, such as by increasing the complexity of the architecture or optimizer (see "retrofit" techniques in Sections 4.1 and 4.3). This can provide a speedup through reduced $n_{\text{iteration}}$.

6.1.5 CPU MEMORY

Overview CPU memory is most often used to store datasets so that samples can be loaded into the GPUs without needing to access storage devices. Without careful implementation, it can also be a limiting factor when saving, loading, or initializing models—for example, multi-GPU training in PyTorch creates one copy of the model per GPU in the machine by default. CPU memory can also be used to offload model parameters, optimizer state, and saved activations (Ren et al., 2021a).

Mitigation To avoid the need to cache the full dataset in RAM, one can ensure that the data is kept in fast storage, preferably a modern NVME solid-state drive. To avoid materializing many copies of a model in memory, one can carefully read the documentation for their deep learning tools (see, e.g., the meta device in PyTorch). To avoid running out of memory when offloading tensors to the CPU, consider offloading fewer or smaller tensors—e.g., offload only the optimizer state but not activations. One may also be able to reduce the size of their dataset using methods from Section 4.2.1, or the size of their model using methods from Section 4.1.2.

6.1.6 CPU MEMORY BANDWIDTH

Overview CPU memory bandwidth is not typically a bottleneck in deep learning training on accelerators. However, if training on a CPU, the considerations for GPU memory bandwidth instead apply to CPU memory bandwidth.

Mitigation If training on a CPU, the mitigations for GPU memory bandwidth apply.

6.1.7 INTER-GPU INTERCONNECT

Overview In a given machine, GPUs are typically connected to each other and the CPU(s) through a motherboard's PCIe slots. The PCIe interface is used when tensors are moved between the CPU(s) and GPU(s) and when GPUs within a machine share gradients with each other. For some GPUs, there are also interfaces like NVLINK that allow faster communication between GPUs.

Mitigation To reduce the communication cost of gradient synchronization, one can use any number of gradient compression schemes—though they may not be worthwhile (Agarwal et al., 2022). One can also synchronize gradients less frequently (Dutta et al., 2018; Ortiz et al., 2021; Stich, 2018; Lin et al., 2018), though with a possible accuracy penalty. As an alternative, one can increase one's ratio of compute to parameter count by using larger batch sizes or input sizes. There are also various parallelism schemes (Shazeer et al., 2017; Rajbhandari et al., 2022; Team and Majumder, 2020; Narayanan et al., 2021), each with their own tradeoffs, that may be beneficial. If one is sharding data across GPUs, sharding less data (e.g., only optimizer states) may be beneficial. See Sections 4.2.2 and 4.3.2 for methods that reduce gradient sizes or communication requirements.

6.1.8 CPU-GPU INTERCONNECT

Overview Supplying data to the GPUs is unlikely to be bottlenecked by PCIe, even for tasks like image classification that often have high ratios of input size to model size. How-

ever, repeated transfer of data between CPU and GPU (most often due to poor data augmentation implementations) can cause this to be a bottleneck. This interconnect can also be a bottleneck when offloading weights, activations or optimizer states to the CPU (Ren et al., 2021a).

Mitigation   One can ensure that their data processing pipeline, debugging statements, and other codes do not move data between the CPU and GPU more than needed. If offloading data to the CPU, one can offload less data (e.g., only optimizer states).

6.1.9 Inter-node interconnect

Overview   When carrying out distributed training, the participating machines must communicate with each other to share information. In the common case of data-parallel training with synchronous gradient-based optimizers, this information consists of gradients for all parameters. Note that it is not necessary to transmit the optimizer state so long as this state is a deterministic function of the gradients. This interconnect can also become a bottleneck when sharding saved activations, model parameters, or optimizer states (Rajbhandari et al., 2020).

Mitigation   The mitigations here are similar to those for Inter-GPU interconnect, though there can be differences when using different parallelism schemes inter-node and intra-node. Also, when training on multiple nodes, one has the option of purchasing better network cards and switches—this is not possible within a node.

6.1.10 Storage capacity

Overview   A storage device, such as a solid-state drive (SSD) or hard drive (HDD), can only store so much data. This can become a limitation when attempting to store large datasets or model checkpoints.

Mitigation   The most straightforward solution is to obtain larger or more storage devices. Alternatively, one can use a data loading library or filesystem that allows streaming in data as needed. One can also attempt to subsample the dataset (see Section 4.2.1). For large model checkpoints, reducing the size of the model (Section 4.1.2) or optimizer state (Section 4.3.2) is helpful.

6.1.11 Storage bandwidth

Overview   An SSD or HDD can only supply data to the CPU so fast. If the storage is too slow, it can cause the data loading to be the bottleneck in the training loop.

Mitigation   One can split data across more local storage devices, preferably in a RAID (Patterson et al., 1988) configuration that allows parallelizing reads across devices. With a sufficiently fast network connection and remote storage, streaming data in as needed can also be effective. If one is willing to alter the semantics of their training run, Data Echoing (Choi et al., 2019), using a larger model, or using more expensive optimization—e.g., Foret et al. (2020); Du et al. (2021); Yao et al. (2021); Zhang et al. (2022); Pauloski et al. (2021); Anil et al. (2020b)—can help conceal this bottleneck: see Sections 4.1.2 and 4.3.

6.1.12 STORAGE NETWORK BANDWIDTH

Overview Similarly, even if a storage device itself is fast, reading data from it can be a bottleneck if the device is connected through a slow network connection (e.g., when using distributed filesystems or remote object stores).

Mitigation The ideal mitigation is to instead store data locally and/or cache it in local RAM. If that isn't possible, the same techniques for alleviating a storage bandwidth bottleneck also apply.

6.2 Overview of Throughputs

In order to reason about the bottlenecks in a given training workload, it is helpful to know roughly how much data each computational resource can move or consume in a given amount of time. We provide the peak throughput of each of the aforementioned resources on representative hardware in Table 4.

Resource	Number per second (trillions)
A100 40GB SXM f16 dense multiply-adds	156 madds
A100 40GB SXM f32 scalar instructions	9.75 FLOPs
Intel Xeon Platinum 8380 CPU f32 dense multiply-adds	2.9 madds
A100 40GB SXM memory bandwidth	1.56 bytes
A100 40GB SXM NVLink bandwidth	0.3 bytes
High-end (800Gbps) inter-node interconnect	0.1 bytes
PCIe v4 bandwidth	0.032 bytes
DDR4-3200 RAM memory bandwidth	0.0256 bytes
Sabrent Rocket 4 Plus SSD read bandwidth	.007 bytes
Low-end (10Gbps) inter-node interconnect	.00125 bytes

Table 4: Computational elements differ in peak throughput by many orders of magnitude. Unless a workload uses these elements in parallel and in proportion to their throughput, some resources will be underutilized while some will be bottlenecks. It is not safe to assume that multiply-adds are the bottleneck.

While the exact numbers will vary based on one's hardware configuration, it is clear that there are extreme differences across computational elements. For example, a single A100 can perform $156/(.00125/2) = 249600$ multiply-adds for every 16-bit value transmitted over a 10 gigabit network connection. Furthermore, when considering entire machines, the differences might be even starker; e.g., with an 8 GPU server, the total multiply-add throughput will increase above 1 quadrillion per second, while machine-level resources like the inter-node interconnect will not increase.

These enormous differences in throughput across devices highlight the importance of incorporating an understanding of hardware into one's application, evaluation, and research of algorithmic speedup methods.

7. Summary and Path Forward

We formalized the training speedup problem and devised a taxonomy through which various speedup approaches can be understood. Using this taxonomy, we surveyed over 200 approaches to accelerating neural network training, classifying methods in terms of fundamental speedup building blocks. Based on observed trends—and experiments with speedup methods and analysis of compute platforms we introduced here—we additionally provided methodological evaluation and practical application guides. Our guidance is enhanced through the connections it makes to the taxonomy and survey, further demonstrating their effectiveness in organizing the literature for the identification of present trends and future opportunities.

Indeed, there are numerous open problems and worthwhile directions for speedup research that we identified. Thus, we conclude with a wide view of the opportunities in this space:

- Speedup methods are often devised for and tested on computer vision problems. Transferring these ideas to large language model (LLM) training, generative modeling, reinforcement learning, adversarial training, and many other problems could be valuable for the community.

- Many subcomponent-action pairs have not been thoroughly explored (see Table 3).

- Existing approaches usually target one or two subcomponents to achieve a speedup. Approaches that target more could be useful.

- Designing new ranking metrics to comprehensively compare speedup methods will aid their evolution. Ranking metrics from the multi-objective community can be the starting place for this research theme.

- A good practice for speedup research involving weight initializations is making pre-trained models publicly available following FAIR principles (i.e., findability, accessibility, interoperability, and reusability). A more systematic effort toward building a comprehensive Model Zoo will avoid duplication of efforts in many cases.

- Some methods have the potential to provide speedups but are hardware-unfriendly, e.g. unstructured sparsity. Algorithm-hardware co-design (i.e., jointly finding an optimal architecture and hardware accelerator) may allow such methods to attain the training efficiency improvements they hint at.

- A crucial building block for speedup research is a benchmarking leaderboard that can evaluate algorithmic speedup methods in a comprehensive, fair, and reliable manner. Such an effort will propel efficient training advances—e.g., related efforts include DawnBench (Coleman et al., 2017) and MLCommons Algorithmic Efficiency.[4]

- Finally, our review strongly suggests that cross-pollination of ideas among diverse research communities—e.g., deep learning, blackbox & multi-objective optimization, approximate computing, and hardware design—is critical to the efficient maturation of a speedup method toolbox.

---

4. See `https://github.com/mlcommons/algorithmic-efficiency` for more information.

Acknowledgments

Appendix A. Motivation for Compute-Efficient Deep Learning

Below we provide a detailed discussion of various limitations that motivate the need for compute-efficient deep learning.

Computing and Hardware Limitations: The doubling of transistor counts every two years predicted by Moore's Law (Moore et al., 1965) suggests one might expect a corresponding doubling in application performance for roughly the same hardware cost. In the "Pre Deep Learning Era" (i.e., before 2010), the amount of computation required to train ML models grew in line with Moore's Law, doubling roughly every 20 months. However, the computational burden of ML training has outpaced Moore's Law since then. In the "Deep Learning Era" (i.e., 2010 to 2015), the scaling of computing power for model training significantly accelerated to doubling approximately every 6 months. In the "Large Model Era" (i.e., after 2015), computing requirements for training massive models really exploded with a 10 to 100-fold increase (Thompson et al., 2020; Schwartz et al., 2020; Jaime Sevilla, 2022). Notably, current large-scale models use 600,000 times more computing power than the noteworthy AlexNet model (Krizhevsky et al., 2012) from 2012 (Lohn and Musser, 2022). By extrapolating the compute costs into the future, Thompson et al. (2021) showed that it will take $10^5\times$ more computing to get a 5 percent error rate on ImageNet. In other words, our current trajectory suggests that progress will be hindered by the pace of hardware advancement.

Economic Cost: Given the fact that compute demands for deep learning are growing significantly faster than the improvements in hardware performance, the training process is becoming more time-consuming and increasingly expensive (Sharir et al., 2020; Thompson et al., 2020; Lohn and Musser, 2022). For example, GPT3 required approximately 3,600 petaFLOPS-days to train. At the advertised maximum performance of a Google TPU v3, it would take approximately $1.65 million to train GPT3. Interestingly, a standard laptop needs about a year to reach one petaFLOPS-day, which implies that it will take several millennia to train GPT-3 (Lohn and Musser, 2022). Similarly, it was estimated to cost around $35 million in computing power to replicate the experiments reported in the AlphaGo Zero paper[5]. By extrapolating the economic costs into the future, Thompson et al. (2020) showed that continued performance improvements in a range of application domains will take billions to trillions of dollars using current training strategies. Similarly, Lohn and Musser (2022) showed that the training cost of the largest AI model in 2026 predicted by their trendline would cost more than the total U.S. GDP, which certainly is infeasible.

---

5. `https://www.yuzeh.com/data/agz-cost.html`

Environmental Cost: Emissions of greenhouse gases such as carbon dioxide or equivalents (CO2eq) due to human activities are the root cause of global warming. Energy production is the major factor in greenhouse gas emissions, contributing around 25% of emissions in 2010 (Tollefson, 2018; Henderson et al., 2020). ML training has the potential to significantly contribute to carbon emissions due to the energy required to power hardware for a long period of time (Dodge et al., 2022). Lacoste et al. (2019) and Henderson et al. (2020) proposed tools to estimate the energy and carbon footprints of ML training. Strubell et al. (2019) showed that the training transformer model (with neural architecture search) could produce an estimated 626,155 lbs of carbon dioxide (CO2) emissions, which is equivalent to the lifetime carbon footprint of five cars (126,000 lbs/car). Thompson et al. (2021) projected that the error level of the best ImageNet classifier would be reduced to just 5% by 2025 but would require emission of as much carbon dioxide as New York City generates in one month. The authors in Patterson et al. (2021, 2022) improved the estimates of energy consumption and CO2 emission in the prior studies. Unfortunately, this revised carbon footprint of ML training is still too large to ignore.

Limited Applications: The compute-intensive nature of deep neural network training severely limits the potential application areas DL can make an impact on. Several application domains where highly accurate predictive models can enable unprecedented success use of resource-limited devices. For example, DL-enhanced ocean monitoring could provide scientists with a powerful tool to enhance the pace of oceanographic research (Ahmad, 2019). Similarly, space exploration using autonomous planetary exploration devices endowed with predictive capabilities offers a huge potential to make unprecedented scientific advances (Diffenderfer et al., 2021). Such resource-limited applications are limitless and can be categorized under the umbrella term Edge-AI (Murshed et al., 2021). Existing Edge-AI relies upon a centralized server for training and updating the ML model, which results in low coverage/accuracy, limited adaptability, and high latency, especially in applications with high spatiotemporal variations. While doing efficient inference at the edge is quite common, training at the edge is still not feasible due to the compute-intensive nature of current training methodologies (Bhardwaj et al., 2022a,b). These limitations are holding us back from exploiting our advanced sensing capabilities to their full potential and, in turn, achieving game-changing societal advances.

Democratization Obstacles: The democratization of AI aims to make it possible to create and use deep learning for everyone. There is a growing consensus among researchers and policymakers to enable techniques so that the benefits of this technology are not limited to a small group of people (Ahmed and Wahed, 2020). Unfortunately, deep learning research is increasingly becoming more computationally intensive, which favors a few resource-rich organizations. This concentration of power can lead to marginalization, causing severe inequalities (Ahmed et al., 2020). Ahmed and Wahed (2020) analyzed 171,394 papers from 57 prestigious computer science conferences and concluded that there is a divergence between the following two groups–large firms and non-elite universities. This rise in divergence is driven by access to computing power, which they termed the "compute divide". The compute-intensive nature of deep learning training presents an obstacle to "democratizing" AI and increases concerns around bias, fairness, and marginalization.

Appendix B. Experimental Details

All models were trained on a single machine with eight A100s and two 32-core AMD EPYC 7513 processors. The only models that were not ResNet-50 instances were those corresponding to the "Smaller ResNets" baseline in Figure 13.

The "Mosaic ResNet" results were taken from the authors' public data. We chose the hyperparameters for speedup methods in Figure 13 based on the hyperparameters used for these recipes. When the hyperparameters did not vary across recipes for a given speedup, we made up similar hyperparameters on a best-effort basis that would allow for assessing alternate speed vs accuracy tradeoffs—e.g., choosing different degrees of progressive resizing. These hyperparameters may not be optimal, so it is important to conclude only that certain baselines *can* outperform these methods, not that they always will. More information about the different sets of methods used for each curve in Figure 14 is shown in Figure 16.

Figure 16: Speedup methods used for different curves in Figure 14. See Blalock et al. (2021) for details about each method.

Microbenchmarking experiments used a single A100, with means and standard deviations computed from five trials. Each trial included 10 executions of the operation in question. All times within a trial were measured after running four "warmup" executions to avoid one-time CUDA and PyTorch overheads. We time only the forward pass for simplicity. All results use half-precision weights and activations.

Appendix C. Additional Experiments

Here, we provide microbenchmarking results for ConvNeXt-B (Liu et al., 2022c) and Swin-B (Liu et al., 2021b) on the CPU to complement our GPU results for these models, shown in Section 2.3. As we found in our experiments on the GPU, the CPU results in Figures 17 and 18 show that FLOP count can be a poor proxy for wall time, and memory bandwidth consumption (while imperfect) can be a more helpful proxy.
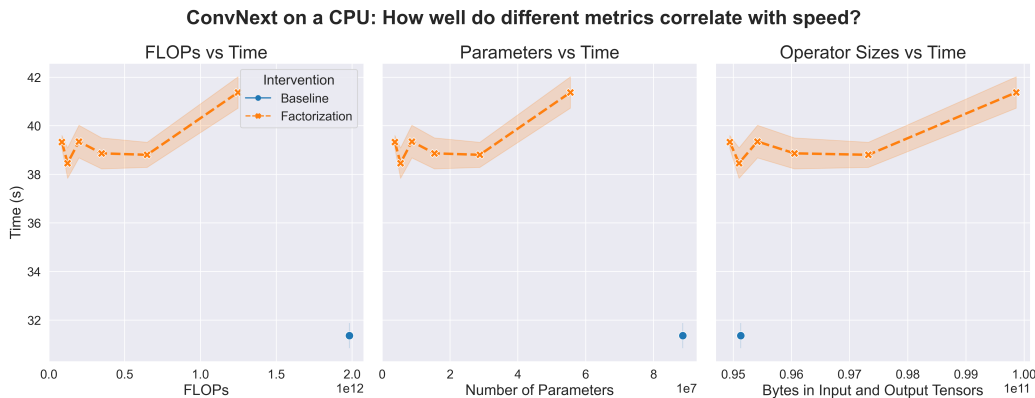
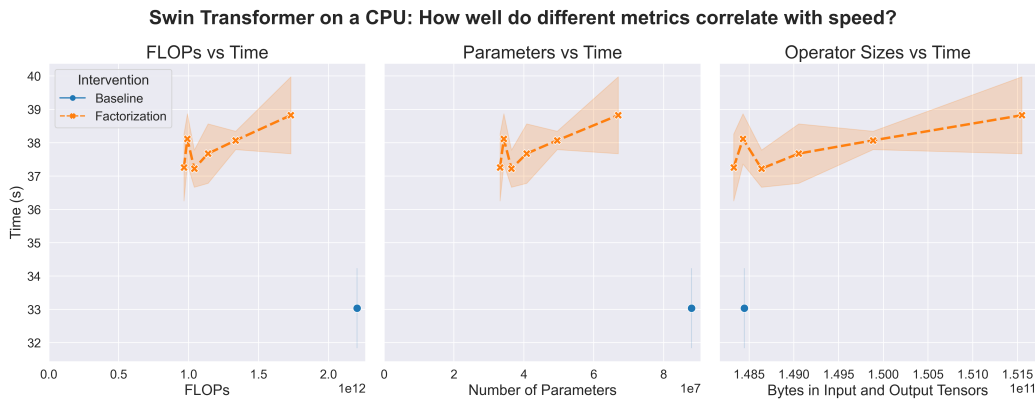Figure 17: Runtime vs various metrics for ConvNeXt-B on 64 CPU cores.

Figure 18: Runtime vs various metrics for Swin-B on 64 CPU cores.

References

Alessandro Achille, Matteo Rovere, and Stefano Soatto. Critical learning periods in deep neural networks. *arXiv preprint arXiv:1711.08856*, 2017.

Menachem Adelman, Kfir Levy, Ido Hakimi, and Mark Silberstein. Faster neural network training with approximate tensor operations. *Advances in Neural Information Processing Systems*, 34:27877–27889, 2021.

Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

Hafez Ahmad. Machine learning applications in oceanography. *Aquatic Research*, 2(3): 161–169, 2019.

Nur Ahmed and Muntasir Wahed. The de-democratization of ai: Deep learning and the compute divide in artificial intelligence research. *arXiv preprint arXiv:2010.15581*, 2020.

Shakkeel Ahmed, Ravi S Mula, and Soma S Dhavala. A framework for democratizing ai. *arXiv preprint arXiv:2001.00818*, 2020.

Ali Akbari, Muhammad Awais, Manijeh Bashar, and Josef Kittler. How does loss function affect generalization performance of deep learning? application to human age estimation. In *International Conference on Machine Learning*, pages 141–151. PMLR, 2021.

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020a.

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Second order optimization made practical. *arXiv preprint arXiv:2002.09018*, 2020b.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Sungyong Baik, Myungsub Choi, Janghoon Choi, Heewon Kim, and Kyoung Mu Lee. Meta-learning with adaptive hyperparameters. *Advances in Neural Information Processing Systems*, 33:20755–20765, 2020.

Atılım Güneş Baydin, Barak A Pearlmutter, Don Syme, Frank Wood, and Philip Torr. Gradients without backpropagation. *arXiv preprint arXiv:2202.08587*, 2022.

Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4161–4168. IEEE, 2021.

Irwan Bello. Lambdanetworks: Modeling long-range interactions without attention. *arXiv preprint arXiv:2102.08602*, 2021.

Irwan Bello, William Fedus, Xianzhi Du, Ekin D Cubuk, Aravind Srinivas, Tsung-Yi Lin, Jonathon Shlens, and Barret Zoph. Revisiting resnets: Improved training and scaling strategies. *arXiv preprint arXiv:2103.07579*, 2021.

Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

Kshitij Bhardwaj, James Diffenderfer, Bhavya Kailkhura, and Maya Gokhale. Benchmarking test-time unsupervised deep neural network adaptation on edge devices. *arXiv preprint arXiv:2203.11295*, 2022a.

Kshitij Bhardwaj, James Diffenderfer, Bhavya Kailkhura, and Maya Gokhale. Unsupervised test-time adaptation of deep neural networks at the edge: a case study. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 412–417. IEEE, 2022b.

Vighnesh Birodkar, Hossein Mobahi, and Samy Bengio. Semantic redundancies in image-classification datasets: The 10% you don't need. *arXiv preprint arXiv:1901.11409*, 2019.

Davis Blalock and John Guttag. Multiplying matrices without multiplying. *arXiv preprint arXiv:2106.10860*, 2021.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2: 129–146, 2020.

Davis W. Blalock, Michael Carbin, Laura Florescu, Jonathan Frankle, Matthew L. Leavitt, Tyler Lee, Moin Nadeem, Jacob Portes, Naveen Rao, Landan Seguin, Cory Stephenson, Hanlin Tang, and Abhinav Venigalla. On evaluating and improving the efficiency of deep networks. Technical report, MosaicML, October 2021. URL https://www.mosaicml.com/blog/methodology.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Freezeout: Accelerate training by progressively freezing layers. *arXiv preprint arXiv:1706.04983*, 2017.

Andrew Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. *arXiv preprint arXiv:2102.06171*, 2021.

Jason Ross Brown, Yiren Zhao, Ilia Shumailov, and Robert D Mullins. Wide attention is the way forward for transformers. *arXiv preprint arXiv:2210.00640*, 2022.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020a.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020b.

Samuel Rota Bulo, Lorenzo Porzi, and Peter Kontschieder. In-place activated batchnorm for memory-optimized training of dnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5639–5647, 2018.

Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. Pixelated butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029*, 2021a.

Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention approximation. *arXiv preprint arXiv:2110.15343*, 2021b.

Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*, pages 1803–1813. PMLR, 2021c.

Tianlong Chen, Xuxi Chen, Xiaolong Ma, Yanzhi Wang, and Zhangyang Wang. Coarsening the granularity: Towards structurally sparse lottery tickets. *arXiv preprint arXiv:2202.04736*, 2022.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015a.

Welin Chen, David Grangier, and Michael Auli. Strategies for training large vocabulary neural language models. *arXiv preprint arXiv:1512.04906*, 2015b.

Xiaohan Chen, Yu Cheng, Shuohang Wang, Zhe Gan, Zhangyang Wang, and Jingjing Liu. Earlybert: Efficient bert training via early-bird lottery tickets. *arXiv preprint arXiv:2101.00063*, 2020.

Xuxi Chen, Tianlong Chen, Yu Cheng, Weizhu Chen, Zhangyang Wang, and Ahmed Hassan Awadallah. Dsee: Dually sparsity-embedded efficient tuning of pre-trained language models. *arXiv preprint arXiv:2111.00160*, 2021d.

Yunpeng Chen, Haoqi Fan, Bing Xu, Zhicheng Yan, Yannis Kalantidis, Marcus Rohrbach, Shuicheng Yan, and Jiashi Feng. Drop an octave: Reducing spatial redundancy in convolutional neural networks with octave convolution. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3435–3444, 2019.

Kashyap Chitta, José M Álvarez, Elmar Haussmann, and Clément Farabet. Training data subset search with ensemble active learning. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

Minsik Cho, Vinod Muthusamy, Brad Nemanich, and Ruchir Puri. Gradzip: Gradient compression using alternating matrix factorization for large-scale deep learning. In *NeurIPS*. 2019.

Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2019.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Sankalan Pal Chowdhury, Adamos Solomou, Avinava Dubey, and Mrinmaya Sachan. On learning the transformer kernel. *arXiv preprint arXiv:2110.08323*, 2021.

Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

Cody Coleman, Christopher Yeh, Stephen Mussmann, Baharan Mirzasoleiman, Peter Bailis, Percy Liang, Jure Leskovec, and Matei Zaharia. Selection via proxy: Efficient data selection for deep learning. *arXiv preprint arXiv:1906.11829*, 2019.

Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.

Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 702–703, 2020.

Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *arXiv preprint arXiv:2106.04803*, 2021.

Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pages 4690–4721. PMLR, 2022a.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*, 2022b.

Mostafa Dehghani, Anurag Arnab, Lucas Beyer, Ashish Vaswani, and Yi Tay. The efficiency misnomer. *arXiv preprint arXiv:2110.12894*, 2021.

Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, et al. Natural neural networks. *Advances in neural information processing systems*, 28, 2015.

Tim Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. *arXiv preprint arXiv:2110.02861*, 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

Sourya Dey, Kuan-Wen Huang, Peter A Beerel, and Keith M Chugg. Pre-defined sparse neural networks with hardware acceleration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):332–345, 2019.

James Diffenderfer, Brian Bartoldson, Shreya Chaganti, Jize Zhang, and Bhavya Kailkhura. A winning hand: Compressing deep networks can improve out-of-distribution robustness. *Advances in Neural Information Processing Systems*, 34:664–676, 2021.

Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. Repvgg: Making vgg-style convnets great again. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13733–13742, 2021.

Jesse Dodge, Taylor Prewitt, Remi Tachet des Combes, Erika Odmark, Roy Schwartz, Emma Strubell, Alexandra Sasha Luccioni, Noah A Smith, Nicole DeCario, and Will Buchanan. Measuring the carbon intensity of ai in cloud instances. In *2022 ACM Conference on Fairness, Accountability, and Transparency*, pages 1877–1894, 2022.

Akshunna S Dogra and William T Redman. Optimizing neural networks via koopman operator theory. *arXiv preprint arXiv:2006.02361*, 2020.

Piotr Dollár, Mannat Singh, and Ross Girshick. Fast and accurate model scaling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 924–932, 2021.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

Jiawei Du, Hanshu Yan, Jiashi Feng, Joey Tianyi Zhou, Liangli Zhen, Rick Siow Mong Goh, and Vincent YF Tan. Efficient sharpness-aware minimization for improved training of neural networks. *arXiv preprint arXiv:2110.03141*, 2021.

Jiawei Du, Daquan Zhou, Jiashi Feng, Vincent YF Tan, and Joey Tianyi Zhou. Sharpness-aware training for free. *arXiv preprint arXiv:2205.14083*, 2022.

Yann Dubois, Benjamin Bloem-Reddy, Karen Ullrich, and Chris J Maddison. Lossy compression for lossless prediction. *arXiv preprint arXiv:2106.10800*, 2021.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. In *International conference on artificial intelligence and statistics*, pages 803–812. PMLR, 2018.

Adam Dziedzic, John Paparrizos, Sanjay Krishnan, Aaron Elmore, and Michael Franklin. Band-limited training and inference for convolutional neural networks. In *International Conference on Machine Learning*, pages 1745–1754. PMLR, 2019.

R David Evans and Tor Aamodt. Ac-gc: Lossy activation compression with guaranteed convergence. *Advances in Neural Information Processing Systems*, 34:27434–27448, 2021.

Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A survey of data augmentation approaches for nlp. *arXiv preprint arXiv:2105.03075*, 2021.

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

Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-aware minimization for efficiently improving generalization. *arXiv preprint arXiv:2010.01412*, 2020.

Stanislav Fort, Andrew Brock, Razvan Pascanu, Soham De, and Samuel L Smith. Drawing multiple augmentation samples per image during training efficiently decreases test error. *arXiv preprint arXiv:2105.13343*, 2021.

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Pruning neural networks at initialization: Why are we missing the mark? *arXiv preprint arXiv:2009.08576*, 2020a.

Jonathan Frankle, David J Schwab, and Ari S Morcos. The early phase of neural network training. *arXiv preprint arXiv:2002.10365*, 2020b.

Dan Fu and Gabriel Guimaraes. Using compression to speed up image classification in artificial neural networks. *Technical report*, 2016.

Jonas Geiping and Tom Goldstein. Cramming: Training a language model on a single gpu in one day. *arXiv preprint arXiv:2212.14034*, 2022.

Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks. *Advances in Neural Information Processing Systems*, 33:2386–2396, 2020.

Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. Efficient training of bert by progressively stacking. In *International conference on machine learning*, pages 2337–2346. PMLR, 2019.

Raphael Gontijo-Lopes, Yann Dauphin, and Ekin D Cubuk. No one representation to rule them all: Overlapping features of training methods. *arXiv preprint arXiv:2110.12899*, 2021.

Santiago Gonzalez and Risto Miikkulainen. Improved training speed, accuracy, and data utilization through loss function optimization. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

Baptiste Goujaud, Damien Scieur, Aymeric Dieuleveut, Adrien Taylor, and Fabian Pedregosa. Super-acceleration with cyclical step-sizes. *arXiv preprint arXiv:2106.09687*, 2021.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Xiaotao Gu, Liyuan Liu, Hongkun Yu, Jing Li, Chen Chen, and Jiawei Han. On the transformer growth for progressive bert training. *arXiv preprint arXiv:2010.12562*, 2020.

Lionel Gueguen, Alex Sergeev, Ben Kadlec, Rosanne Liu, and Jason Yosinski. Faster neural networks straight from jpeg. *Advances in Neural Information Processing Systems*, 31: 3933–3944, 2018.

Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. *IEEE Micro*, 42(4):37–47, 2022.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. In *International Conference on Machine Learning*, pages 2535–2544. PMLR, 2019.

Stewart Hall, Rob Schreiber, and Sean Lie. Training giant neural networks using weight streaming on cerebras wafer-scale systems. Technical report, Cerebras

Systems,Inc., November 2021. URL `https://f.hubspotusercontent30.net/hubfs/8968533/VirtualBoothDocs/CSWeightStreamingWhitePaper111521.pdf`.

Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. Pre-trained models: Past, present and future. *AI Open*, 2:225–250, 2021.

Yuna Han and Byung-Woo Hong. Deep learning based on fourier convolutional neural network incorporating random kernels. *Electronics*, 10(16):2004, 2021.

Boris Hanin and David Rolnick. How to start training: The effect of initialization and architecture. *Advances in Neural Information Processing Systems*, 31, 2018.

Stephen Hanson and Lorien Pratt. Comparing biases for minimal network construction with back-propagation. *Advances in neural information processing systems*, 1, 1988.

Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. Pipetransformer: Automated elastic pipelining for distributed training of large-scale models. In *International Conference on Machine Learning*, pages 4150–4159. PMLR, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning. *Journal of Machine Learning Research*, 21(248):1–43, 2020.

Danny Hernandez and Tom B Brown. Measuring the algorithmic efficiency of neural networks. *arXiv preprint arXiv:2005.04305*, 2020.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

Elad Hoffer, Tal Ben-Nun, Itay Hubara, Niv Giladi, Torsten Hoefler, and Daniel Soudry. Augment your batch: better training with larger batches. *arXiv preprint arXiv:1901.09335*, 2019a.

Elad Hoffer, Berry Weinstein, Itay Hubara, Tal Ben-Nun, Torsten Hoefler, and Daniel Soudry. Mix & match: training convnets with mixed image sizes for improved accuracy, speed and scale resiliency. *arXiv preprint arXiv:1908.08986*, 2019b.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022a.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022b.

Jeremy Howard. Fastai - progressive resizing. Technical report, 2018.

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. *arXiv preprint arXiv:2106.09685*, 2021.

Wei Hu, Lechao Xiao, and Jeffrey Pennington. Provable benefit of orthogonal initialization in optimizing deep linear networks. *arXiv preprint arXiv:2001.05992*, 2020.

Furong Huang, Jordan Ash, John Langford, and Robert Schapire. Learning deep resnet blocks sequentially using boosting theory. In *International Conference on Machine Learning*, pages 2058–2067. PMLR, 2018.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Normalization techniques in training dnns: Methodology, analysis and application. *arXiv preprint arXiv:2009.12836*, 2020.

Nathan Hubens, Matei Mancas, Bernard Gosselin, Marius Preda, and Titus Zaharia. One-cycle pruning: Pruning convnets under a tight training budget. *arXiv preprint arXiv:2107.02086*, 2021.

Yerlan Idelbayev and Miguel A Carreira-Perpinán. Low-rank compression of neural nets: Learning the rank of each layer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8049–8059, 2020.

Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*, 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*, 2018.

et.al. Jaime Sevilla. Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924*, 2022.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

Yunho Jeon and Junmo Kim. Constructing fast network through deconstruction of convolution. *arXiv preprint arXiv:1806.07370*, 2018.

Angela H Jiang, Daniel L-K Wong, Giulio Zhou, David G Andersen, Jeffrey Dean, Gregory R Ganger, Gauri Joshi, Michael Kaminksy, Michael Kozuch, Zachary C Lipton, et al. Accelerating deep learning by focusing on the biggest losers. *arXiv preprint arXiv:1910.00762*, 2019.

Pengzhan Jin, Lu Lu, Yifa Tang, and George Em Karniadakis. Quantifying the generalization error in deep learning in terms of data distribution and neural network smoothness. *Neural Networks*, 130:85–99, 2020.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

Jean Kaddour. Stop wasting my time! saving days of imagenet and bert training with latest weight averaging. *arXiv preprint arXiv:2209.14981*, 2022.

Jean Kaddour, Linqing Liu, Ricardo Silva, and Matt J Kusner. Questions for flat-minima optimization of modern neural networks. *arXiv preprint arXiv:2202.00661*, 2022.

Bhavya Kailkhura, Jayaraman Thiagarajan, Qunwei Li, Jize Zhang, Yi Zhou, and Timo Bremer. A statistical mechanics framework for task-agnostic sample design in machine learning. *Advances in Neural Information Processing Systems*, 33:11925–11935, 2020.

Athresh Karanam, Krishnateja Killamsetty, Harsha Kokel, and Rishabh K Iyer. Orient: Submodular mutual information measures for data subset selection under distribution shift. In *Advances in Neural Information Processing Systems*, 2022.

Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *International conference on machine learning*, pages 2525–2534. PMLR, 2018.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.

Vishal Kaushal, Rishabh Iyer, Suraj Kothawade, Rohan Mahadev, Khoshrav Doctor, and Ganesh Ramakrishnan. Learning from less data: A unified data subset selection and active learning framework for computer vision. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1289–1299. IEEE, 2019.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Krishnateja Killamsetty, Durga Sivasubramanian, Baharan Mirzasoleiman, Ganesh Ramakrishnan, Abir De, and Rishabh Iyer. Grad-match: A gradient matching based data subset selection for efficient learning. *arXiv preprint arXiv:2103.00123*, 2021a.

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, pages 8110–8118, 2021b.

Krishnateja Killamsetty, Xujiang Zhao, Feng Chen, and Rishabh Iyer. Retrieve: Coreset selection for efficient and robust semi-supervised learning. *Advances in Neural Information Processing Systems*, 34:14488–14501, 2021c.

Krishnateja Killamsetty, Guttu Sai Abhishek, Alexandre V Evfimievski, Lucian Popa, Ganesh Ramakrishnan, Rishabh Iyer, et al. Automata: Gradient based data subset selection for compute-efficient hyper-parameter tuning. *arXiv preprint arXiv:2203.08212*, 2022.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Katrin Kirchhoff and Jeff Bilmes. Submodularity for data selection in machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 131–141, 2014.

Boris Knyazev, Michal Drozdzal, Graham W Taylor, and Adriana Romero-Soriano. Parameter prediction for unseen deep architectures. *arXiv preprint arXiv:2110.13100*, 2021.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Jan Kukačka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

Aditya Kusupati, Matthew Wallingford, Vivek Ramanujan, Raghav Somani, Jae Sung Park, Krishna Pillutla, Prateek Jain, Sham Kakade, and Ali Farhadi. Llc: Accurate, multipurpose learnt low-dimensional binary codes. *arXiv preprint arXiv:2106.01487*, 2021.

Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

Nam Le, Honglei Zhang, Francesco Cricri, Ramin Ghaznavi-Youvalari, and Esa Rahtu. Image coding for machines: An end-to-end learned approach. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1590–1594. IEEE, 2021.

Matthew Leavitt. Blazingly fast computer vision training with the mosaic resnet and composer. Technical report, 2022.

Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Madry. ffcv. https://github.com/libffcv/ffcv/, 2022. commit f25386557e213711cc8601833add36ff966b80b2.

James Lee-Thorp, Joshua Ainslie, Ilya Eckstein, and Santiago Ontanon. Fnet: Mixing tokens with fourier transforms. *arXiv preprint arXiv:2105.03824*, 2021.

Changlin Li, Bohan Zhuang, Guangrun Wang, Xiaodan Liang, Xiaojun Chang, and Yi Yang. Automated progressive learning for efficient training of vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12486–12496, 2022a.

Margaret Li, Suchin Gururangan, Tim Dettmers, Mike Lewis, Tim Althoff, Noah A Smith, and Luke Zettlemoyer. Branch-train-merge: Embarrassingly parallel training of expert language models. *arXiv preprint arXiv:2208.03306*, 2022b.

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *arXiv preprint arXiv:2002.11794*, 2020.

Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. *ArXiv, vol. abs/2111.05897*, 2021.

Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. μnas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 70–79, 2021.

Lucas Liebenwein, Alaa Maalouf, Dan Feldman, and Daniela Rus. Compressing neural networks: Towards determining the optimal layer-wise decomposition. *Advances in Neural Information Processing Systems*, 34:5328–5344, 2021.

Tao Lin, Sebastian U Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large minibatches, use local sgd. *arXiv preprint arXiv:1808.07217*, 2018.

Liu Liu, Lei Deng, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, and Yuan Xie. Dynamic sparse graph for efficient deep learning. *arXiv preprint arXiv:1810.00859*, 2018a.

Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*, 2018b.

Shiwei Liu, Tianlong Chen, Zahra Atashgahi, Xiaohan Chen, Ghada Sokar, Elena Mocanu, Mykola Pechenizkiy, Zhangyang Wang, and Decebal Constantin Mocanu. Deep ensembling with no overhead for either training or testing: The all-round blessings of dynamic sparsity. *arXiv preprint arXiv:2106.14568*, 2021a.

Sijia Liu, Pin-Yu Chen, Bhavya Kailkhura, Gaoyuan Zhang, Alfred O Hero III, and Pramod K Varshney. A primer on zeroth-order optimization in signal processing and machine learning: Principals, recent advances, and applications. *IEEE Signal Processing Magazine*, 37(5):43–54, 2020.

Yong Liu, Siqi Mai, Xiangning Chen, Cho-Jui Hsieh, and Yang You. Towards efficient and scalable sharpness-aware minimization. *arXiv preprint arXiv:2203.02714*, 2022a.

Yue Liu, Christos Matsoukas, Fredrik Strand, Hossein Azizpour, and Kevin Smith. Patch-dropout: Economizing vision transformers using patch dropout. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 3953–3962, 2023.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021b.

Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. Swin transformer v2: Scaling up capacity and resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12009–12019, 2022b.

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pages 2736–2744, 2017.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018c.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022c.

Andrew J. Lohn and Micah Musser. Ai and compute, how much longer can computing power drive artificial intelligence progress? 2022.

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

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Yucheng Lu, Conglong Li, Minjia Zhang, Christopher De Sa, and Yuxiong He. Maximizing communication efficiency for large-scale training via 0/1 adam. *arXiv preprint arXiv:2202.06009*, 2022.

Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. *arXiv preprint arXiv:2211.02001*, 2022.

Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Sujay Sanghavi, and Mattan Erez. Prunetrain: fast neural network training by dynamic sparse model reconfiguration. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019.

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

James Martens, Andy Ballard, Guillaume Desjardins, Grzegorz Swirszcz, Valentin Dalibard, Jascha Sohl-Dickstein, and Samuel S Schoenholz. Rapid training of deep neural networks without skip connections or normalization layers using deep kernel shaping. *arXiv preprint arXiv:2110.01765*, 2021.

Dhruv Matani and Suraj Subramanian. Efficient pytorch: Tensor memory format matters. https://pytorch.org/blog/tensor-memory-format-matters/, 2021.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

Gaurav Menghani. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *arXiv preprint arXiv:2106.08962*, 2021.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pages 4556–4565. PMLR, 2019.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

Sören Mindermann, Muhammed Razzak, Winnie Xu, Andreas Kirsch, Mrinank Sharma, Adrien Morisot, Aidan N Gomez, Sebastian Farquhar, Jan Brauner, and Yarin Gal. Prioritized training on points that are learnable, worth learning, and not yet learned. *arXiv preprint arXiv:2206.07137*, 2021.

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

Ashish Mittal, Durga Sivasubramanian, Rishabh Iyer, Preethi Jyothi, and Ganesh Ramakrishnan. Partitioned gradient matching-based data subset selection for compute-efficient robust asr training. *arXiv preprint arXiv:2210.16892*, 2022.

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12, 2018.

Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. Acdc: A structured efficient linear layer. *arXiv preprint arXiv:1511.05946*, 2015.

Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *International workshop on artificial intelligence and statistics*, pages 246–252. PMLR, 2005.

MG Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. *ACM Computing Surveys (CSUR)*, 54(8):1–37, 2021.

Goran Nakerst, John Brennan, and Masudul Haque. Gradient descent with momentum—to accelerate or to super-accelerate? *arXiv preprint arXiv:2001.06472*, 2020.

Sharan Narang, Hyung Won Chung, Yi Tay, William Fedus, Thibault Fevry, Michael Matena, Karishma Malkan, Noah Fiedel, Noam Shazeer, Zhenzhong Lan, et al. Do transformer modifications transfer across implementations and applications? *arXiv preprint arXiv:2102.11972*, 2021.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

Yurii Evgen'evich Nesterov. A method of solving a convex programming problem with convergence rate o(kˆ2). In *Doklady Akademii Nauk*, volume 269, pages 543–547. Russian Academy of Sciences, 1983.

Timothy Nguyen, Zhourong Chen, and Jaehoon Lee. Dataset meta-learning from kernel ridge-regression. *arXiv preprint arXiv:2011.00050*, 2020.

Timothy Nguyen, Roman Novak, Lechao Xiao, and Jaehoon Lee. Dataset distillation with infinitely wide convolutional networks. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

Jose Javier Gonzalez Ortiz, Jonathan Frankle, Mike Rabbat, Ari Morcos, and Nicolas Ballas. Trade-offs of local sgd at scale: An empirical study. *arXiv preprint arXiv:2110.08133*, 2021.

Samet Oymak. Provable super-convergence with a large cyclical learning rate. *IEEE Signal Processing Letters*, 28:1645–1649, 2021.

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Hung Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeffrey Dean. The carbon footprint of machine learning training will plateau, then shrink. 2022.

David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

Mansheej Paul, Surya Ganguli, and Gintare Karolina Dziugaite. Deep learning on a data diet: Finding important examples early in training. *Advances in Neural Information Processing Systems*, 34:20596–20607, 2021.

J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. Kaisa: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

Maxime Pistono, Gouenou Coatrieux, Jean-Claude Nunes, and Michel Cozic. Training machine learning on jpeg compressed images. In *2020 Data Compression Conference (DCC)*, pages 388–388. IEEE, 2020.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

Jacob Portes, Davis Blalock, Cory Stephenson, and Jonathan Frankle. Fast benchmarking of accuracy vs. training time with cyclic learning rates. *arXiv preprint arXiv:2206.00832*, 2022.

Vishak Prasad, Colin White, Paarth Jain, Sibasis Nayak, Rishabh K Iyer, and Ganesh Ramakrishnan. Speeding up nas with adaptive subset selection. In *First Conference on Automated Machine Learning (Late-Breaking Workshop)*, 2022.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*, 2016.

Ofir Press, Noah A Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*, 2021.

Ilan Price and Jared Tanner. Dense for the price of sparse: Improved performance of sparsely initialized networks via a subspace offset. *arXiv preprint arXiv:2102.07655*, 2021.

Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10428–10436, 2020.

Aniruddh Raghu, Maithra Raghu, Simon Kornblith, David Duvenaud, and Geoffrey Hinton. Teaching with commentaries. *arXiv preprint arXiv:2011.03037*, 2020.

Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. *Advances in neural information processing systems*, 30, 2017.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. *arXiv preprint arXiv:2201.05596*, 2022.

Ravi S Raju, Kyle Daruwalla, and Mikko Lipasti. Accelerating deep learning with dynamic data pruning. *arXiv preprint arXiv:2111.12621*, 2021.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021.

Varun Ranganathan and Alex Lewandowski. Zorb: A derivative-free backpropagation algorithm for neural networks. *arXiv preprint arXiv:2011.08895*, 2020.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021a.

Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, 54(4):1–34, 2021b.

Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S Sara Mahdavi, Rapha Gontijo Lopes, et al. Photorealistic text-to-image diffusion models with deep language understanding. *arXiv preprint arXiv:2205.11487*, 2022.

Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6655–6659. IEEE, 2013.

Syed Shakib Sarwar, Priyadarshini Panda, and Kaushik Roy. Gabor filter assisted energy efficient fast learning convolutional neural networks. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2017.

Robin M Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2021.

Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. *Communications of the ACM*, 63(12):54–63, 2020.

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth annual conference of the international speech communication association*. Citeseer, 2014.

Jaime Sevilla and Pablo Villalobos. Review of parameter counts in machine learning. *Published: Alignment Forum (blog)*, 2021.

Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview. *arXiv preprint arXiv:2004.08900*, 2020.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Sheng Shen, Alexei Baevski, Ari S Morcos, Kurt Keutzer, Michael Auli, and Douwe Kiela. Reservoir transformer. *arXiv preprint arXiv:2012.15045*, 2020.

Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

Shoaib Ahmed Siddiqui, Nitarshan Rajkumar, Tegan Maharaj, David Krueger, and Sara Hooker. Metadata archaeology: Unearthing data subsets by leveraging training dynamics. *arXiv preprint arXiv:2209.10015*, 2022.

David Silver, Anirudh Goyal, Ivo Danihelka, Matteo Hessel, and Hado van Hasselt. Learning by directional gradient descent. In *International Conference on Learning Representations*, 2021.

Abhishek Sinha, Mausoom Sarkar, Aahitagni Mukherjee, and Balaji Krishnamurthy. Introspection: Accelerating neural network training by learning weight evolution. *arXiv preprint arXiv:1704.04959*, 2017.

Durga Sivasubramanian, Rishabh Iyer, Ganesh Ramakrishnan, and Abir De. Training data subset selection for regression with controlled generalization error. *arXiv preprint arXiv:2106.12491*, 2021.

Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017.

Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. arxiv e-prints, page. *arXiv preprint arXiv:1708.07120*, 2017.

Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

David R So, Wojciech Manke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V Le. Primer: Searching for efficient transformers for language modeling. *arXiv preprint arXiv:2109.08668*, 2021.

Sanghyun Son, Seungjun Nah, and Kyoung Mu Lee. Clustering convolutional kernels to compress deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 216–232, 2018.

Ben Sorscher, Robert Geirhos, Shashank Shekhar, Surya Ganguli, and Ari S Morcos. Beyond neural scaling laws: beating power law scaling via data pruning. *arXiv preprint arXiv:2206.14486*, 2022.

Aravind Srinivas, Tsung-Yi Lin, Niki Parmar, Jonathon Shlens, Pieter Abbeel, and Ashish Vaswani. Bottleneck transformers for visual recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16519–16529, 2021.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Andreas Steiner, Alexander Kolesnikov, Xiaohua Zhai, Ross Wightman, Jakob Uszkoreit, and Lucas Beyer. How to train your vit? data, augmentation, and regularization in vision transformers. *arXiv preprint arXiv:2106.10270*, 2021.

Cory Stephenson. Colout. Technical report, 2021.

Sebastian U Stich. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

Ilia Sucholutsky and Matthias Schonlau. 'less than one'-shot learning: Learning n classes from m¡ n samples. *arXiv preprint arXiv:2009.08449*, 2020.

Yi-Lin Sung, Varun Nair, and Colin A Raffel. Training neural networks with fixed sparse masks. *Advances in Neural Information Processing Systems*, 34, 2021.

Mingxing Tan and Quoc V Le. Efficientnetv2: Smaller models and faster training. *arXiv preprint arXiv:2104.00298*, 2021.

Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. In *International Conference on Machine Learning*, pages 10118–10129. PMLR, 2021.

Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. Sparse sinkhorn attention. In *International Conference on Machine Learning*, pages 9438–9447. PMLR, 2020a.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020b.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020c.

DeepSpeed Team and Rangan Majumder. DeepSpeed: Extreme-scale model training for everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/, 2020.

The Mosaic ML Team. composer. https://github.com/mosaicml/composer/, 2021.

Kale-ab Tessera, Sara Hooker, and Benjamin Rosman. Keep the gradients flowing: Using gradient flow to study sparse network optimization. *arXiv preprint arXiv:2102.01670*, 2021.

Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 2020.

Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. Deep learning's diminishing returns: the cost of improvement is becoming unsustainable. *IEEE Spectrum*, 58(10):50–55, 2021.

Tijmen Tieleman and Geoffrey Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. *COURSERA Neural Networks Mach. Learn*, 2012.

Nicholas Gerard Timmons and Andrew Rice. Approximating activation functions. *arXiv preprint arXiv:2001.06370*, 2020.

Jeff Tollefson. Ipcc says limiting global warming to 1.5 [degrees] c will require drastic action. *Nature*, 562(7726):172–174, 2018.

Mariya Toneva, Alessandro Sordoni, Remi Tachet des Combes, Adam Trischler, Yoshua Bengio, and Geoffrey J Gordon. An empirical study of example forgetting during deep neural network learning. *arXiv preprint arXiv:1812.05159*, 2018.

Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. *Advances in neural information processing systems*, 32, 2019.

Maxwell Van Gelder, Mitchell Wortsman, and Kiana Ehsani. Deconstructing the structure of sparse neural networks. *arXiv preprint arXiv:2012.00172*, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

Laura Von Rueden, Sebastian Mayer, Katharina Beckh, Bogdan Georgiev, Sven Giesselbach, Raoul Heese, Birgit Kirsch, Julius Pfrommer, Annika Pick, Rajkumar Ramamurthy, et al. Informed machine learning–a taxonomy and survey of integrating knowledge into learning systems. *arXiv preprint arXiv:1903.12394*, 2019.

Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies. In *International Conference on Machine Learning*, pages 3570–3578. PMLR, 2017.

Jun-Kun Wang, Chi-Heng Lin, Andre Wibisono, and Bin Hu. Provable acceleration of heavy ball beyond quadratics for a class of polyak-lojasiewicz functions when the non-convexity is averaged-out. In *International Conference on Machine Learning*, pages 22839–22864. PMLR, 2022.

Rui Wang and Rose Yu. Physics-guided deep learning for dynamical systems: A survey. *arXiv preprint arXiv:2107.01272*, 2021.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A Efros. Dataset distillation. *arXiv preprint arXiv:1811.10959*, 2018.

Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

Kai Wei, Yuzong Liu, Katrin Kirchhoff, and Jeff Bilmes. Using document summarization techniques for speech data subset selection. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 721–726, 2013.

Kai Wei, Yuzong Liu, Katrin Kirchhoff, and Jeff Bilmes. Unsupervised submodular subset selection for speech data. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4107–4111. IEEE, 2014.

Kai Wei, Rishabh Iyer, and Jeff Bilmes. Submodularity in data subset selection and active learning. In *International conference on machine learning*, pages 1954–1963. PMLR, 2015.

Daphna Weinshall, Gad Cohen, and Dan Amir. Curriculum learning by transfer learning: Theory and experiments with deep networks. In *International Conference on Machine Learning*, pages 5238–5246. PMLR, 2018.

Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

Simon Wiesler, Alexander Richard, Ralf Schlüter, and Hermann Ney. Mean-normalized stochastic gradient for large-scale deep learning. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 180–184. IEEE, 2014.

Ross Wightman, Hugo Touvron, and Hervé Jégou. Resnet strikes back: An improved training procedure in timm. *arXiv preprint arXiv:2110.00476*, 2021.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4): 65–76, 2009.

Paul Wimmer, Jens Mehnert, and Alexandru Condurache. Freezenet: Full performance by reduced storage costs. In *Proceedings of the Asian Conference on Computer Vision*, 2020.

Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. Shift: A zero flop, zero parameter alternative to spatial convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9127–9135, 2018a.

Lijun Wu, Fei Tian, Yingce Xia, Yang Fan, Tao Qin, Jianhuang Lai, and Tie-Yan Liu. Learning to teach with dynamic loss functions. *arXiv preprint arXiv:1810.12081*, 2018b.

Xiaoxia Wu, Ethan Dyer, and Behnam Neyshabur. When do curricula work? *arXiv preprint arXiv:2012.03107*, 2020.

Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

Xingyu Xie, Pan Zhou, Huan Li, Zhouchen Lin, and Shuicheng Yan. Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models. *arXiv preprint arXiv:2208.06677*, 2022.

Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, 2020.

Ge Yang, Edward Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tuning large neural networks via zero-shot hyperparameter transfer. *Advances in Neural Information Processing Systems*, 34:17084–17097, 2021.

Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Chris De Sa. Swalp: Stochastic weight averaging in low precision training. In *International Conference on Machine Learning*, pages 7015–7024. PMLR, 2019.

Zichao Yang, Marcin Moczulski, Misha Denil, Nando De Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1476–1483, 2015.

Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. Adahessian: An adaptive second order optimizer for machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 10665–10673, 2021.

Hongwei Yong, Jianqiang Huang, Xiansheng Hua, and Lei Zhang. Gradient centralization: A new optimization technique for deep neural networks. In *European Conference on Computer Vision*, pages 635–652. Springer, 2020.

Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 2019.

Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing gpu energy consumption of dnn training. *arXiv preprint arXiv:2208.06102*, 2022.

Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

Adams Wei Yu, Lei Huang, Qihang Lin, Ruslan Salakhutdinov, and Jaime Carbonell. Block-normalized gradient method: An empirical study for training deep neural network. *arXiv preprint arXiv:1707.04822*, 2017.

Geng Yuan, Xiaolong Ma, Wei Niu, Zhengang Li, Zhenglun Kong, Ning Liu, Yifan Gong, Zheng Zhan, Chaoyang He, Qing Jin, et al. Mest: Accurate and fast memory-economic sparse training framework on the edge. *arXiv preprint arXiv:2110.14032*, 2021.

Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6023–6032, 2019.

X Zhai, A Kolesnikov, N Houlsby, and L Beyer. Scaling vision transformers. arxiv 2021. *arXiv preprint arXiv:2106.04560*, 2021.

Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

Lin Zhang, Shaohuai Shi, Wei Wang, and Bo Li. Scalable k-fac training for deep neural networks with distributed preconditioning. *arXiv preprint arXiv:2206.15143*, 2022.

Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):1943–1955, 2015.

Zhenyu Zhang, Xuxi Chen, Tianlong Chen, and Zhangyang Wang. Efficient lottery ticket finding: Less data is more. In *International Conference on Machine Learning*, pages 12380–12390. PMLR, 2021.

Bo Zhao, Konda Reddy Mopuri, and Hakan Bilen. Dataset condensation with gradient matching. *arXiv preprint arXiv:2006.05929*, 2020.