CUTTLEFISH: LOW-RANK MODEL TRAINING WITHOUT ALL THE TUNING

Hongyi Wang[1] **Saurabh Agarwal**[2] **Pongsakorn U-chupala**[3] **Yoshiki Tanaka**[3]
Eric P. Xing[4 1 5] **Dimitris Papailiopoulos**[6]

ABSTRACT

Recent research has shown that training low-rank neural networks can effectively reduce the total number of trainable parameters without sacrificing predictive accuracy, resulting in end-to-end speedups. However, low-rank model training necessitates adjusting several additional factorization hyperparameters, such as the rank of the factorization at each layer. In this paper, we tackle this challenge by introducing CUTTLEFISH, an automated low-rank training approach that eliminates the need for tuning factorization hyperparameters. CUTTLEFISH leverages the observation that after a few epochs of full-rank training, the *stable rank* (*i.e.,* an approximation of the true rank) of each layer stabilizes at a constant value. CUTTLEFISH switches from full-rank to low-rank training once the stable ranks of all layers have converged, setting the dimension of each factorization to its corresponding stable rank. Our results show that CUTTLEFISH generates models up to 5.6$\times$ smaller than full-rank models, and attains up to a 1.2$\times$ faster end-to-end training process while preserving comparable accuracy. Moreover, CUTTLEFISH outperforms state-of-the-art low-rank model training methods and other prominent baselines. The source code for our implementation can be found at: `https://github.com/hwang595/Cuttlefish`.

1 INTRODUCTION

As neural network-based models have experienced exponential growth in the number of parameters, ranging from 23 million in ResNet-50 (2015) to 175 billion in GPT-3 (2020) and OPT-175B (2022) (Devlin et al., 2018; Brown et al., 2020; Fedus et al., 2021; Zhang et al., 2022), training these models has become increasingly challenging, even with the assistance of state-of-the-art accelerators like GPUs and TPUs. This problem is particularly pronounced in resource-limited settings, such as cross-device federated learning (Kairouz et al., 2019; Wang et al., 2020b; 2021b). In response to this challenge, researchers have explored the reduction of trainable parameters during the early stages of training (Frankle & Carbin, 2018; Waleffe & Rekatsinas, 2020; Khodak et al., 2020; Wang et al., 2021a) as a strategy for speeding up the training process.

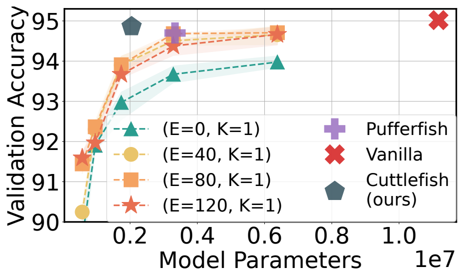
Previous research efforts have focused on developing several approaches to reduce the number of trainable parameters during training. One method involves designing compact neural architectures, such as MobileNets (Howard et al., 2017) and EfficientNets (Tan & Le, 2019), which demand fewer FLOPs. However, this may potentially compromise model performance. Another alternative is weight pruning, which reduces the number of parameters in neural networks (Han et al., 2015a;b; Frankle & Carbin, 2018; Renda et al., 2020; Sreenivasan et al., 2022b). While unstructured sparsity pruning methods can result in low hardware resource utilization, recent advancements have proposed structured pruning based on low-rank weight matrices to tackle this issue (Waleffe & Rekatsinas, 2020; Khodak et al., 2020; Wang et al., 2021a; Hu et al., 2021; Chen et al., 2021b; Vodrahalli et al., 2022). However, training low-rank models necessitates tuning additional hyperparameters for factorization, such as the width/rank of the factorization per layer, in order to achieve both compact model sizes, as measured by the number of parameters, and high accuracy.

Striking the right balance between the size of a low-rank model and its accuracy is crucial, and depends on accurately tuning the rank of the factorized layers. As demonstrated in Figure 1, improper tuning of factorization ranks can lead to either large models or diminished predictive accuracy. Training low-rank networks from scratch may cause significant accuracy loss (Waleffe & Rekatsinas, 2020; Wang et al., 2021a). To address this, previous studies have suggested starting with full-rank model training for a specific number of epochs, E, before transitioning to low-rank model training (Waleffe & Rekatsinas, 2020; Wang et al., 2021a). However, varying the number of full-rank training epochs can influence the final model accuracy, as illustrated in Fig-

[1]Machine Learning Department, Carnegie Mellon University [2]Department of Computer Sciences, University of Wisconsin-Madison [3]Sony Group Corporation [4]Mohamed bin Zayed University of Artificial Intelligence [5]Petuum, Inc. [6]Department of Electrical and Computer Engineering, University of Wisconsin-Madison. Correspondence to: Hongyi Wang <hongyiwa@andrew.cmu.edu>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).

ure 1. Thus, selecting the appropriate number of full-rank training epochs, E, is essential (*e.g.,* neither $E = 0$ nor $E = 120$ yield the optimal model accuracy). Furthermore, to attain satisfactory accuracy, some earlier work (Wang et al., 2021a) has proposed excluding the factorization from the first K layers, resulting in a "hybrid network" that balances model size and accuracy through the choice of K.



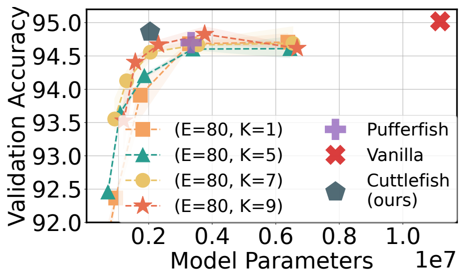


Figure 1. Comparison between CUTTLEFISH and grid search tuning results: (**top**): fixing $K = 1$ (the very first convolution layer is always not factorized) and varying $E \in \{0, 40, 80, 120\}$ and varying the selection of $\mathcal{R}$ by choosing various fixed rank ratios. (**bottom**): fixing a good choice of E, *e.g.,* $E = 80$ and varying K and the rank ratio. The rank ratio varies among $\{\frac{1}{32}, \frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}\}$. Experiments ran on ResNet-18 trained over CIFAR-10.

In this paper, we introduce a novel method for automatically determining the hyperparameters associated with low-rank training, ensuring that the resulting factorized model achieves both a compact size and high final accuracy.

Challenges. We would like to emphasize several reasons why this problem presents considerable challenges. Firstly, the search space $\mathcal{S}$ is vast. For a two hidden layer fully connected (FC) neural network with 100 neurons in each layer (assuming the rank for each layer is 100) and training with 100 epochs, the cardinality of the search space is $|\mathcal{S}| = 100 \times 100 \times 100 \times 2 = 2 \times 10^6$. Furthermore, our objective of automatically optimizing low-rank training factorization hyperparameters while maintaining the advantages of end-to-end training speedups renders traditional neural architecture search (NAS) methods impractical. NAS necessitates concurrent training of both network architecture and network weights, resulting in computational requirements that substantially exceed those of standard model training.

In this work, we present CUTTLEFISH, an automated low-rank factorized training method that eliminates the need for tuning factorization hyperparameters. We observe a key pattern in which the estimated rank of each layer changes rapidly during the initial stages of training and then stabilizes around a constant value (as depicted in Figure 2). We exploit this observation to develop a simple heuristic for selecting the layer ranks $\mathcal{R}$ and the full-rank training duration E: (i) transition from full-rank model training to low-rank model training when all the layer's stable ranks have converged to a constant, and (ii) use these constants as the rank of the factorization.

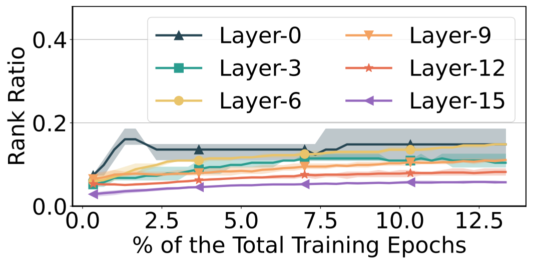


Figure 2. The estimated ranks for various layers in ResNet-18 trained on CIFAR-10, using *stable rank* (which will be discussed in detail later), can be found in our results. The results for other tasks are available in the appendix.

In addition to determining the factorization ranks and full-rank training duration, CUTTLEFISH also addresses the issue of deciding which layers to factorize. For convolutional neural networks (CNNs), CUTTLEFISH observes that factorizing early layers does not lead to considerable speedups, as elaborated in Section 3.5. This observation, along with insights from prior research (Wang et al., 2021a), implies that factorizing early layers may negatively affect the final model accuracy without offering significant performance gains. To tackle this challenge, CUTTLEFISH performs lightweight profiling to identify the layers to factorize, ensuring that factorization occurs only in layers that can effectively enhance the training speed.

Our contributions. We observe a stabilizing effect in the stable ranks of neural network (NN) layers during training, where their stable ranks initially change rapidly and then converge to a constant. Based on this observation, we devise a heuristic to adaptively select the rank of each layer and the duration of full-rank warm-up training. We implement our technique, called CUTTLEFISH, and assess it in large-scale training environments for both language and computer vision tasks. Our comprehensive experimental results demonstrate that CUTTLEFISH automatically selects all factorization hyperparameters during training on-the-fly, eliminating the need for multiple experimental trials for factorization hyperparameter tuning. CUTTLEFISH strikes a balance between model size and final predictive accuracy, excelling in at least one dimension of producing smaller, more accurate models and achieving considerable training speedups compared to state-of-the-art low-rank training, structured pruning, sparse training, quantized training, and learnable factorization methods (Rastegari et al., 2016; Frankle et al., 2019; Wang et al., 2020a; Yu & Huang, 2019; You et al., 2019; Idelbayev & Carreira-Perpinán, 2020; Khodak et al., 2020; Wang et al., 2021a).

1.1 Related work.

Several methods have been developed in the literature to eliminate redundancy in the parameters of modern NNs. Model compression strives to eliminate redundancy in the parameters of trained NNs (Han et al., 2015a). Over time, numerous methods have been devised to remove redundant weights in NNs, encompassing pruning (Li et al., 2016; Wen et al., 2016; Hu et al., 2016; Zhu & Gupta, 2017; He et al., 2017; Yang et al., 2017; Liu et al., 2018; Yu et al., 2018; 2019; Sreenivasan et al., 2022a), quantization (Rastegari et al., 2016; Zhu et al., 2016; Hubara et al., 2016; Wu et al., 2016; Hubara et al., 2017; Zhou et al., 2017), low-rank factorization (Xue et al., 2013; Sainath et al., 2013; Jaderberg et al., 2014; Wiesler et al., 2014; Konečný et al., 2016), and knowledge distillation (Hinton et al., 2015; Yu et al., 2019; Sanh et al., 2019; Jiao et al., 2020; Li et al., 2023).

The *Lottery Ticket Hypothesis* (LTH) suggests that smaller, randomly initialized subnetworks can be trained to attain accuracy levels comparable to those of the full network, although pinpointing these subnetworks can be computationally challenging (Frankle & Carbin, 2018). *Iterative Magnitude Pruning* (IMP) was devised to stabilize LTH while reducing computational costs through warm-up steps (Frankle et al., 2019). Other efforts have sought to eliminate the need for model weight rewinding (Renda et al., 2020) and to identify winning tickets at initialization (Wang et al., 2020a; Sreenivasan et al., 2022b). Moreover, researchers have explored sparsifying NNs during training (Evci et al., 2020). However, these sparsification methods focus on unstructured sparsity, which does not yield actual speedups on current hardware. In contrast, low-rank training can lead to tangible acceleration.

Low-rank factorized training, as well as other structured pruning methods, aim to achieve NNs with structured sparsity during training, allowing for tangible speedups to be obtained (Waleffe & Rekatsinas, 2020; Khodak et al., 2020; You et al., 2020; Wang et al., 2021a; Chen et al., 2021b). Low-rank factorized training has also been employed in federated learning methods to improve communication efficiency and hardware heterogeneity awareness (Hyeon-Woo et al., 2022; Yao et al., 2021). Low-rank factorization techniques have been shown to be combinable and used for training low-rank networks from scratch (Ioannou et al., 2015). However, this method results in a noticeable loss of accuracy, as demonstrated in (Wang et al., 2021a). To tackle this problem, (Khodak et al., 2020) introduces spectral initialization and Frobenius decay, while (Vodrahalli et al., 2022) proposes the *Nonlinear Kernel Projection* method as an alternative to SVD. Low-rank training has also been investigated for fine-tuning large-scale pre-trained models (Hu et al., 2021). These techniques all necessitate additional hyperparameters, which can be tedious to fine-tune. The *LC compression* method attempts to resolve this issue by explicitly learning $\mathcal{R}$ during model training through alternating optimization (Idelbayev & Carreira-Perpinán, 2020). However, this approach is computationally demanding. Our proposed CUTTLEFISH method automatically determines all factorization hyperparameters during training on-the-fly, eliminating the heavy computation overhead and the need for multiple experimental trials for factorization hyperparameter tuning.

Alternative transformations have also been investigated, including Butterfly matrices (Chen et al., 2022), the fusion of low-rank factorization and sparsification (Chen et al., 2021a), and block-diagonal matrices (Dao et al., 2022). Furthermore, novel architectures have been developed for enhanced training or inference efficiency, such as SqueezeNet (Iandola et al., 2016), Eyeriss (Chen et al., 2016), ShuffleNet (Zhang et al., 2018), EfficientNet (Tan & Le, 2019), MobileNets (Howard et al., 2017), Xception (Chollet, 2017), ALBERT (Lan et al., 2019), and Reformer (Kitaev et al., 2020).

2 PRELIMINARY

In this section, we present an overview of the core concepts of low-rank factorization for various NN layers, along with a selection of specialized training methods specifically designed for low-rank factorized training.

2.1 Low-rank factorization of NN layers

FC/MLP Mixer layer. A 2-layer fully connected (FC) neural network can be represented as $f(x) = \sigma(\sigma(x\mathbf{W}_1)\mathbf{W}_2)$, where $\mathbf{W}$s are weight matrices, $\sigma(\cdot)$ is an arbitrary activation function, and x is the input data point. The weight matrix $\mathbf{W}$ can be factorized as $\mathbf{U}\mathbf{V}^\top$. A similar approach can be applied to ResMLP/MLP mixer layers, where each learnable weight can be factorized in the same manner (Touvron et al., 2021a; Tolstikhin et al., 2021).

Convolution layer. For a convolutional layer with dimensions (m, n, k, k), where m and n are the number of input and output channels and k represents the size of a convolution filter, a common approach involves factorizing the unrolled 2D matrix. We will discuss a popular method for factorizing a convolutional layer. Initially, the 4D tensor $\mathbf{W}$ is unrolled to obtain a 2D matrix of shape (mk^2, n), where each column represents the weight of a vectorized convolution filter. The rank of the unrolled matrix is determined by $\min\{mk^2, n\}$. Factorizing the unrolled matrix results in $\mathbf{U} \in \mathbb{R}^{mk^2 \times r}$ and $\mathbf{V}^\top \in \mathbb{R}^{r \times n}$. Reshaping the factorized $\mathbf{U}, \mathbf{V}^\top$ matrices back to 4D yields $\mathbf{U} \in \mathbb{R}^{m \times r \times k \times k}$ and $\mathbf{V}^\top \in \mathbb{R}^{r \times n}$. Consequently, factorizing a convolutional layer produces a thinner convolutional layer $\mathbf{U}$ with r convolution filters and a linear projection layer $\mathbf{V}^\top$. The $\mathbf{V}^\top$s can also be represented by a 1×1 convolutional layer, such as

$\mathbf{V}^\top \in \mathbb{R}^{r\times n\times 1\times 1}$, which is more suited for computer vision tasks since it operates directly in the spatial domain (Lin et al., 2013; Wang et al., 2021a).

Multi-head attention (MHA) layer. A p-head attention layer learns p attention mechanisms on the key, value, and query ($\mathbf{K}, \mathbf{V}, \mathbf{Q}$) of each input token:

$$\texttt{MHA}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \texttt{Concat}(\text{head}_1,\dots,\text{head}_p)\mathbf{W}^O.$$

Each head performs the computation of:

$$\begin{aligned}\text{head}_i &= \texttt{Attention}(\mathbf{QW}_Q^{(i)},\mathbf{KW}_K^{(i)},\mathbf{VW}_V^{(i)})\\ &= \texttt{softmax}\left(\frac{\mathbf{QW}_Q^{(i)}\mathbf{W}_K^{(i)\top}\mathbf{K}^\top}{\sqrt{d/p}}\right)\mathbf{VW}_V^{(i)}.\end{aligned}$$

where d is the hidden dimension. The trainable weights $\mathbf{W}_Q^{(i)}, \mathbf{W}_K^{(i)}, \mathbf{W}_V^{(i)}, i \in \{1,2,\dots,p\}$ can be factorized by simply decomposing all learnable weights $\mathbf{W}\cdot$ in an attention layer and obtaining $\mathbf{U}\cdot\mathbf{V}^\top\cdot$ (Vaswani et al., 2017).

2.2 Training methods for low-rank networks

Hybrid NN architecture. It has been noted that factorizing the initial layers may negatively impact a model's accuracy (Konečnỳ et al., 2016; Waleffe & Rekatsinas, 2020; Wang et al., 2021a). One possible explanation is that NN layers can be viewed as feature extractors, and poor features extracted by the early layers can accumulate and propagate throughout the NN. To address this issue, the *hybrid NN architecture* was proposed, which only factorizes the lower layers while keeping the initial layers full-rank (Wang et al., 2021a). The weights of a full-rank L-layer NN can be represented as $\mathcal{W} = \{\mathbf{W}_i | 1 \le i \le L\}$. The corresponding hybrid model's weights can be represented as $\mathcal{H} = \{\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K, \mathbf{U}_{K+1}, \mathbf{V}_{K+1}^\top, \dots, \mathbf{U}_{L-1}, \mathbf{V}_{L-1}^\top, \mathbf{W}_L\}$, where K is the number of layers that are not factorized and is treated as a hyperparameter to be tuned manually (Wang et al., 2021a). It is important to note that the last classification layer, *i.e.,* $\mathbf{W}_L$, is usually not factorized (Khodak et al., 2020; Wang et al., 2021a).

Full-rank to low-rank training. Training low-rank factorized models from scratch often results in a decrease in accuracy (Waleffe & Rekatsinas, 2020; Khodak et al., 2020; Wang et al., 2021a). To mitigate this drop, it is common to train the full-rank model for E epochs before factorizing it (Waleffe & Rekatsinas, 2020; Khodak et al., 2020; Wang et al., 2021a). However, determining the appropriate number of full-rank training epochs is treated as a hyperparameter and typically tuned manually in experiments (Wang et al., 2021a). Observations indicate that finding the right number of full-rank training epochs is crucial for achieving optimal final model accuracy in low-rank factorized NNs.

Initialization and weight decay. Factorized low-rank networks can benefit from tailored initialization methods (Ioannou et al., 2015; Khodak et al., 2020). One such method, called *spectral initialization*, aims to approximate the behavior of existing initialization methods (Khodak et al., 2020). Spectral initialization represents a special case of transitioning from full-rank to low-rank training with $E = 0$. Additionally, specific regularization techniques have been proposed to enhance the accuracy of low-rank networks. For example, *Frobenius decay* applies weight decay on $\|\mathbf{UV}^\top\|_F^2$ instead of $\|\mathbf{U}\|_F^2 + \|\mathbf{V}^\top\|_F^2$ during factorized low-rank training.

3 CUTTLEFISH: AUTOMATED LOW-RANK FACTORIZED TRAINING

In this section, we outline the problem formulation of CUTTLEFISH, elaborate on each factorization hyperparameter, and describe CUTTLEFISH's heuristics for determining all of these factorization hyperparameters.

3.1 Problem formulation.

The search space for adaptive factorized tuning is defined by three sets of hyperparameters, namely $\mathcal{S} = (E, K, \mathcal{R})$ (full-rank training epochs, the number of initial layers that remain unfactorized, and layer factorization ranks). The objective of CUTTLEFISH is to find an optimal $\hat{s} \in \mathcal{S}$ on-the-fly, with minimal computational overhead during training, such that the resulting low-rank factorized models are both compact and maintain high accuracy, comparable to their full-rank counterparts.

3.2 Components in the search space and the trade-offs among hyperparameter selections.

Full-rank training epochs E. The value of E can range from 0 to $T-1$. Neither too small (*e.g.,* $E = 0$) nor too large (*e.g.,* $E = 120$) values of E result in the best accuracy (Figure 1), highlighting the necessity of tuning E.

The number of full-rank layers K. As previously mentioned, the weights of a hybrid NN architecture can be represented by $\mathcal{H} = \{\mathbf{W}_1, \dots, \mathbf{W}_K, \mathbf{U}_{K+1}, \mathbf{V}_{K+1}^\top, \dots, \mathbf{U}_{L-1}, \mathbf{V}_{L-1}^\top, \mathbf{W}_L\}$, where K is a hyperparameter to be tuned. The selection of K can range from 1 to $L-1$, meaning the very first and very last layers are always not factorized. Factorizing additional layers results in increased accuracy loss but also reduces the model size and computational complexity. Thus, an optimal choice for K should balance the trade-off between accuracy loss and model compression rate.

Rank selections for factorized layers $\mathcal{R}$. $\mathcal{R}$ specifies the ranks used when factorizing the $(L-K-1)$ layers in a hybrid NN architecture, *i.e.,* $\mathcal{R} = \{r_i | K+1 \le i \le L-1\}$. For a layer weight $\mathbf{W}_i \in \mathbb{R}^{m\times n}$, the full rank of the layer is $\text{rank}(\mathbf{W}_i) = \min\{m,n\}$. Thus, $1 \le r_i \le \text{rank}(\mathbf{W}_i), \forall i \in \{K+1,\dots,L-1\}$. Using a too small r for factorizing a layer may result in a decrease in accuracy. However, employing a relatively large r to factorize the layer could negatively impact the model compression rate.

In this paper, we develop heuristics that automatically identify optimal choices for each component within the search space, *i.e.,* $\hat{s} \in \mathcal{S}$, in order to strike a balance between the final accuracy and model size.

Why is finding an appropriate $s \in \mathcal{S}$ challenging? Firstly, the search space's cardinality, $|\mathcal{S}|$, is vast. Furthermore, the primary goal of low-rank factorized training is to accelerate model training. Therefore, it is crucial to identify $\hat{s}$ without introducing significant computational overhead. While Neural Architecture Search (NAS) style methods could potentially be employed to search for $s \in \mathcal{S}$, they result in high computational overhead. Consequently, adopting NAS-based algorithms is not suitable for our scenario, as our objective is to achieve faster training.

3.3 Determining factorization ranks ($\mathcal{R}$) for NN layers.

In previous work, ranks of the factorized layers have typically been treated as a hyperparameter, with a fixed global rank ratio ρ often employed, for example, $\mathcal{R} = \{\rho \cdot \text{rank}(\mathbf{W}_i) | 1 \leq i \leq L\}$ (Khodak et al., 2020; Wang et al., 2021a). However, a crucial question to consider is *do all layers converge to the same ρ during training?*

Rank estimation metric. One might wonder why we cannot simply use the normal rank for estimating the layer ranks of NNs. The answer is that the normal rank is always full for layer weights of NNs. However, NN weight matrices are "nearly" low-rank when they exhibit a rapid spectral decay. Therefore, we require a metric to estimate layer ranks. In CUTTLEFISH, we utilize the *stable rank*, which serves as a valuable proxy for the actual rank since it remains largely unaffected by small singular values, to estimate the rank of model layer weights $\mathbf{W}$. The definition of stable rank is $\texttt{stable rank}(\mathbf{\Sigma}) = \frac{\mathbf{1}^\top \mathbf{\Sigma}^2 \mathbf{1}}{\sigma^2_{\max}(\mathbf{W})}$, where $\mathbf{1}$, $\sigma^2_{\max}(\cdot)$, and Σ represent the identity column vector, the maximum squared singular value, and the diagonal matrix that stores all singular values in descending order, *i.e.,* $\mathbf{1}^\top \mathbf{\Sigma} = \left[\sigma_1, \ldots, \sigma_{\text{rank}(\mathbf{W})}\right]$, respectively. Another advantage of using stable rank is that its calculation does not require specifying any additional hyperparameters.

The scaled stable rank. Stable rank, which disregards minuscule singular values, often results in very low rank estimations. This can work well for relatively small tasks, such as CIFAR-10. However, for larger scale tasks like ImageNet, using stable rank directly leads to a non-trivial accuracy drop of 2.3% (details can be found in the appendix). To address this issue, we propose using *scaled stable rank*. Scaled stable rank assumes that the estimated rank of a randomly initialized matrix, *i.e.,* $\mathbf{W}^0$ (model weight at the 0-th epoch), should be close or equal to full rank. Nevertheless, based on our experimental observations, stable rank estimation of randomly initialized weights tends not to be full rank. Therefore, we store the ratio of full rank to initial stable rank (denoted as ξ, *e.g.,* if $\text{rank}(\mathbf{W}) = 512$ and $\texttt{stable rank}(\mathbf{\Sigma}^0) = 200$, then $\xi = 512/200$). We scale each epoch's stable rank by:

$$\texttt{scaled stable rank}(\mathbf{\Sigma}, \xi) = \xi \cdot \texttt{stable rank}(\mathbf{\Sigma});$$
$$\xi = \frac{\text{rank}(\mathbf{W}^0)}{\texttt{stable rank}(\mathbf{\Sigma}^0)}, \forall t \in \{1, 2, \ldots, T\}.$$

CUTTLEFISH rank selection. We observe that different layers tend to converge to varying stable ranks (an example is shown in Figure 3, with similar trends found in other tasks, as detailed in the appendix). Middle layers generally converge to larger ρs, indicating greater redundancy. As a result, it is unlikely that a fixed rank ratio is optimal, as it may either fail to eliminate all redundancy in the layer weights or be too aggressive in compressing model weights, thereby compromising final accuracy. CUTTLEFISH employs the scaled stable rank at epoch E (*i.e.,* the transition point from full-rank to low-rank) to factorize the full-rank model and obtain a low-rank factorized model.

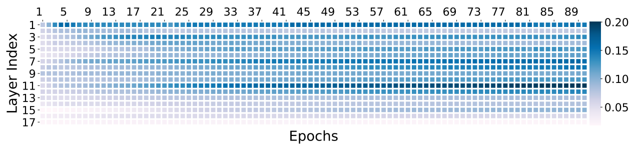


Figure 3. The rank ratios (ρs) of stable ranks for ResNet-18 trained on CIFAR-10, where darker colors indicate higher ρ values (results for other datasets can be found in the appendix).

3.4 Determining full-rank training epochs E

As discussed in Section 1, neither too small nor too large E values result in optimal accuracy. Furthermore, larger E values also lead to slower training time, as full-rank models have higher computational complexity. CUTTLEFISH is inspired by the observation that the estimated ranks for all NN layers, $\mathcal{R}$, change rapidly during the early training phase but stabilize in later training epochs. A reasonable heuristic, therefore, is to switch from full-rank training to low-rank training when the estimated ranks no longer vary significantly. The question now is, *how can we determine if the curves of the estimated ranks have stabilized?* CUTTLEFISH tracks the sequences of stable ranks for each layer at each epoch, *i.e.,* $\varrho = \{r^0, r^1, \ldots, r^t\}$. CUTTLEFISH measures the derivative of the estimated rank sequences for all layer weights ($\frac{\mathrm{d}\varrho_l}{\mathrm{d}t}$) to detect when they cease to change significantly, using a condition: $\frac{\mathrm{d}\varrho_l}{\mathrm{d}t} \leq \epsilon, \forall l \in \{K+1, \ldots, L-1\}$, where ϵ is a close-to-zero rank stabilization threshold.

3.5 Determining K for hybrid architectures

K balances the final accuracy and model compression rate. However, discerning the relationship between K and final accuracy without fully training the model to convergence is challenging and impractical for achieving faster training speeds. For each task, CUTTLEFISH conducts lightweight profiling to measure the runtime of the low-rank NN when factorizing each layer stack, as layers within the same stack have identical weights and input sizes, and assesses whether it results in a significant speedup. CUTTLEFISH only performs factorization (with profiling rank ratio candidates: $\bar{\rho}$) if it leads to meaningful acceleration (determined by a threshold υ). For example, if full-rank time $>$ 1.5 $\times$ factorized time for $\upsilon = 1.5$ when

$\bar{\rho} = \frac{1}{4}$, then CUTTLEFISH proceeds with factorization. Figure 4 illustrates one example benchmark, where factorizing the first convolution stack (*i.e.,* layer 2 to layer 5) does not yield a substantial speedup. Consequently, CUTTLEFISH does not factorize these layers and returns $\hat{K} = 6$.

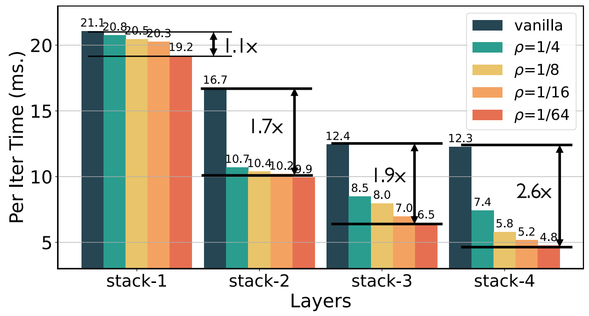


Figure 4. The per-iteration forward time in milliseconds, benchmarked using ResNet-18 on CIFAR-10 with a batch size of 1,024 on an EC2 p3.2xlarge instance.

Why does not factorizing the initial layers result in a significant speedup? The reason for this can be attributed to the concept of *arithmetic intensity*, which is defined as the ratio of FLOPS to the bytes of data that must be accessed for a specific computation (Jeffers et al., 2016). When a certain layer has low arithmetic intensity, the GPU cannot operate at peak performance, and therefore, even if the FLOPs are substantially reduced, the actual speedup will not be significant. For a convolution layer, its arithmetic intensity is proportional to $\mathcal{O}(\frac{Bmnk^2HW}{mnk^2+BmHW})$ where B, H, W stand for the batch size, height, and width of the input image. In convolution networks, it is generally assumed that the initial layers have small mn but large HW, leading to $BmHW \gg mnk^2$ and $\mathcal{O}(\frac{Bmnk^2HW}{mnk^2+BmHW}) \to \mathcal{O}(nk^2)$. For later layers, where H, W are small and mnk^2 are large, and thus $mnk^2 \gg BmHW$, $\mathcal{O}(\frac{Bmnk^2HW}{mnk^2+BmHW}) \to \mathcal{O}(BHW)$. In the example shown in Figure 4, $nk^2 = 64 \times 9 = 576$ for the first layer stack, while for the last layer stack, $BHW = 1024 \times 8 \times 8 = 65,536 \gg 576$. Consequently, the bottom layers exhibit much higher arithmetic intensity, and as a result, factorization leads to significant speed improvements. For Transformers, where each layer has identical weight and input sizes (*i.e.,* the same arithmetic intensity), we consistently factorize all Transformer layers except for the word/image sequence embedding layers.

3.6 Putting things together

The main algorithm of CUTTLEFISH is outlined in Algorithm 1. CUTTLEFISH begins with profiling to determine $\hat{K}$. Following this, the training method commences with full-rank training until the stable ranks for the layers to be factorized converge, *i.e.,* at epoch $\hat{E}$. Subsequently, CUTTLEFISH factorizes the partially trained full-rank network using the converged scaled stable ranks $\mathcal{R}$ to obtain the factorized low-rank model. Finally, the low-rank model is trained until it reaches full convergence. In our experiments, we set $\epsilon = 0.1$ and $\upsilon = 1.5$.

Algorithm 1 CUTTLEFISH

```
Input: The dataset $\mathcal{D}$, initial full-rank neural network weights
    $\mathcal{W}^0 = \{\mathbf{W}_1^0, \dots, \mathbf{W}_L^0\}$, the training algorithm $A(\cdot)$, such
    as SGD, Adam, etc., the total number of epochs $T$, and a
    rank stabilization threshold $\epsilon$.
Output: The trained low-rank factorized NN.
Initialize the following: $\hat{E} = T$; $\mathcal{H} = \{\}$; $\varrho_{K+1}, \dots, \varrho_{L-1} = \{\}, \dots, \{\}$; $\hat{K}$ = Profiling($\mathcal{D}, \mathcal{W}, \tau, \bar{\rho}$) (Algorithm 2), $\xi_l$ = rank($\mathbf{W}_l^0$)/stable rank($\Sigma_l^0$), $\forall l \in \{1, \dots, L\}$.
for $t \in \{0, 1, 2, \dots, T-1\}$ do
    if $t \leq \hat{E}$ then
        $\mathcal{W}^{t+1} \leftarrow A(\mathcal{W}^t, \mathcal{D})$
        for $\mathbf{W}_l \in \mathcal{W}^t$ do
            if $K + 1 \leq l < L$ then
                $\tilde{\mathbf{U}}_l \mathbf{\Sigma}_l \tilde{\mathbf{V}}_l^\top$ = SVD($\mathbf{W}_l$);
                $r_l$ = stable rank($\mathbf{\Sigma}_l$), $\varrho_l = \varrho_l \cup \{r_l\}$;
        end
        if $\frac{\mathrm{d}\varrho_l}{\mathrm{d}x} \leq \epsilon, \forall l \in \{K+1, \dots, L-1\}$ then
            $\hat{E} = t + 1$;
    else if $t = \hat{E} + 1$ then
        for $\mathbf{W}_l \in \mathcal{W}^t$ do
            if $K + 1 \leq l < L$ then
                $\tilde{\mathbf{U}}_l \mathbf{\Sigma}_l \tilde{\mathbf{V}}_l^\top$ = SVD($\mathbf{W}_l$);
                $r_l$ = scaled stable rank($\mathbf{\Sigma}_l, \xi_l$);
                $\mathbf{U}_l = \tilde{\mathbf{U}}_l \mathbf{\Sigma}_l^{\frac{1}{2}}, \mathbf{V}_l^\top = \mathbf{\Sigma}_l^{\frac{1}{2}} \tilde{\mathbf{V}}_l^\top$;
                $\mathcal{H} = \mathcal{H} \cup \{\mathbf{U}_l[:, 1:r_l], \mathbf{V}_l^\top[1:r_l, :]\}$ (with
                necessary NN weights reshaping);
            else
                $\mathcal{H} = \mathcal{H} \cup \{W_l\}$;
            end
        end
        $\mathcal{H}^t = \mathcal{H}$; $\mathcal{H}^{t+1} \leftarrow A(\mathcal{H}^t, \mathcal{D})$;
    else
        $\mathcal{H}^{t+1} \leftarrow A(\mathcal{H}^t, \mathcal{D})$;
    end
end
```

Algorithm 2 Profiling

```
Input: The dataset $\mathcal{D}$, full-rank model weights $\mathcal{W}$, the profiling
    iterations $\tau$, and a profiling rank ratio candidates: $\bar{\rho}$.
Output: Determined $\hat{K}$.
init_timer()
for layer range ($l_{beg}$, $l_{end}$) $\in$ layer stacks do
    $\mathcal{H}$ = factorize_layer_stack($\mathcal{W}, \bar{\rho}, l_{beg}, l_{end}$);
    start_time = timer.tic();
    for iter $\in \{1, 2, \dots, \tau\}$ do
        Train $\mathcal{H}$ for one iteration
    end
    end_time = timer.toc();
    avg_low-rank_time = (end_time-start_time)/$\tau$;
    start_time = timer.tic();
    for iter $\in \{1, 2, \dots, \tau\}$ do
        Train $\mathcal{W}$ for one iteration;
    end
    end_time = timer.toc();
    avg_fullrank_time = (end_time-start_time)/$\tau$;
    if fullrank_time $> \upsilon \cdot$ avg_low-rank_time then
        $\hat{K} = l_{end}$
end
```

4 EXPERIMENTS

We have developed an efficient implementation of CUTTLEFISH and conducted extensive experiments to evaluate its performance across various vision and natural language processing tasks. Our study focuses on the following aspects: (i) the sizes of factorized models CUTTLEFISH discovers and their final accuracy; (ii) the end-to-end training speedups that CUTTLEFISH achieves in comparison to full-rank training and other baseline methods; (iii) how the $\hat{s}$s found by CUTTLEFISH compare to manually tuned and explicitly learned ones. Our comprehensive experimental results demonstrate that CUTTLEFISH automatically selects all factorization hyperparameters during training on-the-fly, eliminating the need for multiple experimental trials for factorization hyperparameter tuning. More specifically, the experimental results reveal that CUTTLEFISH generates models up to 5.6× smaller than full-rank models, and attains up to a 1.2× faster end-to-end training process while preserving comparable accuracy. Moreover, CUTTLEFISH outperforms state-of-the-art low-rank model training methods and other prominent baselines.

4.1 Experimental setup and implementation details

Pre-training ML tasks. We conducted experiments on various computer vision pre-training tasks, including CIFAR-10, CIFAR-100 (Krizhevsky et al., 2009), SVHN (Netzer et al., 2011), and ImageNet (ILSVRC2012) (Deng et al., 2009). For CIFAR-10, CIFAR-100, and SVHN (Netzer et al., 2011), we trained VGG-19-BN (referred to as VGG-19) (Simonyan & Zisserman, 2014) and ResNet-18 (He et al., 2016). In the case of the SVHN dataset, we utilized the original training images and excluded the additional images. For ImageNet, our experiments involved ResNet-50, WideResNet-50-2 (referred to as WideResNet-50), DeiT-base, and ResMLP-S36 (He et al., 2016; Zagoruyko & Komodakis, 2016; Touvron et al., 2021b;a). Further details about the machine learning tasks can be found in the appendix.

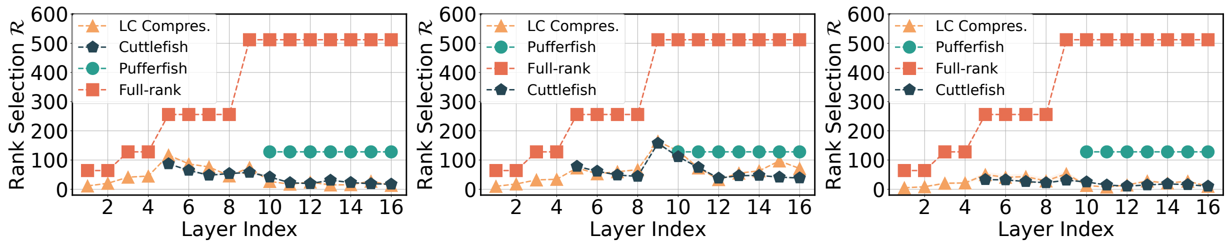
Fine-tuning ML tasks. We experiment on fine-tuning $\text{BERT}_{\text{BASE}}$ over the GLUE benchmark (Wang et al., 2018).

Hyperparameters & training schedule. For VGG-19 and ResNet-18 training on CIFAR-10 and CIFAR-100, we train the NNs for 300 epochs, while on SVHN, we train the NNs for 200 epochs. We employ a batch size of 1,024 for CIFAR and SVHN tasks to achieve high arithmetic intensity. The initial learning rate is linearly scaled up from 0.1 to 0.8 within five epochs and then decayed at milestones of 50% and 75% of the total training epochs (Goyal et al., 2017). For WideResNet-50 and ResNet-50 training on the ImageNet dataset, we follow the hyperparameter settings in (Goyal et al., 2017), where the models are trained for 90 epochs with an initial learning rate of 0.1, which is decayed by a factor of 0.1 at epochs 30, 60, and 80 using a batch size of 256. In addition to (Goyal et al., 2017), we apply label smoothing as described in (Wang et al., 2021a). For DeiT and ResMLP, we train them from scratch, adhering to the training schedule proposed in (Touvron et al., 2021b). For the GLUE fine-tuning benchmark, we follow the default hyperparameter setup in (Devlin et al., 2018; Jiao et al., 2020). Since CUTTLEFISH generalizes spectral initialization (SI) and is compatible with Frobenius decay (FD), we deploy FD in conjunction with CUTTLEFISH when it contributes to better accuracy. Further details on hyperparameters and training schedules can be found in the appendix.

Experimental environment. We employ the NVIDIA NGC Docker container for software dependencies. Experiments for CIFAR, SVHN, and GLUE tasks are conducted on an EC2 p3.2xlarge instance (featuring a single V100 GPU) using FP32 precision. For BERT fine-tuning and ImageNet training of ResNet-50 and WideResNet-50, the experiments are carried out on an EC2 g4dn.metal instance (equipped with eight T4 GPUs) using FP32 precision. For ImageNet training of DeiT and ResMLP, the experiments are performed on a single p4d.24xlarge instance (housing eight A100 GPUs) with mixed-precision training enabled.

Baseline methods. We implement CUTTLEFISH and all considered baselines in PyTorch (Paszke et al., 2019). The baseline methods under consideration are: (i) PUFFERFISH, which employs manually tuned s (Wang et al., 2021a). To compare with PUFFERFISH, we use the same factorized ResNet-18, VGG-19, ResNet-50, and WideResNet-50 as reported in (Wang et al., 2021a). For DeiT and ResMLP models, not explored in the original PUFFERFISH paper, we adopt the same heuristic of using a fixed global rank ratio $\rho = \frac{1}{4}$, tuning K to match the factorized model sizes found by CUTTLEFISH, and setting $E = 80$ for the entire training epochs $T = 300$ (Wang et al., 2021a); (ii) the factorized low-rank training method with SI and FD proposed by (Khodak et al., 2020) (referred to as "SI&FD"), with ρs of SI&FD tuned to match the sizes of factorized models found by CUTTLEFISH; (iii) training time structured pruning method, or "*early bird ticket*" (EB Train) (You et al., 2020); (iv) the IMP method where each pruning round follows the training length and prunes 20% of the remaining model weights at each level, rewinding to the 6th epoch (Frankle et al., 2019); (v) Gradient Signal Preservation (GraSP) (Wang et al., 2020a); (vi) the learnable factorized low-rank training method, or LC model compression, where layer ranks $\mathcal{R}$ are explicitly optimized jointly with model weights $\mathcal{W}$ via an alternating optimization process (Idelbayev & Carreira-Perpinán, 2020). For GLUE fine-tuning, we compare CUTTLEFISH against DistillBERT and TinyBERT (Sanh et al., 2019; Jiao et al., 2020); (vii) XNOR-Nets for training time quantization method (Rastegari et al., 2016), based on the public PyTorch implementation available at [1].

[1]https://github.com/jiecaoyu/XNOR-Net-PyTorch



(a) VGG-19 on CIFAR-10 (b) VGG-19 on CIFAR-100 (c) VGG-19 on SVHN

Figure 5. Comparisons on the selected ranks $\mathcal{R}$ found by CUTTLEFISH, PUFFERFISH, LC compression, and full ranks for VGG-19 trained on CIFAR-10, CIFAR-100, and SVHN datasets.

CUTTLEFISH with FD. To implement FD, *i.e.,* $\ell(\cdot) + \frac{\lambda}{2}\|\mathbf{UV}^\top\|_F^2$ (where $\ell(\cdot)$ stands for the loss function), one has to compute the gradient on the regularization term, *i.e.,*

$$\nabla_{\mathbf{U}}\frac{\lambda}{2}\|\mathbf{UV}^\top\|_F^2 = \lambda\mathbf{UV}^\top\mathbf{V}; \nabla_{\mathbf{V}}\frac{\lambda}{2}\|\mathbf{UV}^\top\|_F^2 = \lambda\mathbf{U}^\top\mathbf{UV}^\top$$

where one can see there is a shared term $\mathbf{UV}^\top$, which does not need to be recomputed. We optimize the implementation to only compute $\mathbf{UV}^\top$ once. For the hybrid NN architectures, normal ℓ_2 weight decay is conducted over full-rank layers when FD is enabled for factorized low-rank layers.

Extra BatchNorm layers. In experiments where FD is not enabled for CUTTLEFISH, we incorporate an extra BatchNorm (BN) layer following the $\mathbf{U}$ layer, *i.e.,* $\text{BN}_{\mathbf{V}}(\text{BN}_{\mathbf{U}}(\mathbf{x}\mathbf{U})\mathbf{V}^\top)$, drawing inspiration from the network architecture design of MobileNets (Howard et al., 2017). We also apply this approach to PUFFERFISH.

4.2 Experimental results and analysis

How does CUTTLEFISH s compare to manually tuned/learned ones? A crucial question to consider is the appearance of the s returned by CUTTLEFISH. We display the $\mathcal{R}$s discovered by CUTTLEFISH, PUFFERFISH, and LC compression for VGG-19 trained on CIFAR-10, CIFAR-100, and SVHN datasets (results presented in Figure 5). Here, it is evident that CUTTLEFISH provides a selection of $\mathcal{R}$ that closely aligns with explicitly trained rank selections, *i.e.,* LC compression, where rank selection and low-rank model weights are jointly learned during model training. This demonstrates the effectiveness of the rank selection heuristic employed by CUTTLEFISH.

Parameter reduction and model accuracy. We thoroughly investigate the effectiveness of CUTTLEFISH and conduct extensive comparisons against the baselines, with results displayed in Tables 1, 2, 3, and 4. The primary observation is that CUTTLEFISH successfully reduces the number of parameters while only causing minimal loss in accuracy. Notably, for VGG-19 trained on CIFAR-10, CUTTLEFISH identifies a factorized model that is 10.8× smaller than the full-rank (vanilla) VGG-19 model, while achieving even better validation accuracy. In comparison to PUFFERFISH (shown in Table 1), CUTTLEFISH discovers a factorized low-rank model that is 4.4× smaller with similar final model accuracy for VGG-19 trained on CIFAR-10. To achieve a factorized model of the same size, SI&FD does not always yield comparable accuracy to the full-rank model. For instance, for CIFAR-10 and CIFAR-100 trained on VGG-19, SI&FD results in a non-trivial accuracy drop of 1.2% and 1.8%, respectively, because $K = 1$ is always used in SI&FD, which negatively affects the final model accuracy. On ImageNet, CUTTLEFISH attains smaller factorized ResNet-50 (0.5M fewer parameters) and WideResNet-50 (2.7M fewer parameters) with higher top-1 and top-5 validation accuracy compared to PUFFERFISH. For DeiT and ResMLP, we use a fixed rank ratio $\rho = \frac{1}{4}$ and tune the Ks of PUFFERFISH to match the factorized low-rank model sizes of CUTTLEFISH for fair comparisons. PUFFERFISH factorized DeiT and ResMLP consistently result in inferior model accuracy compared to CUTTLEFISH. This occurs because the model weights of DeiT and ResMLP are less likely to be low rank, so using $\rho = \frac{1}{4}$ following the original PUFFERFISH heuristic leads to overly aggressive rank estimations. CUTTLEFISH, in contrast, detects this through a more appropriate rank estimation heuristic.

End-to-end runtime and computational complexity. As discussed in Section 3.5, factorized low-rank training achieves substantial speedups when arithmetic intensity is high. One way to achieve high arithmetic intensity is by using a large batch size for training. Consequently, we use a large batch size of 1,024 and measure the end-to-end training time for the experiments on CIFAR. The results, presented in Table 1, demonstrate that CUTTLEFISH consistently leads to faster end-to-end training time (including full-rank epochs and all other overhead computations, such as profiling and stable rank computing) compared to full-rank training. For instance, CUTTLEFISH achieves 1.2× end-to-end training speedups on both ResNet-18 and VGG-19 trained on CIFAR-10. CUTTLEFISH yields comparable runtime to PUFFERFISH for ResNet-18 and faster runtime on VGG-19 because it finds smaller K for VGG-19, *i.e.,* $K = 4$, while PUFFERFISH uses $K = 9$. SI&FD achieves faster runtime than CUTTLEFISH due to its use of $K = 1$ (and generally higher computational complexities for the initial convolution layers). However, employing such an aggressive value for K inevitably results in accuracy loss, as discussed earlier. Both FC compression and IMP require heavy computation to achieve small models, which are significantly slower than full-rank training. XNOR-Nets

Table 1. The results, averaged across three independent trials with different random seeds, showcase the performance of CUTTLEFISH and other baselines on ResNet-18 and VGG-19 trained over CIFAR-10 and CIFAR-100 using a batch size of 1,024. The runtime benchmark is conducted on a single EC2 p3.2xlarge instance. †: The SI&FD baseline is tuned such that the model size is comparable (albeit slightly larger) to the models that CUTTLEFISH discovers. ⋆: CUTTLEFISH is tested with both FD enabled and disabled, and the results with the best accuracy are reported in the table. ¶: XNOR-Net employs binary weights and activations; although the overall number of trainable parameters remains the same as the vanilla network, each model weight is quantized from 32-bit to 1-bit. Therefore, we report a compression rate of 3.125% for XNOR-Nets. A comprehensive ablation study can be found in the appendix.

	CIFAR-10			CIFAR-100		
Model: ResNet-18	Params. (M)	Val. Acc. (%)	Time (hrs.)	Params. (M)	Val. Acc. (%)	Time (hrs.)
Full-rank	11.2 (100%)	$94.41_{\pm 0.14}$	0.82 (1×)	11.2 (100%)	$75.95_{\pm 0.23}$	0.82 (1×)
PUFFERFISH	3.3 (29.9%)	$94.18_{\pm 0.15}$	0.70 (1.16×)	3.4 (30.2%)	$72.43_{\pm 0.18}$	0.70 (1.17×)
SI&FD †	2.1 (18.5%)	$94.38_{\pm 0.03}$	0.59 (1.39×)	2.7 (24.1%)	$75.80_{\pm 0.17}$	0.70 (1.16×)
IMP	1.9 (16.8%)	$95.04_{\pm 0.03}$	6.55 (0.13×)	2.4 (21.0%)	$75.51_{\pm 0.09}$	5.73 (0.14×)
XNOR-Net¶	11.2 (3.1%)	$88.76_{\pm 0.14}$	3.61 (0.23×)	11.2 (3.1%)	$57.23_{\pm 0.40}$	3.62 (0.23×)
CUTTLEFISH⋆	2.0 (17.9%)	$94.73_{\pm 0.08}$	0.70 (1.18×)	2.6 (23.4%)	$75.57_{\pm 0.24}$	0.69 (1.19×)
Model: VGG-19	Params. (M)	Val. Acc. (%)	Time (hrs.)	Params. (M)	Val. Acc. (%)	Time (hrs.)
Full-rank	20.0 (100%)	$93.41_{\pm 0.15}$	0.50 (1×)	20.1 (100%)	$72.17_{\pm 0.37}$	0.49 (1×)
PUFFERFISH	8.1 (40.5%)	$93.36_{\pm 0.09}$	0.46 (1.09×)	8.2 (40.6%)	$72.43_{\pm 0.18}$	0.46 (1.09×)
SI&FD†	2.0 (10.0%)	$92.23_{\pm 0.08}$	0.34 (1.44×)	3.3 (16.5%)	$70.42_{\pm 0.48}$	0.39 (1.26×)
LC Compress.	1.7 (8.7%)	$93.23_{\pm 0.15}$	5.9 (0.08×)	3.8 (19.0%)	$71.51_{\pm 0.07}$	15.98 (0.03×)
IMP	1.7 (8.6%)	$93.68_{\pm 0.28}$	5.48 (0.09×)	3.4 (16.8%)	$73.39_{\pm 0.32}$	3.96 (0.13×)
XNOR-Net¶	20.0 (3.1%)	$86.61_{\pm 0.10}$	1.43 (0.35×)	20.1 (3.1%)	$49.07_{\pm 0.28}$	1.43 (0.35×)
CUTTLEFISH⋆	1.9 (9.3%)	$93.54_{\pm 0.10}$	0.42 (1.18×)	3.3 (16.3%)	$72.23_{\pm 0.09}$	0.44(1.14×)

employ binary model weights and activations, which results in a reduction of final accuracy for tasks compared to dense networks. Ideally, the use of binarized weights and activations should greatly speed up model training and substantially decrease memory consumption during the process. However, PyTorch lacks an efficient implementation of a binarized convolution operator. Consequently, our experiments utilized FP32 networks and activations to simulate binary networks, leading to a notably slower runtime compared to conventional FP32 training. This is because each layer's output necessitates binarization, and model weights must be re-binarized for every iteration. For ImageNet experiments (presented in Table 2), the memory footprints are high, limiting us to a batch size of 256. CUTTLEFISH identifies factorized ResNet-50 and WideResNet-50 models that achieve 1.2× and 1.3× end-to-end speedups for ImageNet training, respectively. Although the factorized models found by CUTTLEFISH are comparable to PUFFERFISH, it eliminates the need for extensive hyperparameter tuning for such large-scale tasks.

4.3 Computation overheads introduced by CUTTLEFISH.

Computational overheads of profiling. The profiling process in CUTTLEFISH is a lightweight operation. For instance, with ResNet-18 on the CIFAR-10 dataset, we perform profiling using $\tau = 11$ iterations and exclude the running time for the first iteration for both full-rank and low-rank models (*i.e.,* running 22 iterations in total). We then average the running time figures for the remaining 10 iterations for benchmarking purposes. Averaged from three independent runs, the entire profiling stage takes 3.98 seconds, which accounts for a mere 0.16% of the total running time of CUTTLEFISH on ResNet-18 trained on the CIFAR-10 dataset.

Computational overheads of rank estimation. It is important to emphasize that CUTTLEFISH needs to compute the singular values of the entire network weights at the end of each epoch. It is worth noting that to calculate stable ranks, only singular values are required, rather than singular vectors. This process can be accelerated by leveraging APIs, such as `scipy.linalg.svdvals`, which only compute singular values of a given matrix. Taking ResNet-18 trained on CIFAR-10 as an example, the average time taken for rank estimation using the stable rank is 0.49 seconds per epoch. For CUTTLEFISH, which requires $E = 82.3$ epochs (on average) for full-rank training, the stable rank estimation takes a total of 39.97 seconds, accounting for 1.6% of the entire end-to-end running time.

4.4 Ablation Study

Accuracy and runtime efficiency of extra BNs. We conduct additional ablation studies to examine the influence

Table 2. The results presented include vanilla, PUFFERFISH, and CUTTLEFISH implementations of ResNet-50 and WideResNet-50, trained on ImageNet. The FLOPs numbers represent model inference latency, measured using simulated single-batch input with dimensions of $(3, 224, 224)$. Runtime benchmarks are conducted on EC2 g4dn.metal instances.

	# Params. (M)	Val. Acc. Top-1	Val. Acc. Top-5	FLOPs (G)	Time (hrs.)
WideResNet-50	68.9 (100%)	78.1	94.0	11.4	147.8 (1×)
PUFFERFISH	40.0 (58.1%)	$77.86_{\pm 0.05}$	$93.97_{\pm 0.05}$	10.0	112.7 (1.31×)
CUTTLEFISH	37.4 (54.3%)	$78.0_{\pm 0.06}$	$94.04_{\pm 0.09}$	10.0	112.7 (1.31×)
ResNet-50	25.6 (100%)	77.0	93.4	4.1	67.0 (1×)
PUFFERFISH	15.2 (59.5%)	$76.36_{\pm 0.03}$	$93.21_{\pm 0.03}$	3.6	55.6 (1.20×)
CUTTLEFISH	14.7 (57.4%)	$76.44_{\pm 0.16}$	$93.21_{\pm 0.03}$	3.6	56.7 (1.18×)

Table 3. The results for vanilla, PUFFERFISH, and CUTTLEFISH implementations of DeiT-base and ResMLP-S36, trained on ImageNet, are presented. FLOPs numbers, which measure model inference latency, are determined using simulated single-batch input with dimensions of $(3, 224, 224)$.

	# Params. (M)	Val. Acc. Top-1	Val. Acc. Top-5	FLOPs (G)
DeiT-base	86.6	81.8	95.6	17.6
PUFFERFISH	58.3	$81.15_{\pm 0.04}$	$95.58_{\pm 0.04}$	12.0
CUTTLEFISH	58.3	$81.52_{\pm 0.03}$	$95.59_{\pm 0.04}$	12.0
ResMLP-S36	44.7	80.1	95.0	8.9
PUFFERFISH	29.3	$77.78_{\pm 0.20}$	$94.00_{\pm 0.06}$	5.9
CUTTLEFISH	29.4	$78.94_{\pm 0.04}$	$94.52_{\pm 0.05}$	5.8

of incorporating extra BN layers on CUTTLEFISH performance. In our primary experiments, we use FD and disable extra BN layers to ensure accurate FD gradient computation when FD leads to better accuracy. Our ablation studies involve training ResNet-18 and VGG-19 on CIFAR-10 and CIFAR-100 datasets, as well as ResNet-50 on ImageNet, and evaluating model sizes, best validation accuracy (top-1 for ImageNet), end-to-end training time, and per-iteration time on low-rank models. The hyperparameters used in the ablation studies are consistent with those used for the main results in the Experiment section. The ablation study results, shown in Table 5, reveal that adding extra BN layers generally leads to a marginally larger model size and slower per-iteration and end-to-end runtimes. For example, when training ResNet-18 on CIFAR-10 without extra BNs, the end-to-end training time is 1.4% faster, and the per-iteration runtime is 2.8% faster. The impact of extra BNs on final validation accuracy varies across experiments: enabling extra BNs slightly improves accuracy for ResNet-18 and VGG-19 on CIFAR-10, but not for CIFAR-100. However, for the ImageNet experiment, adding extra BNs leads to a non-trivial increase in model accuracy by an average of 0.21% across three independent runs with different random seeds. This improvement is significant, considering it relates to top-1 accuracy for a 1000-class classification problem. There are two potential explanations for why extra BNs help improve accuracy for ImageNet experiments more than CIFAR experiments: 1) The model capacity of ResNet-18 and VGG-19 seems sufficiently large for CIFAR datasets, allowing high compression rates (*e.g.,* , 5×-10×). In contrast, for ResNet-50 on ImageNet, the model capacity appears inadequate. CUTTLEFISH, in this case, does not achieve exceptionally high compression rates (*i.e.,* less than 2×). Consequently, the inclusion of extra BNs appears to provide the low-rank factorized model with additional capacity to enhance its accuracy. 2) For CIFAR experiments, we used a batch size of 1024, constrained by GPU memory, while a batch size of 256 was employed for ImageNet experiments. It is possible that extra BNs offer more substantial benefits in smaller batch settings. Note that for Transformer model-based experiments, we do not enable extra BNs as LayerNorm is commonly used instead of BNs, which is beyond the scope of this ablation study.

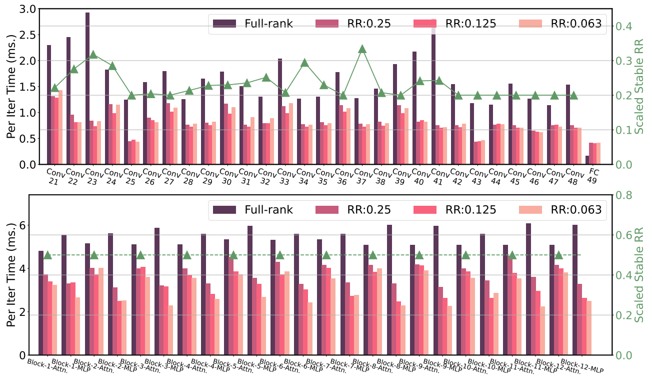


Figure 6. Ablation study on the layer-wise costs of (**Top**): ResNet-50 training on ImageNet (along with the scaled stable rank ratios selected by CUTTLEFISH); (**Bottom**): DeiT-Small training on ImageNet, batch size of both experiments are 128, and time for both experiments are measured on an EC2 p3.2xlarge instance with a batch size of 128. "RR" stands for rank ratio in the figure.

Effectiveness of low-rank factorization on various layer types. In order to compare the efficiency of low-rank factorization against convolution, FC, MLP, and multi-head attention layers, we conducted an ablation study using

Table 4. Vanilla BERT$_{\text{BASE}}$, Distill BERT, Tiny BERT$_6$, as well as CUTTLEFISH BERT$_{\text{BASE}}$ are evaluated on the GLUE benchmark. F1 scores are used as the metric for QQP and MRPC, while Spearman correlations are reported for STS-B, and accuracy scores are reported for the remaining tasks.

Model	# Params. (M)	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B	Avg.
BERT$_{\text{BASE}}$	108.3	**83.9**/**84.4**	**90.9**	**87.6**	66.8	92.2	88.6	**60.1**	88.6	**82.5**
Distill BERT	65.8	81.1/82.0	89.1	86.2	57.8	90.6	88.6	47.3	83.4	78.4
Tiny BERT$_6$	67.0	**83.9**/83.8	90.6	86.8	**72.9**	91.5	**90.6**	46.2	**89.3**	81.7
CUTTLEFISH	**48.8**	83.7/**84.4**	90.8	86.7	67.0	**92.3**	88.4	56.8	87.9	82.0

Table 5. In this ablation study, we evaluate the impact of extra BNs on ResNet-18 and VGG-19 trained on CIFAR-10 and CIFAR-100, as well as ResNet-50 trained on the ImageNet dataset. The end-to-end and per-iteration running time results are measured on a single p3.2xlarge EC2 instance for ResNet-18 and VGG-19, and a single g4dn.metal EC2 instance for ResNet-50. The results are averaged from three independent experiments using different random seeds.

	CIFAR-10				CIFAR-100			
Model: ResNet-18	Params. (M)	Val. Acc. (%)	Time End2end (hrs.)	Time Iter. (ms)	Params. (M)	Val. Acc. (%)	Time End2end (hrs.)	Time Iter. (ms)
w/ extra BNs	2.02	$94.36_{\pm 0.07}$	0.716	$163.42_{\pm 0.51}$	2.62	$73.64_{\pm 0.26}$	0.719	$164.72_{\pm 1.63}$
w/o extra BNs	1.97	$94.32_{\pm 0.22}$	0.706	$158.94_{\pm 0.53}$	2.60	$73.77_{\pm 0.14}$	0.689	$158.53_{\pm 1.32}$
Model: VGG-19	Params. (M)	Val. Acc. (%)	Time End2end (hrs.)	Time Iter. (ms)	Params. (M)	Val. Acc. (%)	Time End2end (hrs.)	Time Iter. (ms)
w/ extra BNs	1.86	$93.54_{\pm 0.10}$	0.422	$85.53_{\pm 0.59}$	3.31	$71.99_{\pm 0.02}$	0.436	$91.76_{\pm 0.52}$
w/o extra BNs	1.86	$93.49_{\pm 0.08}$	0.419	$84.55_{\pm 0.26}$	3.31	$72.15_{\pm 0.24}$	0.432	$89.90_{\pm 0.18}$

ResNet-50 on ImageNet	Params. (M)	Top-1 Val. Acc. (%)	Time End2end (hrs.)	Time Iter. (sec.)
w/ extra BNs	14.7	$76.44_{\pm 0.16}$	56.7	$0.43_{\pm 0.002}$
w/o extra BNs	14.7	$76.23_{\pm 0.21}$	55.6	$0.42_{\pm 0.003}$

ResNet-50 and DeiT-small on the ImageNet dataset. The per-iteration time of each layer was measured and the results are depicted in Figure 6 (for ResNet-50, we also illustrated the scaled stable rank ratios selected by CUTTLEFISH). We focused on forward computation time for this study, as it is well known that there is a constant factor between forward and backward computing time, and the former serves as a good proxy for per-iteration time. Due to space constraints, we only plotted the results for the 21st convolution layer onwards in the ResNet-50 experiments, although meaningful speedups were observed for the first 20 layers as well. In the case of convolution layers, our experiments revealed an average speedup of $2.1\times$ across all 49 layers when using a rank ratio of $\frac{1}{4}$. However, we observed that the last FC layer actually slowed down when factorized, regardless of the rank ratio used. This could be attributed to the small size of the FC layer, which incurs a large kernel launching overhead when split into two smaller layers, thereby nullifying any computation cost savings. Our experiments with DeiT showed that factorizing both multi-head attention and MLP layers resulted in significant speedups for all 12 Transformer encoder blocks. Additionally, factorizing the MLP layer led to greater speedup gains compared to factorizing the multi-head attention layer. Specifically, factorizing the multi-head attention layer resulted in $1.26\times$ speedups on average, while factorizing the MLP layer resulted in $1.73\times$ speedups on average for all 12 blocks at a rank ratio of $\frac{1}{4}$.

4.5 Limitations of CUTTLEFISH

A limitation of CUTTLEFISH is that the hyperparameters s it tunes are influenced by the randomness of the training algorithm and model initialization. Consequently, different trial runs may not yield factorized models with identical sizes (although the variance is minimal). This can potentially impact exact reproducibility.

5 CONCLUSION

We present CUTTLEFISH, an automated low-rank training method that eliminates the need for tuning additional factorization hyperparameters, *i.e.,* $\mathcal{S} = (E, K, \mathcal{R})$. CUTTLEFISH leverages two key insights related to the emergence of stable ranks during training and the actual speedup gains achieved by factorizing different NN layers. Utilizing these insights, it designs heuristics for automatically selecting $s \in \mathcal{S}$. Our extensive experiments demonstrate that CUTTLEFISH identifies low-rank models that are not only smaller, but also yield better final accuracy in most cases when compared to state-of-the-art low-rank training methods.

Acknowledgments

We thank our shepherd, Bilge Acun, and the anonymous MLSys reviewers for their valuable insights and recommendations, which have enhanced our work. This research has been graciously funded by ONR Grant No. N00014-21-1-2806, three Sony Faculty Innovation Awards, NSF IIS1563887, NSF CCF1629559, NSF IIS1617583, NSF IIS1955532, NSF CNS2008248, NSF IIS2123952, and NSF BCS2040381, and NGA HM04762010002.

REFERENCES

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020.

Chen, B., Dao, T., Winsor, E., Song, Z., Rudra, A., and Ré, C. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems*, 2021a.

Chen, B., Dao, T., Liang, K., Yang, J., Song, Z., Rudra, A., and Re, C. Pixelated butterfly: Simple and efficient sparse training for neural network models. In *International Conference on Learning Representations*, 2022.

Chen, P., Yu, H.-F., Dhillon, I., and Hsieh, C.-J. Drone: Data-aware low-rank compression for large nlp models. *Advances in neural information processing systems*, 34: 29321–29334, 2021b.

Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

Chollet, F. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

Dao, T., Chen, B., Sohoni, N., Desai, A., Poli, M., Grogan, J., Liu, A., Rao, A., Rudra, A., and Ré, C. Monarch: Expressive structured matrices for efficient and accurate training. *arXiv preprint arXiv:2204.00595*, 2022.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

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

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, pp. 2943–2952. PMLR, 2020.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. Stabilizing the lottery ticket hypothesis. *arXiv preprint arXiv:1903.01611*, 2019.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015b.

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

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1389–1397, 2017.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In *Advances in neural information processing systems*, pp. 4107–4115, 2016.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

Hyeon-Woo, N., Ye-Bin, M., and Oh, T.-H. Fedpara: Low-rank hadamard product for communication-efficient federated learning. In *International Conference on Learning Representations*, 2022.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Idelbayev, Y. and Carreira-Perpinán, M. A. 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*, pp. 8049–8059, 2020.

Ioannou, Y., Robertson, D., Shotton, J., Cipolla, R., and Criminisi, A. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*, 2015.

Izsak, P., Berchansky, M., and Levy, O. How to train bert with an academic budget. *arXiv preprint arXiv:2104.07705*, 2021.

Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

Jeffers, J., Reinders, J., and Sodani, A. *Intel Xeon Phi processor high performance programming: knights landing edition*. Morgan Kaufmann, 2016.

Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. Tinybert: Distilling bert for natural language understanding. In *EMNLP 2020*, pp. 4163–4174, 2020.

Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z., Cormode, G., Cummings, R., et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

Khodak, M., Tenenholtz, N. A., Mackey, L., and Fusi, N. Initialization and regularization of factorized neural layers. In *International Conference on Learning Representations*, 2020.

Kitaev, N., Kaiser, L., and Levskaya, A. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

Li, D., Wang, H., Shao, R., Guo, H., Xing, E., and Zhang, H. Mpcformer: fast, performant and private transformer inference with mpc. In *The Eleventh International Conference on Learning Representations*, 2023.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Lin, M., Chen, Q., and Yan, S. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, pp. 5, 2011.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pp. 8026–8037, 2019.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*, pp. 525–542. Springer, 2016.

Renda, A., Frankle, J., and Carbin, M. Comparing rewinding and fine-tuning in neural network pruning. In *International Conference on Learning Representations*, 2020.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6655–6659. IEEE, 2013.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sreenivasan, K., Sohn, J.-y., Yang, L., Grinde, M., Nagle, A., Wang, H., Xing, E., Lee, K., and Papailiopoulos, D. Rare gems: Finding lottery tickets at initialization. *Advances in Neural Information Processing Systems*, 2022a.

Sreenivasan, K., yong Sohn, J., Yang, L., Grinde, M., Nagle, A., Wang, H., Xing, E., Lee, K., and Papailiopoulos, D. Rare gems: Finding lottery tickets at initialization. In *Advances in Neural Information Processing Systems*, 2022b.

Tan, M. and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

Tolstikhin, I. O., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., et al. Mlp-mixer: An all-mlp architecture for vision. *Advances in Neural Information Processing Systems*, 34, 2021.

Touvron, H., Bojanowski, P., Caron, M., Cord, M., El-Nouby, A., Grave, E., Izacard, G., Joulin, A., Synnaeve, G., Verbeek, J., et al. Resmlp: Feedforward networks for image classification with data-efficient training. *arXiv preprint arXiv:2105.03404*, 2021a.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jégou, H. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pp. 10347–10357. PMLR, 2021b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Vodrahalli, K., Shivanna, R., Sathiamoorthy, M., Jain, S., and Chi, E. Algorithms for efficiently learning low-rank neural networks. *arXiv preprint arXiv:2202.00834*, 2022.

Waleffe, R. and Rekatsinas, T. Principal component networks: Parameter reduction early in training. *arXiv preprint arXiv:2006.13347*, 2020.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

Wang, C., Zhang, G., and Grosse, R. Picking winning tickets before training by preserving gradient flow. *International Conference on Learning Representations*, 2020a.

Wang, H., Yurochkin, M., Sun, Y., Papailiopoulos, D., and Khazaeni, Y. Federated learning with matched averaging. In *International Conference on Learning Representations*, 2020b.

Wang, H., Agarwal, S., and Papailiopoulos, D. Pufferfish: Communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems*, 3, 2021a.

Wang, J., Charles, Z., Xu, Z., Joshi, G., McMahan, H. B., Al-Shedivat, M., Andrew, G., Avestimehr, S., Daly, K., Data, D., et al. A field guide to federated optimization. *arXiv preprint arXiv:2107.06917*, 2021b.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pp. 2074–2082, 2016.

Wiesler, S., Richard, A., Schluter, R., and Ney, H. Mean-normalized stochastic gradient for large-scale deep learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 180–184. IEEE, 2014.

Wu, J., Leng, C., Wang, Y., Hu, Q., and Cheng, J. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4820–4828, 2016.

Xue, J., Li, J., and Gong, Y. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, pp. 2365–2369, 2013.

Yang, T.-J., Chen, Y.-H., and Sze, V. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5687–5695, 2017.

Yao, D., Pan, W., Wan, Y., Jin, H., and Sun, L. Fedhm: Efficient federated learning for heterogeneous models via low-rank factorization. *arXiv preprint arXiv:2111.14655*, 2021.

You, H., Li, C., Xu, P., Fu, Y., Wang, Y., Chen, X., Baraniuk, R. G., Wang, Z., and Lin, Y. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 2019.

You, H., Li, C., Xu, P., Fu, Y., Wang, Y., Chen, X., Baraniuk, R. G., Wang, Z., and Lin, Y. Drawing early-bird tickets: Toward more efficient training of deep networks. In *International Conference on Learning Representations*, 2020.

Yu, J. and Huang, T. S. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1803–1811, 2019.

Yu, J., Yang, L., Xu, N., Yang, J., and Huang, T. Slimmable neural networks. In *International Conference on Learning Representations*, 2019.

Yu, R., Li, A., Chen, C.-F., Lai, J.-H., Morariu, V. I., Han, X., Gao, M., Lin, C.-Y., and Davis, L. S. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9194–9203, 2018.

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

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6848–6856, 2018.

Zhou, A., Yao, A., Guo, Y., Xu, L., and Chen, Y. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

Zhu, M. and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

A ARTIFACT APPENDIX

A.1 Abstract

We have made available the necessary artifacts to replicate all results presented in the paper. Our experiments utilize Amazon EC2 computing resources, including *p3.2xlarge* (for ResNet-18 and VGG-19 training on CIFAR-10 and CIFAR-100 datasets, as well as BERT fine-tuning on the GLUE benchmark), *g4dn.metal* (for example, ResNet-50 and WideResNet-50 training on ImageNet), and *p4d.24xlarge* (for DeiT-base and ResMLP execution on ImageNet) instances. Additionally, we employ the NVIDIA driver and Docker to construct the software stack.

To facilitate the replication of all reported experiments, we provide scripts in our GitHub repository, accessible at `https://github.com/hwang595/Cuttlefish`. Running those provided scripts will set up and launch experiments to reproduce our experimental results. For ease of reproducibility, we also offer a public Amazon Machine Image (AMI) – *ami-05c0b3732203032b3* (in the region of *US West (Oregon)*) where experimental environments and the ImageNet dataset, which is time-consuming to download and set up are prepared.

A.2 Artifact check-list (meta-information)

In this section, we offer meta-information regarding the configuration, datasets, implementation, and other aspects of our artifacts.

- **Algorithm:** Our artifact encompasses the CUTTLEFISH automatic low-rank training schedule, along with the baseline methods compared in the main paper, such as PUFFERFISH, SI&FD, XNOR-Net, GraSP, and others.
- **Program:** N/A
- **Compilation:** All methods and baselines are implemented in PyTorch, therefore requiring no compilation.
- **Transformations:** N/A
- **Binary:** N/A
- **Data set:** For our main experiments, we employ CIFAR-10, CIFAR-100, SVHN, ImageNet (ILSVRC 2012), and GLUE datasets. As preparing the ImageNet dataset can be time-consuming, we provide a ready-to-use public AMI - *ami-05c0b3732203032b3* in the *US West (Oregon)* region for convenience.
- **Run-time environment:** N/A
- **Hardware:** Our experiments were conducted using Amazon EC2 instances, specifically *p3.2xlarge*, *g4dn.metal*, and *p4d.24xlarge*.
- **Run-time state:** N/A
- **Execution:** We provide scripts to execute and launch the experiments. Detailed descriptions and instructions can be found in the README of our GitHub repository.
- **Metrics:** We collect metrics such as the number of parameters, validation accuracy (or similar scores for assessing model quality), wall-clock time (including end-to-end and per iteration/epoch durations), and computational complexity (measured in FLOPS).
- **Output:** Our existing code writes checkpoints to the local disk and also prints experimental outputs/logs directly.
- **Experiments:** N/A
- **How much disk space required (approximately)?:** Around 1 Terabyte of disk space.
- **How much time is needed to prepare workflow (approximately)?:** Setting up the experimental environment should take less than an hour. Downloading the ImageNet (ILSVRC 2012) dataset can take a few days. If the evaluators have access to AWS, we have also provided a public AMI - *ami-05c0b3732203032b3* (in the region of *US West (Oregon)*) which has the datasets and dependencies pre-installed.
- **How much time is needed to complete experiments (approximately)?:** Completing the CIFAR-10 and CIFAR-100 experiments with CUTTLEFISH typically takes less than an hour for each task (see Table 1 for details). BERT fine-tuning experiments on all GLUE datasets require approximately a few hours, while ImageNet experiments may take several days to a week to reach full convergence for each method.
- **Publicly available?:** All our code is publicly available on the GitHub repository: `https://github.com/hwang595/Cuttlefish`. For easy setup on AWS, we also provide a public AMI - with ID *ami-05c0b3732203032b3* (in the region of *US West (Oregon)*), which can be used to launch large-scale experiments.
- **Code licenses (if publicly available)?:** N/A
- **Data licenses (if publicly available)?:** We use CIFAR-10, CIFAR-100, SVHN, Imagenet (ILSVRC 2012) datasets and the GLUE benchmark which come with their own licenses. All datasets are publicly available.
- **Workflow framework used?:** N/A
- **Archived (provide DOI)?:** We use Zenedo to create a publicly accessible archival repository for our GitHub repository, *i.e.,* `https://doi.org/10.5281/zenodo.7884872`.

A.3 Description

We have made available the code necessary to replicate all the experiments presented in this paper through a public GitHub repository that contains comprehensive documentation, allowing users to seamlessly execute the experiments.

A.3.1 How delivered

Our entire codebase is available on the GitHub repository: `https://github.com/hwang595/Cuttlefish`. To facilitate easy setup on AWS, we offer a public AMI - identified by *ami-05c0b3732203032b3* (in the region of *US West (Oregon)*) - which can be utilized to launch large-scale experiments.

A.3.2 Hardware dependencies

For all our experiments we used *p3.2xlarge*, *g4dn.metal*, and *p4d.24xlarge* Amazon EC2 instances. To reproduce our results, one instance of each type is required.

A.3.3 Software dependencies

We established our experimental environments using Docker, configuring them through PyTorch Docker containers from NVIDIA GPU Cloud (NGC). The experiments involving the CIFAR-10, CIFAR-100, SVHN, and ImageNet datasets were based on the NGC container *nvcr.io/nvidia/pytorch:20.07-py3*, while those focused on the GLUE benchmark utilized the *nvcr.io/nvidia/pytorch:22.01-py3* container. Since there are additional software dependencies not included in the Docker containers, we have supplied installation scripts, accompanied by instructions in the README file of our GitHub repository, to facilitate the installation of these necessary components.

A.3.4 Data sets

For all our experiments we used CIFAR-10, CIFAR-100, SVHN, ImageNet (ILSVRC 2012), and GLUE datasets. For CIFAR-10, CIFAR-100, SVHN, and GLUE datasets, our code will automatically download them. For the ImageNet dataset, we have it ready and provided via the public AMI - with ID *ami-05c0b3732203032b3* (in the region of *US West (Oregon)*).

A.4 Installation

In the GitHub README, we offer comprehensive instructions for installing dependencies and configuring the Docker environments.

A.5 Experiment workflow

We have provided a detailed README along with our GitHub repository which provides bash scripts to execute and launch the experiments.

A.6 Evaluation and expected result

During the experiment, logs containing details such as accuracy and running time will be displayed directly. However, given the inherent variability in machine learning tasks and the diversity of hardware and system configurations, it is important to note that the exact accuracy and running time figures reported in the paper may not be replicated. Nevertheless, by using the artifacts provided, one can expect to achieve comparable accuracy and running time outcomes.

A.7 Experiment customization

The experiment can be customized by trying on different hardware setups. One example of this will be to run these experiments on slower GPUs (or other hardware, *e.g.,* CPUs). Another option would be to try to support more model architectures using the heuristics of CUTTLEFISH (an interesting example will be adopting CUTTLEFISH for some recently designed large language models).

A.8 Notes

If the evaluator utilizes our provided AMI, the initial disk initialization will take an extended period of time during the first run.

B EXPERIMENTAL SETUP

In this section, we delve into the specifics of the datasets B.1 and model architectures B.2 employed in our experiments. Additionally, we elaborate on the software environment B.3 and the implementation details of all methods included in our experiments B.4. Our code can be accessed at `https://github.com/hwang595/Cuttlefish`.

B.1 Dataset

We carried out experiments across multiple computer vision and NLP tasks to evaluate the performance of CUTTLEFISH and the other considered baselines. In this section, we discuss the specifics of each task in greater detail.

CIFAR-10 and CIFAR-100. Both CIFAR-10 and CIFAR-100 comprise 60,000 color images with a resolution of 32×32 pixels, where 50,000 images are used for training and 10,000 for validation (since there is no provided test set for CIFAR-10 and CIFAR-100, we follow the convention of other papers by conducting experiments and reporting the highest achievable accuracy on the validation datasets) (Krizhevsky et al., 2009). CIFAR-10 and CIFAR-100 involve 10-class and 100-class classification tasks, respectively. For data processing, we employ standard augmentation techniques: channel-wise normalization, random horizontal flipping, and random cropping. Each color channel is normalized with the following mean and standard deviation values: $\mu_r = 0.485, \mu_g = 0.456, \mu_b = 0.406$; $\sigma_r = 0.229, \sigma_g = 0.224, \sigma_b = 0.225$. The normalization of each channel pixel is achieved by subtracting the corresponding channel's mean value and dividing by the color channel's standard deviation.

SVHN. The SVHN dataset comprises 73,257 training images and 26,032 validation images, all of which are colored with a resolution of 32×32 pixels (Netzer et al., 2011). This classification dataset consists of 10 classes. As there is no clear test-validation split for the SVHN dataset, we follow the convention of other papers by conducting experiments and reporting the highest achievable accuracy on the validation datasets. There are 531,131 additional images for SVHN, but we do not include them in our experiments for this paper. For data processing, we employ the same data augmentation and normalization techniques used for CIFAR-10 and CIFAR-100, as described above.

ImageNet (ILSVRC 2012). The ImageNet ILSVRC 2012 dataset consists of 1,281,167 colored training images spanning 1,000 classes and 50,000 colored validation images, also covering 1,000 classes (Deng et al., 2009). Augmentation techniques include normalization, random rotation, and random horizontal flip. The training images are randomly resized and cropped to a resolution of 224×224 using the torchvision API `torchvision.transforms.RandomResizedCrop`. The validation images are first resized to a resolution of 256×256 and then center cropped to a resolution of 224×224. Each color channel is normalized with the following mean and standard deviation values: $\mu_r = 0.485, \mu_g = 0.456, \mu_b = 0.406$; $\sigma_r = 0.229, \sigma_g = 0.224, \sigma_b = 0.225$. Each channel pixel is normalized by subtracting the corresponding channel's mean value and then dividing by the color channel's standard deviation.

GLUE benchmark. For the GLUE benchmark, we utilize the data preparation and pre-processing pipeline implemented by Hugging Face [2]. In accordance with prior work (Devlin et al., 2018; Jiao et al., 2020; Hu et al., 2021), we exclude the problematic WNLI downstream task.

B.2 Model architectures

In this section, we provide a summary of the network architectures utilized in our experiments.

ResNet-18, ResNet-50, and WideResNet-50-2. The ResNet-18 and ResNet-50 architectures are derived from the original design with minor modifications (He et al., 2016). The WideResNet-50-2 adheres to the original wide residual network design (Zagoruyko & Komodakis, 2016). As we employed ResNet-18 for CIFAR-10 classification, we adjusted the initial convolution layer to use a 3×3 convolution with padding at 1 and stride at 1. Our ResNet-18 implementation follows the GitHub repository [3]. For all ResNet-18, ResNet-50, and WideResNet-50-2 networks, the strides used for the four convolution layer stacks are respectively $1, 2, 2, 2$. Bias terms for all layers are deactivated (owing to the BatchNorm layers), except for the final FC layer.

VGG-19-BN. In our experiments, we employ the VGG-19-BN network architecture, which is a modified version of the original VGG-19 (Simonyan & Zisserman, 2014). The original VGG-19 network consists of 16 convolution layers and 3 FC layers, including the final linear classification layer. We adopt the VGG-19 architectures from (Frankle & Carbin, 2018; Khodak et al., 2020), which remove the first two FC layers following the last convolution layer while retaining the final linear classification layer. This results in a 17-layer architecture, but we continue to refer to it as VGG-19-BN since it stems from the original VGG-19 design. Another modification is replacing the max pooling layer after the last convolution layer (`conv16`) with an average pooling layer. The detailed architecture is displayed in Table 7. We follow the implementation from this repository [3]. Due to the BatchNorm layers, bias terms for all layers are deactivated, except for the final FC layer.

DeiT and ResMLP. Our implementations of DeiT-base and ResMLP-S36 are sourced directly from the model implementations provided by the Pytorch Image Models (*i.e.,* `timm`) library [4]. For DeiT-base, we do not use the scaled ImageNet resolution version and we deactivate the distillation options. More specifically, we initiate the training of a `deit_base_patch16_224` model from scratch, as provided by the `timm` library. For training, we employ the training method and hyperparameters specified in the original GitHub repository [5]. For ResMLP-S36, we adhere to the same training methodology used for DeiT-base, utilizing the `resmlp_36_224` provided by the `timm` library.

BERT$_{\text{BASE}}$, DistillBERT, and TinyBERT. The implementations of BERTBASE, DistillBERT, and TinyBERT$_6$ are directly provided by Hugging Face. For BERTBASE, we use the model named `bert-base-cased`. For DistillBERT, we

[2]`https://github.com/huggingface/transformers/tree/main/examples/pytorch/text-classification`

[3]`https://github.com/kuangliu/pytorch-cifar`

[4]`https://github.com/rwightman/pytorch-image-models`

[5]`https://github.com/facebookresearch/deit`

Table 6. The ResNet-18, ResNet-50, and WideResNet-50-2 network architectures used in our experiments. It should be noted that when using ResNet-18 for CIFAR-10 training, we make corresponding adjustments to the initial convolution layer. Each convolution layer is followed by a BatchNorm layer. In the notation used in this table, "$7 \times 7, 64$" signifies that the convolution layer contains 64 convolution kernels, *i.e.,* each kernel has a dimension of 7×7 and the output dimension is 64.

Model	ResNet-18	ResNet-50	WideResNet-50-2
Conv 1	3×3, 64 padding 1 stride 1 -	7×7, 64 padding 3 stride 2 Max Pool, kernel size 3, stride 2, padding 1	7×7, 64 padding 3 stride 2
Layer stack 1	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 256 \end{bmatrix}\times3$
Layer stack 2	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 512 \end{bmatrix}\times4$
Layer stack 3	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 1024 \end{bmatrix}\times6$
Layer stack 4	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 1024 \\ 3\times3, 1024 \\ 1\times1, 2048 \end{bmatrix}\times3$
FC	Avg Pool, kernel size 4 512×10	Adaptive Avg Pool, output size $(1, 1)$ 2048×1000	 2048×1000

employ the model named `distilbert-base-cased`. For BERT$_{\text{BASE}}$, we use the model named `bert-base-cased` again. For TinyBERT$_6$, we utilize the model named `huawei-noah/TinyBERT_General_6L_768D`. All model names are supplied through the API of `--model_name_or_path` in Hugging Face.

B.3 Software details

For the experiments on CIFAR-10, CIFAR-100, and SVHN, which include CUTTLEFISH and all considered baseline methods, our software setup is built on the NVIDIA NGC Docker container for PyTorch. We use the docker image `nvcr.io/nvidia/pytorch:20.07-py3` to set up the experiment environment on `p3.2xlarge` EC2 instances. The CUDA version we used is 11.6. For the BERT$_{\text{BASE}}$ fine-tuning experiment on the GLUE benchmark, we employ the docker image, `nvcr.io/nvidia/pytorch:22.01-py3`. We install Hugging Face with version `4.17.0.dev0`.

B.4 Implementation details

For all our experiments, we set

```
torch.backends.cudnn.benchmark = True
```

and

```
torch.backends.cudnn.deterministic = False
```

to optimize the running speed of the experiments, as the cuDNN benchmark searches for the fastest low-level implementations. However, perfect reproducibility cannot be guaranteed under this setup. To measure runtime, we employ `torch.cuda.Event(enable_timing=True)` to determine the elapsed time between two CUDA Event records. We fine-tune the `num_workers` and enable `pin_memory` for all experiments to achieve faster end-to-end runtimes. For the DeiT and ResMLP experiments on ImageNet, we enable mixed-precision training using PyTorch AMP.

Examples of factorized low-rank layers. As discussed, a full-rank layer $\mathbf{W}$ can be factorized to obtain $\mathbf{U}$ and $\mathbf{V}^\top$. For fully connected (or linear) layers in ResMLP, BERT, and DeiT models, the dimensions of $\mathbf{U}$ and $\mathbf{V}^\top$ are straightforward. For instance, if $\mathbf{W}$ is a $(784, 784)$ linear projection implemented using `nn.Linear` in PyTorch, then $\mathbf{U}$ and $\mathbf{V}^\top$ can be implemented using $(784, r)$ and $(r, 784)$ as input dimensions for `nn.Linear` in PyTorch. For convolution layers, we use 1×1 convolution for $\mathbf{V}^\top$, following the suggestion of (Wang et al., 2021a). As a concrete example, when factorizing the 16th convolution layer in the VGG-19 architecture we used with $r = 32$, our factorization results in $\mathbf{U}$ as a $512 \times 32 \times 3 \times 3$ dimensional `nn.Conv2d` layer in PyTorch and $\mathbf{V}^\top$ as a $32 \times 512 \times 1 \times 1$ dimensional `nn.Conv2d` layer in PyTorch. Other factorized low-rank convolution networks in our implementations follow the same approach.

C DETAILS ON HYPERPARAMETERS.

In this section, we discuss general-purpose hyperparameters, such as learning rate and training schedule, used in our experiments for each task. Additionally, we discuss the hyperparameter setup of CUTTLEFISH and the details on the final $\hat{s} \in \mathcal{S}$ that CUTTLEFISH manages to find for each experiment.

C.1 General purpose hyperparameters

CIFAR-10 and CIFAR-100. For the CIFAR-10 and CIFAR-100 tasks, training on ResNet-18 and VGG-19, we train for $T = 300$ epochs in total using the SGD optimizer with momentum

Table 7. A detailed description of the VGG-19 architecture in our experiments. After each convolution layer, a BatchNorm layer followed by a ReLU activation is included (though not shown in the table). The shapes for convolution layers are represented as (m, n, k, k).

Parameter	Shape	Layer hyperparameter
layer1.conv1.weight	$3 \times 64 \times 3 \times 3$	stride:1;padding:1
layer2.conv2.weight	$64 \times 64 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer3.conv3.weight	$64 \times 128 \times 3 \times 3$	stride:1;padding:1
layer4.conv4.weight	$128 \times 128 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer5.conv5.weight	$128 \times 256 \times 3 \times 3$	stride:1;padding:1
layer6.conv6.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer7.conv7.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer8.conv8.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer9.conv9.weight	$256 \times 512 \times 3 \times 3$	stride:1;padding:1
layer10.conv10.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer11.conv11.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer12.conv12.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer13.conv13.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer14.conv14.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer15.conv15.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer16.conv16.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
pooling.avg	N/A	kernel size:2
classifier.weight	512×10	N/A
classifier.bias	10	N/A

(Nesterov momentum disabled). We use a batch size of $B = 1,024$, scaling an initial learning rate from $\gamma = 0.1$ to $\gamma = 0.8$ in 5 epochs linearly, following the procedure proposed in (Goyal et al., 2017). We set momentum and weight decay coefficients at 0.9 and 1×10^{-4}, respectively (weight decay is disabled for BatchNorm layers). We employ a multi-step learning rate schedule, decaying the learning rate by a factor of 0.1 at the 150-th and 225-th epochs (to $\gamma = 0.08$ and $\gamma = 0.008$ respectively). The methodology we follow for this multi-step learning rate decay is to decay the learning rate by a factor of 0.1 at the points of 50% and 75% of the entire training epochs.

SVHN. For the SVHN task, training on ResNet-18 and VGG-19, we train for $T = 200$ epochs in total using the SGD optimizer with momentum (Nesterov momentum disabled). We train SVHN for a shorter number of epochs, as it is a relatively easier task compared to CIFAR-10 and CIFAR-100. We use a batch size of $B = 1,024$, scaling an initial learning rate from $\gamma = 0.1$ to $\gamma = 0.8$ in 5 epochs linearly, following the procedure proposed in (Goyal et al., 2017). We set momentum and weight decay coefficients at 0.9 and 1×10^{-4}, respectively (weight decay is disabled for BatchNorm layers). We employ a multi-step learning rate schedule, decaying the learning rate by a factor of 0.1 at the 100-th and 150-th epochs (to $\gamma = 0.08$ and $\gamma = 0.008$ respectively).

ImageNet (ILSVRC 2012) on CNNs. For the ImageNet training task on ResNet-50 and WideResNet-50-2, we train for $T = 90$ epochs in total using the SGD optimizer with momentum (Nesterov momentum disabled) and a batch size of 256. We set momentum and weight decay coefficients at 0.9 and 1×10^{-4}, respectively (weight decay is disabled for BatchNorm layers). We employ a multi-step learning rate schedule, decaying the learning rate by a factor of 0.1 at the 30-th, 60-th, and 80-th epochs (to $\gamma = 0.01$, $\gamma = 0.001$, and 0.0001 respectively), following (Goyal et al., 2017). We adopt the label smoothing technique with a probability of 0.1 to achieve better final accuracy and to compare against PUFFERFISH (Wang et al., 2021a). For the comparison against PUFFERFISH, GraSP, and EBTrain, we disable the learning rate decay at the 80-th epoch and label smoothing to align the experiment setup.

ImageNet (ILSVRC 2012) on DeiT and ResMLP. For ImageNet training on DeiT and ResMLP, we adhere to the training schedule proposed by (Touvron et al., 2021b), wherein the models are trained on ImageNet directly from scratch. Our experiments adopt the model initialization, data augmentation, and Exponential Moving Average (EMA) methods suggested by (Touvron et al., 2021b). We do not enable distillation for DeiT and ResMLP. The AdamW optimizer is used for our experiments (Loshchilov & Hutter, 2019). More details on the hyperparameter values can be found at `https://github.com/facebookresearch/deit/blob/main/README_deit.md`, where we use the default hyperparameter setup and set the batch size to 256.

BERT fine-tuning on the GLUE benchmark. We directly utilize the training script provided by Hugging Face at `https:`

`//github.com/huggingface/transformers/tree/main/examples/pytorch/text-classification`. We set the maximum sequence length to 128 and the batch size to 32, using the AdamW optimizer (Loshchilov & Hutter, 2019) for all downstream tasks in the GLUE benchmark. For each downstream task in GLUE, we conduct a small hyperparameter sweep within the range of $\{1e-5, 2e-5, 3e-5, 4e-5\}$, employing early stopping. For the relatively small downstream task MRPC, we fine-tune for 5 epochs, while for all other downstream tasks, we fine-tune for 3 epochs. We disable weight decay and learning rate warm-up during the fine-tuning process.

C.2 CUTTLEFISH hyperparameters

CIFAR-10, CIFAR-100, and SVHN. For ResNet-18 and VGG-19 models trained on CIFAR-10, CIFAR-100, and SVHN datasets, we report the details of the hyperparameters $\hat{s} \in \mathcal{S}$ discovered by CUTTLEFISH and the manually tuned hyperparameters used by PUFFERFISH in Table 8. It is evident that CUTTLEFISH identifies larger K values and smaller E values (except for CIFAR-10) compared to PUFFERFISH for ResNet-18, while for VGG-19, CUTTLEFISH discovers smaller E values with slightly longer E values than PUFFERFISH. Additionally, the selected ranks, i.e., $\mathcal{R}$, for CUTTLEFISH, PUFFERFISH, and LC compression (only for VGG-19) methods are illustrated in Figure 7. Notably, CUTTLEFISH consistently returns lower estimated ranks for bottom layers than PUFFERFISH, as these layers contain greater redundancy. The most striking observation from Figure 7 is that the $\mathcal{R}$ values returned by CUTTLEFISH closely align with the explicitly learned $\mathcal{R}$ values of LC compression, demonstrating CUTTLEFISH's effectiveness. Another interesting finding from Figure 7 is that more challenging tasks generally require higher ranks for the factorized low-rank network to achieve satisfactory accuracy. For example, both CUTTLEFISH and LC compression identify larger $\mathcal{R}$ values for CIFAR-100 and smaller $\mathcal{R}$ values for SVHN and CIFAR-10. This is because CIFAR-100, a 100-class classification task, is more challenging than CIFAR-10 and SVHN for a given model architecture.

ImageNet (ILSVRC 2012) on CNNs. For ResNet-50 and WideResNet-50-2 trained on ImageNet, we report the details of the hyper-parameters $\hat{s} \in \mathcal{S}$ found by CUTTLEFISH and the manually tuned hyper-parameters by PUFFERFISH in Table 9. We can observe that for both ResNet-50 and WideResNet-50-2, CUTTLEFISH identifies the same K and longer E compared to PUFFERFISH. The ranks ($\mathcal{R}$s) chosen by CUTTLEFISH and PUFFERFISH can be found in Figure 8, where it is evident that CUTTLEFISH employs lower ranks to factorize layers in ResNet-50 and WideResNet50-2 while using longer full-rank training epochs.

ImageNet (ILSVRC 2012) on DeiT and ResMLP. For DeiT-base and ResMLP-S36 trained on ImageNet, we report the details of the hyperparameters $\hat{s} \in \mathcal{S}$ found by CUTTLEFISH and the manually tuned hyperparameters by PUFFERFISH in Table 10. We can observe that for DeiT-base and ResMLP-S36, CUTTLEFISH identifies the same K and longer E compared to PUFFERFISH. When selecting the ranks $\mathcal{R}$s, we found that even with scaled stable rank, the factorized low-rank models still result in a significant final accuracy drop. To understand why this occurs, we plot the cumulative distribution function (CDF) of the singular values of the Transformer encoder layers in the DeiT-base model at the full-rank to low-rank switching epoch, *i.e.,* $\hat{E}$ (the results are shown in Figure 9). If the curves are closer to the reference line, it indicates that the model weights are more like full-rank, *i.e.,* contain less redundancy.

From Figure 9, we can see that to approximate the major information of the weight matrix, *e.g.,* preserving 80% of the singular value information, relatively higher $\mathcal{R}$s should be used, *e.g.,* $\rho = \frac{1}{2}$. Additionally, the attention weights, such as $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$, as well as the projection layer after the query, key, and value layers (*i.e.,* $\mathbf{W}^o$ in our notation) tend to have lower ranks (higher redundancy) compared to the FFN layers, *i.e.,* FC1 and FC2 in Figure 9. In CUTTLEFISH, for DeiT and ResMLP models, we use a global rank ratio $\rho = \frac{1}{2}$ for all factorized layers. It is worth noting that the linear projection layer after each $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$ has dimensions of $(768, 768)$. Using ρ for this layer, the factorized $\mathbf{U}, \mathbf{V}^\top$ layers will have dimensions of $(768, 384), (384, 768)$, which will not result in any model size reduction or computational complexity savings. Thus, we opt not to factorize the linear projection layers in each multi-head attention layer in DeiT-base. For ResMLP-S36, we factorize all layers except for the embedding layers with a fixed global rank ratio of $\rho = \frac{1}{2}$.

A concern arises from the fact that our proposed stable rank selection heuristic for choosing $\mathcal{R}$ may not generally apply to both CNN and Transformer models, as Transformer model weights tend to have higher ranks. To address this issue, future work can adjust CUTTLEFISH's rank selection heuristic to:

$$\max\{\texttt{scaled stable rank}(\Sigma),\\ \texttt{accumulative rank}(\Sigma, p)\}$$

Here, $\texttt{accumulative rank}(\Sigma, p)$ measures the smallest rank value r such that (where σs represent the singular values of a model weight matrix $\mathbf{W}$, and are also the elements on the diagonal of matrix Σ):

$$\sum_{i=1}^{r} \sigma_i \geq p \cdot \sum_{j=1}^{\text{rank}(\mathbf{W})} \sigma_j .$$

In the DeiT example mentioned earlier, we know that the $\texttt{accumulative rank}(\Sigma, 80\%)$ for most model layers is generally greater than $\frac{1}{2} \times \text{rank}(\mathbf{W})$ and the scaled stable rank for these layers is generally lower than those values. Consequently, the new metric defined above will consistently return $\frac{1}{2} \times \text{rank}(\mathbf{W})$ for all factorized layers in the DeiT-base model. Another hyperparameter tuning we performed in our experiments is decaying the base learning rate by a certain fraction after switching from full-rank to low-rank training at epoch $\hat{E}$. For CUTTLEFISH DeiT-base, we decay the base learning rate by $\frac{1}{3}$. For CUTTLEFISH ResMLP-S36, we decay the base learning rate by $\frac{1}{2}$.

CUTTLEFISH begins factorizing layers after the first embedding layer, which is simply a convolution layer, *i.e.,* $K = 1$ for both DeiT-base and ResMLP-S36. Since we tune K for PUFFERFISH such that the end model sizes match CUTTLEFISH DeiT and ResMLP, PUFFERFISH starts factorizing layers from the 7th encoder block for DeiT, *i.e.,* $K = 19$ (6 blocks, 3 layers in each block), and the 18th ResMLP block, *i.e.,* $K = 52$ (17 blocks, 3 layers in each block).

GLUE Benchmark on BERT$_{\text{BASE}}$. For BERTBASE fine-tuning on the GLUE benchmark, the fine-tuning epochs for all downstream tasks are typically small, such as $T = 3, 5$. Therefore, we set $E = 1$ for CUTTLEFISH. Additionally, we free the fully connected layers following each multi-head attention layer when fine-tuning the factorized BERT$_{\text{BASE}}$. As in LoRA, during the fine-tuning stage, we do not update the feed-forward network (FFN) in

Table 8. The hyperparameters $\hat{s} \in \mathcal{S}$ optimized by CUTTLEFISH for ResNet-18 and VGG-19 models trained on CIFAR-10 use a batch size of 1,024. The runtime benchmark was conducted on a single EC2 p3.2xlarge instance, equipped with one V100 GPU.

	CIFAR-10		CIFAR-100		SVHN	
Model: ResNet-18	E	K	E	K	E	K
CUTTLEFISH	$82.3_{\pm 10.1}$	5	$55.7_{\pm 8.7}$	5	$61.0_{\pm 2.2}$	5
PUFFERFISH	80	3	80	3	80	3
SI&FD	0	1	0	1	0	1
Model: VGG-19	E	K	E	K	E	K
CUTTLEFISH	$97.3_{\pm 1.2}$	4	$86.0_{\pm 5.7}$	4	$84.0_{\pm 0.8}$	4
PUFFERFISH	80	9	80	9	80	9
SI&FD	0	1	0	1	0	1

Table 9. The hyperparameters $\hat{s} \in \mathcal{S}$ obtained by CUTTLEFISH, as well as the manually tuned s from PUFFERFISH, for ResNet-50 and WideResNet-50-2 trained on ImageNet using a batch size of 256.

	ImageNet	
Model: ResNet-50	E	K
CUTTLEFISH	$19.3_{\pm 0.5}$	40
PUFFERFISH	10	40
Model: WideResNet-50-2	E	K
CUTTLEFISH	$21.3_{\pm 0.5}$	40
PUFFERFISH	10	40

BERT at all; we freeze the FC1 and FC2 layers contained in the FFN (Hu et al., 2021). We perform learning rate sweeping over the learning rates γs for all methods during GLUE fine-tuning. The tuned learning rates are shown in Table 11. For the relatively challenging CoLA task, we fine-tune all models except for the vanilla BERTBASE for 5 epochs instead of 3 epochs. For the RTE task, we fine-tune Distill BERT for 5 epochs.

C.3 Hyperparameters for other baselines.

SI&FD. We adjust the fixed global rank ratios, denoted as ρs, for SI&FD so that the resulting model sizes align with the factorized low-rank models produced by CUTTLEFISH. Detailed information on the ρs used in our experiments can be found in Table 12.

LC compression. The implementation and hyperparameter configurations for our experiments are taken directly from the original GitHub repository[6] associated with (Idelbayev & Carreira-Perpinán, 2020). We modified the VGG-19 model implementation in the LC compression setup to ensure consistency with the VGG-19 architecture used in our experiments.

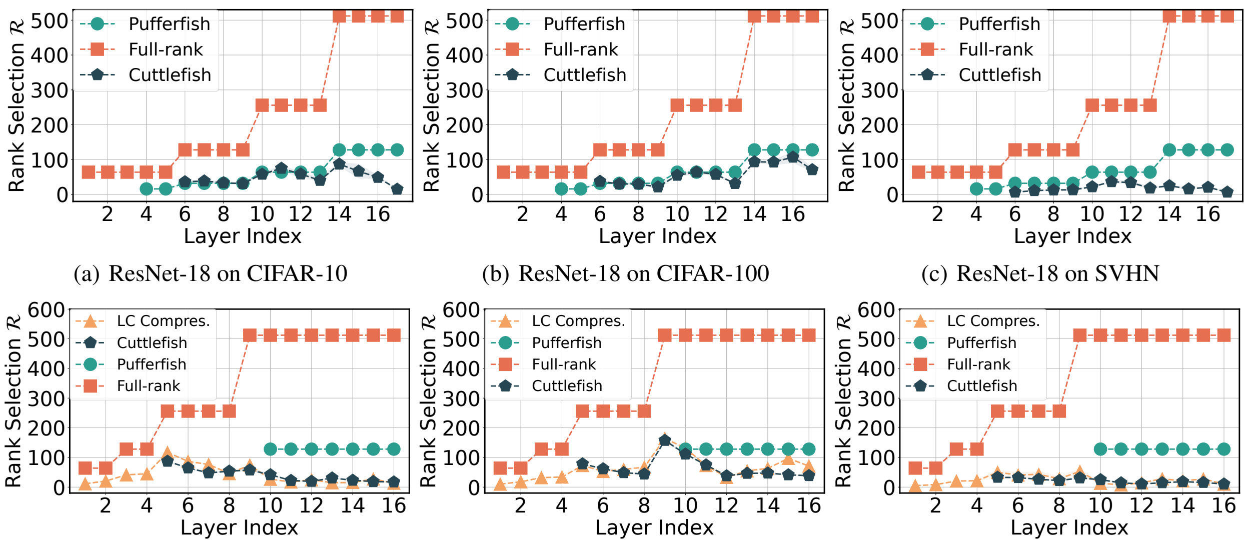
[6] https://github.com/UCMerced-ML/LC-model-compression

D ADDITIONAL EXPERIMENTAL RESULTS

D.1 Ablation study.

Combining CUTTLEFISH with Frobenius decay. In this section, we present the results of an ablation study examining the combination of *Frobenius decay* (FD) with CUTTLEFISH across various machine learning tasks. The results can be found in Table 13 and Table 14. It is evident that applying FD to CUTTLEFISH does not consistently lead to better model accuracy. For instance, combining CUTTLEFISH with FD yields a 1.8% higher accuracy for ResNet-18 training on CIFAR-100. However, for other tasks, incorporating FD with ResNet-18 either results in worse final model accuracy or only marginal accuracy improvements. This observation aligns with the findings of (Vodrahalli et al., 2022), indicating that FD does not always enhance the accuracy of factorized low-rank models.

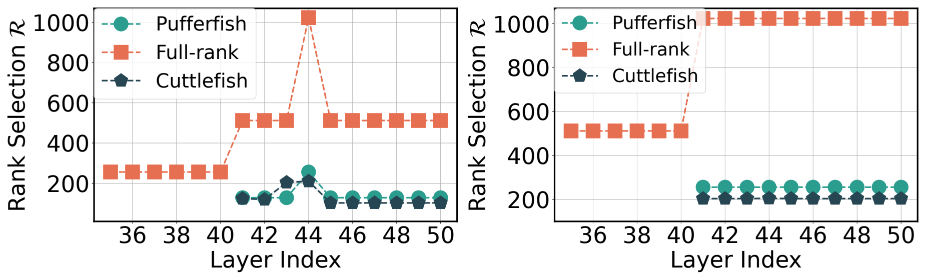
The impact of scaled stable rank. As mentioned in the main paper, the use of *stable rank* can lead to overly aggressive low rank estimations, potentially harming the final accuracy of factorized low-rank models. To address this issue, CUTTLEFISH employs scaled stable rank. The ablation study results are presented in Table 15. We find that for CIFAR-10 and CIFAR-100 datasets, utilizing scaled stable rank is crucial for achieving satisfactory final accuracy in factorized low-rank networks. For SVHN, which is a comparatively simpler task, even the vanilla stable rank is



(a) ResNet-18 on CIFAR-10 (b) ResNet-18 on CIFAR-100 (c) ResNet-18 on SVHN

(d) VGG-19 on CIFAR-10 (e) VGG-19 on CIFAR-100 (f) VGG-19 on SVHN

Figure 7. The ranks ($\mathcal{R}$) determined by CUTTLEFISH, PUFFERFISH, and LC compression (available only for VGG-19 experiments) for various layers in ResNet-18 ((a), (b), (c)) and VGG-19 ((d), (e), (f)) were trained on CIFAR-10 using a batch size of 1,024.



(a) ResNet-50 on ImageNet (b) WideResNet-50-20 on ImageNet

Figure 8. The ranks $\mathcal{R}$s obtained by CUTTLEFISH and PUFFERFISH methods for different layers in ResNet-50 (a) and WideResNet-50-2 (b) trained on ImageNet with a batch size of 256.

sufficient to attain good accuracy. Thus, in our main paper's reported experiment, we use vanilla stable rank for SVHN and scaled stable rank for CIFAR-10 and CIFAR-100. For larger scale tasks on ImageNet, it is evident that adopting scaled stable rank is essential; otherwise, the model accuracy will suffer a significant drop, as shown in Table 16.

D.2 Additional experimental results.

BERT pre-training using CUTTLEFISH. We perform BERT pre-training on the Wikipedia and Bookcorpus datasets, adhering to the training schedule and codebase of the 24h $\text{BERT}_{\text{LARGE}}$ for faster training speed and due to limited computing resources (Izsak et al., 2021). The results, shown in Table 17, indicate that CUTTLEFISH enables pre-training a BERT model with only 72% of the total model parameters while achieving the same final MLM loss.

Rank varying trend of other datasets. In our main paper, we presented the rank varying trend only for ResNet-18 on CIFAR-10. In this section, we expand our analysis and report additional experimental results on rank varying trends. Specifically, we provide results for VGG-19 trained on CIFAR-10, as well as VGG-19 and ResNet-18 on CIFAR-100 and SVHN datasets. For ResNet-50 on ImageNet, we also show the same results. Figures 10 and 14 display the results for VGG-19 on CIFAR-10, while Figures 11, 15, 16, 13, and 17 illustrate the results for the remaining datasets.

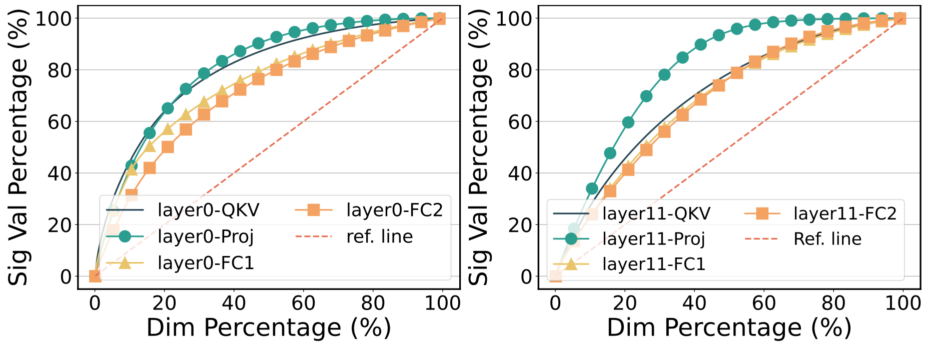
Overall, our observations indicate that the stable rank of the network layers fluctuates significantly in the early stages of training but eventually converges to a constant value. This trend holds across all the datasets we analyzed.

Comparison of CUTTLEFISH to EB Train and GraSP. Furthermore, we compare CUTTLEFISH with two other state-of-the-art approaches, namely EB Train and GraSP (as shown in Table 18). Notably, CUTTLEFISH outperforms both EB Train and GraSP in terms of accuracy while achieving smaller model sizes.

ResNet-18 and VGG-19 Experiments on SVHN. The results of ResNet-18 and VGG-19 experiments on the SVHN dataset are presented in Table 19.

Table 10. The tuned hyperparameters $\hat{s} \in \mathcal{S}$, as determined by CUTTLEFISH, and the manually tuned s from PUFFERFISH for DeiT-base and ResMLP-S36, trained on the ImageNet dataset using a batch size of 256.

	ImageNet	
Model: DeiT-base	E	K
CUTTLEFISH	$59.5_{\pm 6.5}$	1
PUFFERFISH	80	19
Model: ResMLP-S36	E	K
CUTTLEFISH	$40.5_{\pm 1.5}$	1
PUFFERFISH	80	52



(a) Encoder 0 of DeiT-base (b) Encoder 11 of DeiT-base

Figure 9. The Cumulative Distribution Function (CDF) of singular values for the first Transformer encoder (*i.e.,* Encoder 0, denoted as layer0 in the figure) (a) and the last Transformer encoder (*i.e.,* Encoder 11, denoted as layer11 in the figure) (b) of DeiT-base trained on the ImageNet dataset using a batch size of 256. Other Transformer encoders exhibit similar trends.

Table 11. Tuned learning rates, denoted as γs, for vanilla BERT$_{\text{BASE}}$, Distill BERT, Tiny BERT6, and CUTTLEFISH BERTBASE on the GLUE benchmark.

Model	# Params. (M)	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B
BERT$_{\text{BASE}}$	108.3	2e-5	2e-5	2e-5	4e-5	2e-5	2e-5	4e-5	2e-5
Distill BERT	65.8	2e-5	2e-5	2e-5	2e-5	4e-5	2e-5	2e-5	2e-5
Tiny BERT$_6$	67.0	2e-5	2e-5	2e-5	2e-5	2e-5	2e-5	2e-5	2e-5
CUTTLEFISH	**48.8**	2e-5	2e-5	2e-5	3e-5	3e-5	2e-5	3e-5	2e-5

Table 12. The fixed rank ratios (ρs) employed in SI&FD experiments for CIFAR-10, CIFAR-100, and SVHN on ResNet-18 and VGG-19.

Model: ResNet-18	CIFAR-10	CIFAR-100	SVHN
SI&FD	0.08	0.105	0.032
Model: VGG-19	CIFAR-10	CIFAR-100	SVHN
SI&FD	0.1	0.165	0.059

Table 13. The ablation study results, averaged across three independent trials with different random seeds, showcase the performance of CUTTLEFISH combined with Frobenius decay (FD) on ResNet-18 and VGG-19 trained on CIFAR-10 using a batch size of 1,024.

	CIFAR-10		CIFAR-100		SVHN	
Model: ResNet-18	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)
CUTTLEFISH wo. FD	2.0	$94.52_{\pm 0.01}$	2.6	$73.75_{\pm 0.24}$	0.96	$\mathbf{96.47}_{\pm 0.02}$
CUTTLEFISH w. FD	2.0	$\mathbf{94.62}_{\pm 0.09}$	2.6	$\mathbf{75.54}_{\pm 0.18}$	0.94	$96.34_{\pm 0.08}$
Model: VGG-19	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)
CUTTLEFISH wo. FD	1.9	$\mathbf{93.49}_{\pm 0.18}$	3.3	$\mathbf{72.27}_{\pm 0.25}$	1.2	$\mathbf{96.33}_{\pm 0.04}$
CUTTLEFISH w. FD	1.9	$93.42_{\pm 0.25}$	3.3	$72.15_{\pm 0.27}$	1.2	$\mathbf{96.33}_{\pm 0.02}$

Table 14. The ablation study results, averaged across three independent trials, demonstrate the performance of CUTTLEFISH combined with Frobenius decay (FD) on ResNet-50 trained on ImageNet using a batch size of 256.

Model: ResNet-50	# Params. (M)	Top-1 Val. Acc. (%)	Top-5 Val. Acc. (%)
CUTTLEFISH wo. FD	14.7	$76.16_{\pm 0.04}$	$92.97_{\pm 0.06}$
CUTTLEFISH w. FD	14.7	$\mathbf{76.44}_{\pm 0.16}$	$\mathbf{93.21}_{\pm 0.03}$

Table 15. The ablation study results (averaged over 3 independent trials with varying random seeds) for CUTTLEFISH using both scaled stable rank and vanilla stable rank on ResNet-18 and VGG-19, trained on CIFAR-10, CIFAR-100, and SVHN with a batch size of $1,024$.

	CIFAR-10		CIFAR-100		SVHN	
Model: ResNet-18	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)
CUTTLEFISH vanilla stable rank	1.2	$94.27_{\pm 0.05}$	1.6	$74.21_{\pm 0.20}$	0.94	$96.47_{\pm 0.02}$
CUTTLEFISH scaled stable rank	2.0	$\mathbf{94.62}_{\pm 0.09}$	2.6	$\mathbf{75.54}_{\pm 0.18}$	1.4	$\mathbf{96.53}_{\pm 0.08}$
Model: VGG-19	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)	# Params. (M)	Val. Acc. (%)
CUTTLEFISH vanilla stable rank	1.1	$93.07_{\pm 0.10}$	1.9	$70.62_{\pm 0.23}$	1.2	$96.33_{\pm 0.04}$
CUTTLEFISH scaled stable rank	1.9	$\mathbf{93.49}_{\pm 0.18}$	3.3	$\mathbf{72.27}_{\pm 0.25}$	2.0	$\mathbf{96.42}_{\pm 0.07}$

Table 16. The ablation study results (averaged over 3 independent trials with distinct random seeds) for CUTTLEFISH using both scaled stable rank and vanilla stable rank on DeiT-base, ResNet-50, and WideResNet-50, trained on ImageNet with a batch size of 256.

	# Params. (M)	Top-1 Val. Acc.	Top-5 Val. Acc.
CUTTLEFISH DeiT-base vanilla stable rank	12.5	$64.80_{\pm 0.82}$	$85.46_{\pm 0.60}$
CUTTLEFISH DeiT-base scaled stable rank	58.3	$\mathbf{81.52}_{\pm 0.03}$	$\mathbf{95.59}_{\pm 0.04}$
CUTTLEFISH WideResNet-50 vanilla stable rank	29.1	$76.86_{\pm 0.01}$	$93.50_{\pm 0.03}$
CUTTLEFISH WideResNet-50 scaled stable rank	37.4	$\mathbf{78.0}_{\pm 0.06}$	$\mathbf{94.04}_{\pm 0.09}$
CUTTLEFISH ResNet-50 vanilla stable rank	11.9	$74.96_{\pm 0.01}$	$92.39_{\pm 0.07}$
CUTTLEFISH ResNet-50 scaled stable rank	14.7	$\mathbf{76.44}_{\pm 0.16}$	$\mathbf{93.21}_{\pm 0.03}$

Table 17. Vanilla and CUTTLEFISH BERT pre-training on Wikipedia and Bookcorpus datasets.

Model	# Params. (M)	MLM Loss
Vanilla BERT$_{\text{LARGE}}$	345	1.58
Cuttlefish BERT$_{\text{LARGE}}$	249	1.6

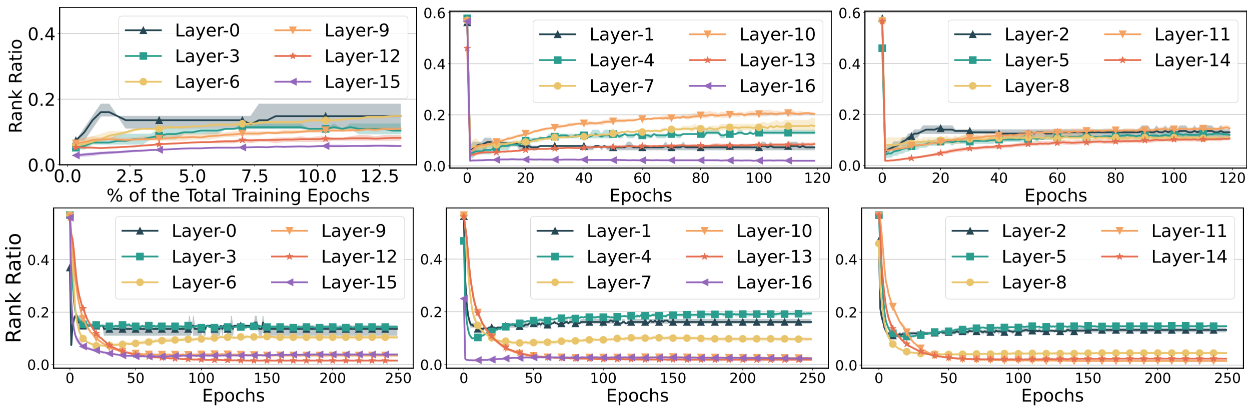


Figure 10. The stable ranks for various layers in ResNet-18 (**the top row**) and VGG-19 (**the bottom row**) trained on CIFAR-10 using stable rank.

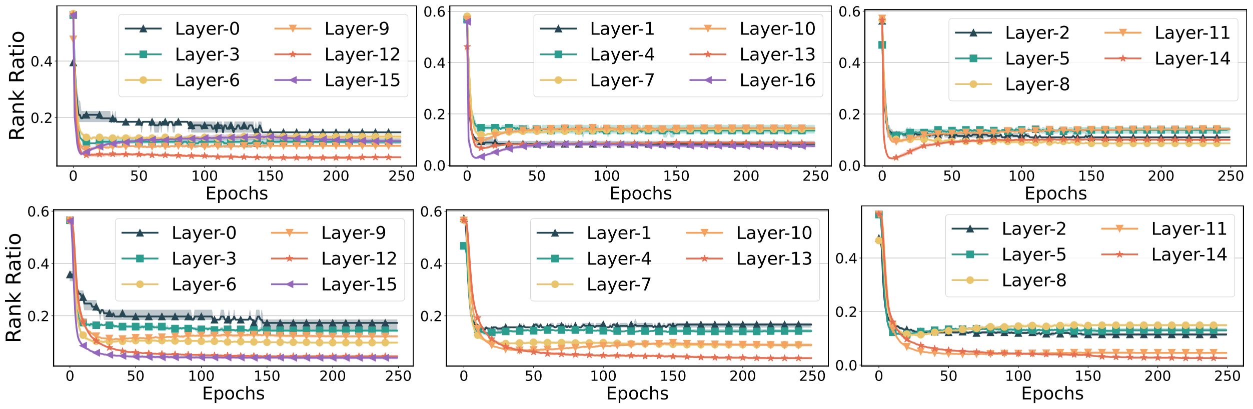


Figure 11. The stable ranks for various layers in ResNet-18 (**the top row**) and VGG-19 (**the bottom row**) trained on CIFAR-100 using stable rank with batch size $1,024$.

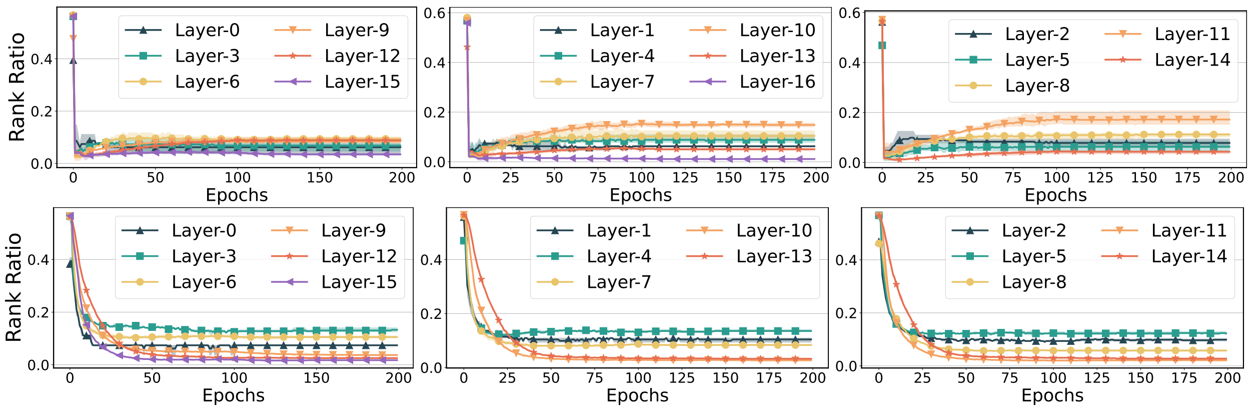


Figure 12. The stable ranks for various layers in ResNet-18 (**the top row**) and VGG-19 (**the bottom row**) trained on SVHN using stable rank with batch size $1,024$.

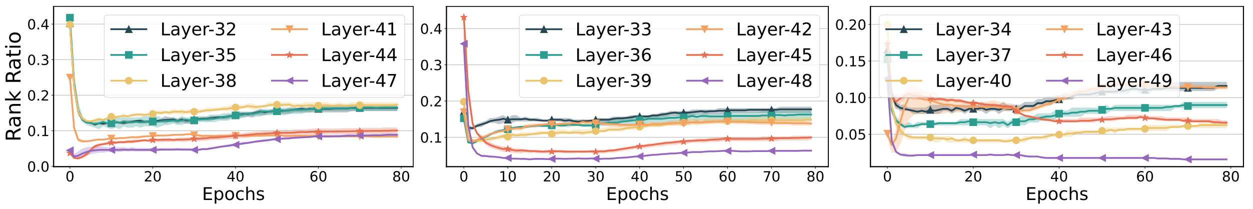
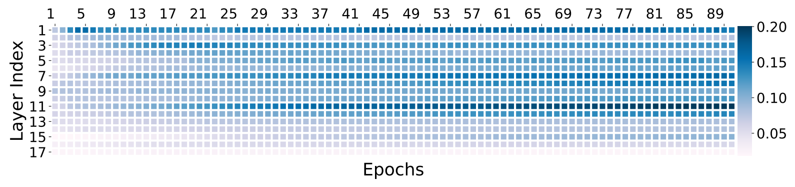
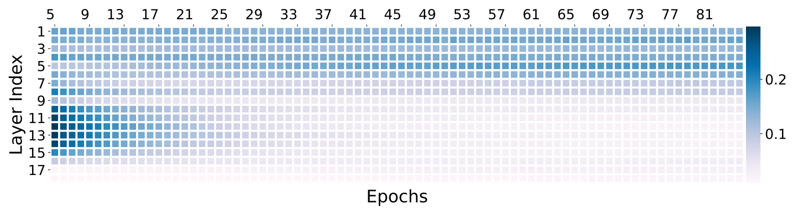


Figure 13. The stable ranks for various layers in ResNet-50 trained on ImageNet using stable rank with batch size 256.



(a) ResNet-18

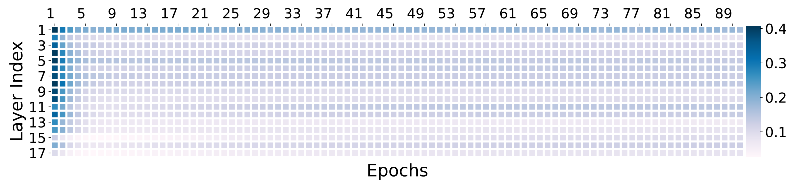


(b) VGG-19

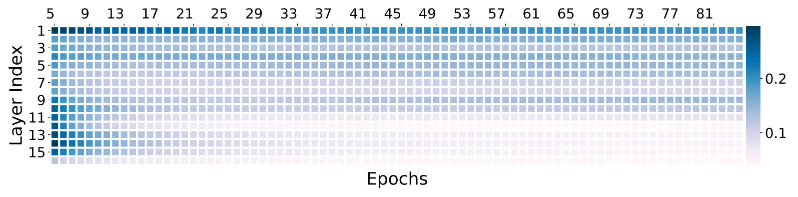
Figure 14. The stable ranks for various layers in ResNet-18 and VGG-19 trained on CIFAR-10 using stable rank.

Table 18. Comparison of CUTTLEFISH and other baseline methods: PUFFERFISH, EB Train (30%, 50%) and GraSP (30%, 60%) over the task of ResNet-50 on ImageNet.

Model ResNet-50	# Params. (M)	Val. Acc. Top-1(%)	Val. Acc. Top-5(%)
Full-rank	25.6	75.99	92.98
PUFFERFISH	15.2	75.62	92.55
EB Train (30%)	16.5	73.86	91.52
EB Train (50%)	15.1	73.35	91.36
GraSP (30%)	17.9	74.64	92.08
GraSP (60%)	10.2	74.02	91.86
CUTTLEFISH	14.7	75.80	92.70

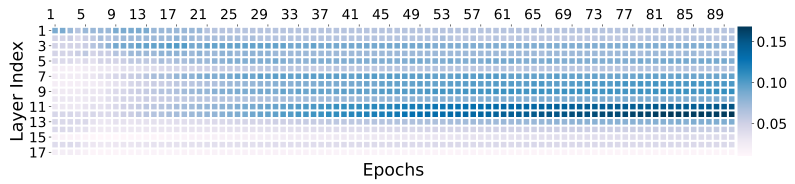


(a) ResNet-18



(b) VGG-19

Figure 15. The stable ranks for various layers in ResNet-18 and VGG-19 trained on CIFAR-100 using stable rank.


(a) ResNet-18

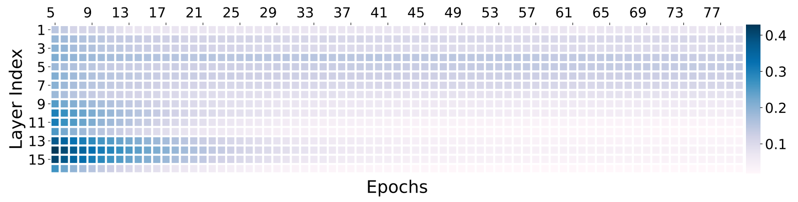

(b) VGG-19

Figure 16. The stable ranks for various layers in ResNet-18 and VGG-19 trained on SVHN using stable rank.

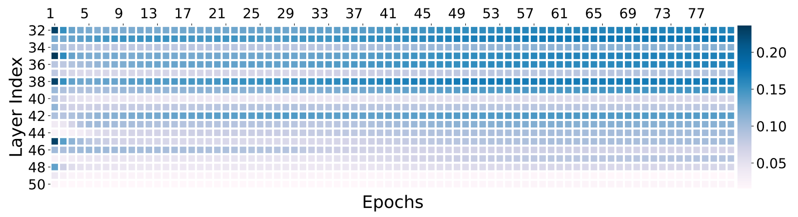


Figure 17. The stable ranks for various layers in ResNet-50 trained on ImageNet using stable rank.

Table 19. The results, averaged over three independent trials with different random seeds, showcase the performance of CUTTLEFISH and other baselines on ResNet-18 and VGG-19 trained on the SVHN dataset using a batch size of 1,024. Runtime benchmarks are conducted on a single EC2 p3.2xlarge instance.

Model: ResNet-18	Params. (M)	Val. Acc. (%)	Time (hrs.)
Full-rank	11.2	$96.27_{\pm 0.08}$	0.81
PUFFERFISH	3.3	$96.54_{\pm 0.06}$	0.71
SI&FD	0.94	$96.45_{\pm 0.04}$	0.38
IMP	0.96	$96.43_{\pm 0.01}$	9.40
CUTTLEFISH	0.96	$96.47_{\pm 0.02}$	0.65
CUTTLEFISH+FD	0.94	$96.34_{\pm 0.08}$	0.65
Model: VGG-19	Params. (M)	Val. Acc. (%)	Time (hrs.)
Full-rank	20.0	$96.31_{\pm 0.05}$	0.49
PUFFERFISH	8.1	$96.08_{\pm 0.11}$	0.45
SI&FD	1.2	$96.04_{\pm 0.16}$	0.30
FC Compress.	1.1	$96.42_{\pm 0.07}$	6.68
CUTTLEFISH	1.2	$96.33_{\pm 0.04}$	0.39
CUTTLEFISH+FD	1.2	$96.33_{\pm 0.02}$	0.39