**PUFFERFISH: COMMUNICATION-EFFICIENT MODELS AT NO EXTRA COST**

Hongyi Wang, Saurabh Agarwal, Dimitris Papailiopoulos

**ABSTRACT**

To mitigate communication overheads in distributed model training, several studies propose the use of compressed stochastic gradients, usually achieved by sparsification or quantization. Such techniques achieve high compression ratios, but in many cases incur either significant computational overheads or some accuracy loss. In this work, we present PUFFERFISH, a communication and computation efficient distributed training framework that incorporates the gradient compression into the model training process via training low-rank, pre-factorized deep networks. PUFFERFISH not only reduces communication, but also completely bypasses any computation overheads related to compression, and achieves the same accuracy as state-of-the-art, off-the-shelf deep models. PUFFERFISH can be directly integrated into current deep learning frameworks with minimum implementation modification. Our extensive experiments over real distributed setups, across a variety of large-scale machine learning tasks, indicate that PUFFERFISH achieves up to 1.64× end-to-end speedup over the latest distributed training API in PyTorch without accuracy loss. Compared to the *Lottery Ticket Hypothesis* models, PUFFERFISH leads to equally accurate, small-parameter models while avoiding the burden of “winning the lottery”. PUFFERFISH also leads to more accurate and smaller models than SOTA structured model pruning methods.

**1 INTRODUCTION**

Distributed model training plays a key role in the success of modern machine learning systems. Data parallel training, a popular variant of distributed training, has demonstrated massive speedups in real-world machine learning applications and systems (Li et al., 2014; Dean et al., 2012; Chen et al., 2016a). Several machine learning frameworks such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019) come with distributed implementations of popular training algorithms, such as mini-batch SGD. However, the empirical speed-ups offered by distributed training, often fall short of a best-case linear scaling. It is now widely acknowledged that communication overheads are one of the key sources of this saturation phenomenon (Dean et al., 2012; Seide et al., 2014; Strom, 2015; Qi et al., 2017; Grubic et al., 2018).

Communication bottlenecks are attributed to frequent gradient updates, transmitted across compute nodes. As the number of parameters in state-of-the-art (SOTA) deep models scales to hundreds of billions, the size of communicated gradients scales proportionally (He et al., 2016; Hu et al., 2017; Devlin et al., 2018; 2019; Brown et al., 2020). To reduce the cost of communicating model updates, recent studies propose compressed versions of the computed gradients. A large number of recent studies revisited the idea of low-precision training as a means to reduce communication (Seide et al., 2014; De Sa et al., 2015; Alistarh et al., 2017; Zhou et al., 2016; Wen et al., 2017; Zhang et al., 2017; De Sa et al., 2017; 2018; Bernstein et al., 2018a; Konečný et al., 2016). Other approaches for low-communication training focus on sparsification of gradients, either by thresholding small entries or by random sampling (Strom, 2015; Mania et al., 2015; Suresh et al., 2016; Leblond et al., 2016; Aji & Heafield, 2017; Konečný & Richtárik, 2016; Lin et al., 2017; Chen et al., 2017; Renggli et al., 2018; Tsuzuku et al., 2018; Wang et al., 2018; Vogels et al., 2019).

However, the proposed communication-efficient training techniques via gradient compression usually suffer from some of the following drawbacks: (i) The computation cost for gradient compression (e.g., sparsification or quantization) can be high. For instance, ATOMO (Wang et al., 2018) requires to compute gradient factorizations using SVD for every single batch, which can be computationally expensive for large-scale models. (ii) Existing gradient compression methods either do not fully utilize the full gradients (Alistarh et al., 2017; Wen et al., 2017; Bernstein et al., 2018a; Wang et al., 2018) or require additional memory. For example, the “error feedback” scheme (Seide et al., 2014; Stich et al., 2018; Karimireddy et al., 2019) utilizes stale gradients aggregated in memory for future iterations, but requires storing additional information pro-
portional to the model size. (iii) Significant implementation efforts are required to incorporate an existing gradient compression technique within high-efficiency distributed training APIs in current deep learning frameworks e.g., 

**DistributedDataParallel (DDP)** in PyTorch.

Due to the above shortcomings of current communication-efficient techniques, it is of interest to explore the feasibility of incorporating elements of the gradient compression step into the model architecture itself. If this is feasible, then communication efficiency can be attained at no extra cost. In this work, we take a first step towards bypassing the gradient compression step via training low-rank, pre-factorized deep network, starting from full-rank counterparts. We observe that training low-rank models from scratch incurs non-trivial accuracy loss. To mitigate that loss, instead of starting from a low-rank network, we initialize at a full-rank counterpart. We train for a small fraction, e.g., 10% of total number epochs, with the full-rank network, and then convert to a low-rank counterpart. To obtain such a low-rank model we apply SVD on each of the layers. After the SVD step, we use the remaining 90% of the training epochs to fine-tune this low-rank model. The proposed method bares similarities to the “Lottery Ticket Hypothesis” (LTH) (Frankle & Carbin, 2018), in that we find “winning tickets” within full-rank/dense models, but without the additional burden of “winning the lottery”. Winning tickets seem to be in abundance once we seek models that are sparse in their spectral domain.

**Our contributions.** In this work, we propose **PUFFERFISH**, a computation and communication efficient distributed training framework. **PUFFERFISH** takes any deep neural network architecture and finds a pre-factorized low-rank representation. **PUFFERFISH** then trains the pre-factorized low-rank network to achieve both computation and communication efficiency, instead of explicitly compressing gradients. **PUFFERFISH** supports several types of architectures including fully connected (FC), convolutional neural nets (CNNs), LSTMs, and Transformer networks (Vaswani et al., 2017). As **PUFFERFISH** manipulates the model architectures instead of their gradients, it is directly compatible with all SOTA distributed training frameworks, e.g., PyTorch DDP and BytePS (Jiang et al., 2020).

We further observe that direct training of those pre-factorized low-rank deep networks leads to non-trivial accuracy loss, especially for large-scale machine learning tasks, e.g., ImageNet (Deng et al., 2009). We develop two techniques for mitigating this accuracy loss: (i) a **hybrid architecture** and (ii) **vanilla warm-up training**. The effectiveness of these two techniques is justified via extensive experiments.

We provide experimental results over real distributed systems and large-scale vision and language processing tasks.

**Figure 1.** We propose to replace fully connected layers represented by a matrix $W$, by a set of trainable factors $UV^T$, and represent each of the $N$ convolutional filters of each conv layer as a linear combination of $\frac{N}{S}$ filters. This latter operation can be achieved by using fewer filters per layer, and then applying a trainable up-sampling embedding to the output channels.

We compare **PUFFERFISH** against a wide range of SOTA baselines: (i) communication-efficient distributed training methods e.g., **POWERSGD** (Vogels et al., 2019) and **SIGNUM** (Bernstein et al., 2018a); (ii) structured pruning methods, e.g., the **Early Bird Ticket** (EB Train) (You et al., 2019); and model sparsification method, e.g., the iterative pruning algorithm in LTH (Frankle & Carbin, 2018). Our experimental results indicate that **PUFFERFISH** achieves better model training efficiency compared to **POWERSGD**, **SIGNUM**, and LTH models. **PUFFERFISH** also leads to smaller and more accurate model compared to EB Train. We further show that the performance of **PUFFERFISH** remains stable under mixed-precision training.

**Related work.** **PUFFERFISH** is closely related to the work on communication-efficient distributed training methods. To reduce the communication cost in distributed training, the related literature has developed several methods for gradient compression. Some of the methods use quantization over the gradient elements (Seide et al., 2014; Alistarh et al., 2017; Wen et al., 2017; Lin et al., 2017; Luo et al., 2017; Bernstein et al., 2018a; Tang et al., 2019; Wu et al., 2018). Other methods study sparsifying the gradients in the element-wise or spectral domains (Lin et al., 2017; Wang et al., 2018; Stich et al., 2018; Vogels et al., 2019). It has also been widely observed that adopting the “error feedback” scheme is generally helpful for gradient compression methods to achieve better final model accuracy (Stich et al., 2018; Wu et al., 2018; Karimireddy et al., 2019; Vogels et al., 2019). Compared to the previously proposed gradient compression methods, **PUFFERFISH** merges the gradient compression into model training, thus achieves communication-efficiency at no extra cost.

**PUFFERFISH** is also closely related to model compression. Partially initialized by deep compression (Han et al., 2015), a lot of research proposes to remove the redundant weights in the trained neural networks. The trained neural networks
can be compressed via model weight 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., 2018a,b), 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), and low-rank factorization (Xue et al., 2013; Sainath et al., 2013; Jaderberg et al., 2014; Wiesler et al., 2014; Konečný et al., 2016). Different from model compression methods, PUFFERFISH proposes to train the factorized networks, which achieves better overall training time, rather than compressing the model after fully training it.

Finally, our work is also related to efficient network architecture design, where the network layers are re-designed to be smaller, more compact, and more efficient (Iandola et al., 2016; Chen et al., 2016b; Zhang et al., 2018; Tan & Le, 2019; Howard et al., 2017; Chollet, 2017; Lan et al., 2019; Touvron et al., 2020; Waleffe & Rekatsinas, 2020). The most related low-rank efficient training framework to PUFFERFISH is the one proposed in (Ioannou et al., 2015), where a pre-factorized network is trained from scratch. However, we demonstrate that training the factorized network from scratch leads to non-trivial accuracy loss. In PUFFERFISH, we propose to warm-up the low-rank model via factorizing a partially trained full-rank model. Our extensive experiments indicate that PUFFERFISH achieves significantly higher accuracy compared to training the factorized network from scratch. Moreover, (Ioannou et al., 2015) only studies low-rank factorizations for convolutional layers, whereas PUFFERFISH supports FC, CNN, LSTM, and Transformer layers.

2 PUFFERFISH: EFFECTIVE DEEP FACTORIZED NETWORK TRAINING

In the following subsections, we discuss how model factorization is implemented for different model architectures.

2.1 Low-rank factorization for FC layers

For simplicity, we discuss a 2-layer FC network that can be represented as \( h(x) = \sigma(W_1 \sigma(W_2x)) \) where \( W_i, \forall i \in \{1, 2\} \) are weight matrices, \( \sigma(\cdot) \) is an arbitrary activation function, and \( x \) is the input data point. We propose to pre-factorize the matrices \( W_i \) into \( U_iV_i^T \) where the factors are of significantly smaller dimensions while also reducing the computational complexity of the full-rank FC layer.

2.2 Low-rank factorization for convolution layers

Basics on convolution layers. The above low-rank factorization strategy extends to convolutional layers (see Fig. 1 for a sketch). In a convolution layer, a \( c_{in} \)-channel input image of size \( H \times W \) pixels is convolved with \( c_{out} \) filters of size \( c_{in} \times k \times k \) to create a \( c_{out} \)-channel output feature map. Therefore, the computational complexity for the convolution of the filter with a \( c_{in} \)-channel input image is \( O(c_{in}c_{out}k^2HW) \).

In what follows, we describe schemes for modifying the architecture of the convolution layers via low-rank factorization to reduce computational complexity and the number of parameters. The idea is to replace vanilla (full-rank) convolution layers with factorized versions. These factorized filters amount to the same number of convolution filters, but are constructed through linear combinations of a sparse, i.e., low-rank filter basis.

Factorizing a convolution layer. For a convolution layer with dimension \( W \in \mathbb{R}^{c_{in} \times c_{out} \times k \times k} \) where \( c_{in} \) and \( c_{out} \) are the number of input and output channels and \( k \) is the size of the convolution filters, e.g., \( k = 3 \) or 5. Instead of factorizing the 4D weight of a convolution layer directly, we consider factorizing the unrolled 2D matrix. Unrolling the 4D tensor \( W \) leads to a 2D matrix with shape \( W_{unrolled} \in \mathbb{R}^{c_{in}\times k\times c_{out}} \) where each column represents the weight of a vectorized convolution filter. The rank of the unrolled matrix is determined by \( \min\{c_{in}k^2, c_{out}\} \). Factorizing the unrolled matrix returns \( U \in \mathbb{R}^{c_{in}\times k^2 \times r}, V^T \in \mathbb{R}^{r \times c_{out}}, i.e., W_{unrolled} \approx UV^T \). Reshaping the factorized \( U, V^T \) matrices back to 4D filters leads to \( U_{unrolled} \in \mathbb{R}^{c_{in}\times r \times k\times k}, V_{unrolled} \in \mathbb{R}^{c_{in}\times c_{out}} \).

Therefore, factorizing a convolution layer returns a thinner convolution layer \( U \) with width \( r \), i.e., the number of convolution filters, and a linear projection layer \( V^T \). In other words, the full-rank original convolution filter bank is approximated by a linear combination of \( r \) basis filters. The \( V^T \)'s can also be represented by a \( 1 \times 1 \) convolution layer, e.g., \( V^T_{i} \in \mathbb{R}^{r \times c_{out} \times 1 \times 1} \), which is more natural for computer vision tasks as it operates directly on the spatial domain (Lin et al., 2013). In PUFFERFISH, we use the \( 1 \times 1 \) convolution for all \( V^T_{i} \) layers in the considered CNNs. One can also use tensor decomposition, e.g., the Tucker decomposition to directly factorize the 4D tensor weights (Tucker, 1966). In this work, for simplicity, we do not consider tensor decompositions.

2.3 Low-rank factorization for LSTM layers

LSTMs have been proposed as a means to mitigate the “vanishing gradient” issue of traditional RNNs (Hochreiter & Schmidhuber, 1997). The forward pass of an LSTM is as follows:

\[
\begin{align*}
  i_t &= \sigma(W_{ii}x_t + b_i + W_{hi}h_{t-1} + b_h) \\
  f_t &= \sigma(W_{if}x_t + b_f + W_{hf}h_{t-1} + b_f) \\
  g_t &= \tanh(W_{ig}x_t + b_g + W_{hg}h_{t-1} + b_g) \\
  o_t &= \sigma(W_{io}x_t + b_o + W_{ho}h_{t-1} + b_o) \\
  c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
  h_t &= o_t \odot \tanh(c_t).
\end{align*}
\]

\( h_t, c_t, x_t \) represent the hidden state, cell state, and input at time \( t \) respectively. \( h_{t-1} \) is the hidden state of the layer at
time \( t - 1 \). \( i_t, f_t, g_t, o_t \) are the input, forget, cell, and output gates, respectively. \( \sigma() \) and \( \odot \) denote the sigmoid activation function and the Hadamard product, respectively. The trainable weights are the matrices \( W_i \in \mathbb{R}^{h \times d}, W_h \in \mathbb{R}^{k \times h} \), where \( d \) and \( h \) are the embedding and hidden dimensions. Thus, similarly to the low-rank FC layer factorization, the factorized LSTM layer is represented by

\[
i_t = \sigma(U_i h_i^T x_t + b_i + U_{hi} h_{t-1} + b_{hi})
\]

\[
f_t = \sigma(U_f h_i^T x_t + b_f + U_{hf} h_{t-1} + b_{hf})
\]

\[
g_t = \tanh(U_g h_i^T x_t + b_g + U_{hg} h_{t-1} + b_{hg})
\]

\[
o_t = \sigma(U_o h_i^T x_t + b_o + U_{ho} h_{t-1} + b_{ho})
\]

\[
c_t = f_t \odot c_{t-1} + i_t \odot g_t
\]

\[
h_t = o_t \odot c_t
\]

### 2.4 Low-rank network factorization for Transformer

A Transformer layer consists of a stack of encoders and decoders (Vaswani et al., 2017). Both encoder and decoder contain three main building blocks, i.e., the multi-head attention layer, position-wise feed-forward networks (FFN), and positional encoding. A \( p \)-head attention layer learns \( p \) independent attention mechanisms on the input key \((K)\), value \((V)\), and queries \((Q)\) of each input token:

\[
\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \cdots, \text{head}_p)W^O
\]

where \( \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \).

In the above, \( W_i^Q, W_i^K, W_i^V, i \in \{1, \cdots, p\} \) are trainable weight matrices. The particular attention, referred to as “scaled dot-product attention”, is used in Transformers, i.e.,

\[
\text{Attention}(\hat{Q}, \hat{K}, \hat{V}) = \text{softmax} \left( \frac{\hat{Q}^T \hat{K}}{\sqrt{d}} \right) \hat{V}
\]

where \( \hat{Q} = QW_i^Q, \hat{K} = KW_i^K, \hat{V} = VW_i^V \). \( W^O \) projects the output of the multi-head attention layer to match the embedding dimension. Following (Vaswani et al., 2017), we assume the projected key, value, and query are embedded to \( pd \) dimensions, and are projected to \( d \) dimensions in the attention layer. In Transformer, a sequence of \( N \) input tokens are usually batched before passing to the model where each input token is embedded to a \( pd \) dimensional vector. Thus, dimensions of the inputs are \( Q, K, V \in \mathbb{R}^{N \times pd} \). The learnable weight matrices are \( W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{pd \times d}, W^O \in \mathbb{R}^{pd \times pd} \). The FFN in Transformer consists of two learnable FC layers: \( \text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \) where \( W_1 \in \mathbb{R}^{pd \times 4pd}, W_2 \in \mathbb{R}^{4pd \times pd} \) (the relationships between the notations in our paper and the original Transformer paper (Vaswani et al., 2017) are \( pd = d_{\text{model}}, d = d_k = d_v \), and \( d_{ff} = 4pd \)).

In PUFFERFISH, we factorize all learnable weight matrices in the multi-head attention and the FFN layers. We leave the positional encoding as is, since there are no trainable weights. For the bias term of each layer and the “Layer Normalization” weights, we use the vanilla weights directly, as they are represented by vectors.

### 2.5 Computational complexity and model size

A low-rank factorized network enjoys a smaller number of parameters and lower computational complexity. Thus, both the computation and communication efficiencies are improved, as the amount of communication is proportional to the number of parameters. We summarize the computational complexity and the number of parameters in the vanilla and low-rank FC, convolution, LSTM, and the Transformer layers in Table 1. We assume the FC layer has shape \( W = \mathbb{R}^{m \times n} \), the convolution layer has shape \( W_{\text{Conv}} = \mathbb{R}^{c_{\text{in}} \times c_{\text{out}} \times k \times k} \), the LSTM layer has shape \( W_i, W_h \in \mathbb{R}^{4h \times d}, \tilde{W}_h \in \mathbb{R}^{4h \times h} \) (where \( W_i \) and \( W_h \) is the concatenated input-hidden and hidden-hidden weight matrices), and the shapes of the model weights in the encoder of a Transformer follow the discussion in Section 2.4. For Transformers, we show the computational complexity of a single encoder block. We assume the low-rank layers have rank \( r \). As the computation across the \( p \) heads can be done in parallel, we report the computational complexity of a single attention head. Note that for the LSTM layer, our complexity analysis assumes the low-rank layer uses the same rank for the input-hidden weights \( W_i \) and the hidden-hidden weights \( W_h \). Similarly, for the Transformer layer, we assume the low-rank layer uses the same rank \( r \) for all \( W_i^Q, W_i^K, W_i^V, W^O \). Further details can be found in the Appendix.

| Networks       | # Params. | Computational Complexity                  |
|----------------|----------|------------------------------------------|
| Vanilla FC     | \( m \times n \) | \( O(mn) \)                             |
| Factorized FC  | \( r(m + n) \) | \( O(r(m + n)) \)                        |
| Vanilla Conv.  | \( c_{\text{in}} \times c_{\text{out}} \times k^2 \) | \( O(c_{\text{in}}c_{\text{out}}k^2HW) \) |
| Factorized Conv.| \( c_{\text{in}}rk^2 + r_{\text{out}} \) | \( O(c_{\text{in}}k^2HW + rHW_{\text{out}}) \) |
| Vanilla LSTM   | \( 4dh + h^2 \) | \( O(dh + h^2) \)                         |
| Factorized LSTM| \( 4dr + 12hr \) | \( O(dr + hr) \)                         |
| Vanilla Attention| \( 4p^2d^2 \) | \( O(Np^2d^2 + N^2d) \)                  |
| Factorized Attention| \( (3p + 5)pdr \) | \( O(rpdN + N^2d) \)                     |
| Vanilla FFN    | \( 8p^2d^2 \) | \( O(p^2d^2N) \)                         |
| Factorized FFN | \( 10pdr \) | \( O(rpdN) \)                           |
model accuracy loss is to only factorize the later layers. Moreover, for most of CNNs, the number of parameters in later layers dominates the entire network size. Thus, factorizing the later layers does not sacrifice the degree of model compression we can achieve. Specifically, for an $L$ layer network $\{W_1, W_2, \ldots, W_L\}$, factorizing every layer leads to $\{U_1, V_1^T, U_2, V_2^T, \ldots, U_L, V_L^T\}$. In the hybrid network architecture, the first $K-1$ layers are not factorized, i.e., $\{W_1, W_2, \ldots, W_{K-1}, U_K, V_K^T, \ldots, U_L, V_L^T\}$, where we define $K$ as the index of the first low-rank layer in a hybrid architecture. We treat $K$ as a hyper-parameter, which balances the model compression ratio and the final model accuracy. In our experiments, we tune $K$ for all models. The effectiveness of the hybrid network architecture is shown in Figure 3(a), from which we observe that the hybrid VGG-19 with $K = 9$ mitigates $\sim 6\%$ test accuracy loss.

Vanilla warm-up training. It has been widely observed that epochs early in training are critical for the final model accuracy (Jastrzebski et al., 2020; Keskar et al., 2016; Achille et al., 2018; Leclerc & Madry, 2020; Agarwal et al., 2020). For instance, sparsifying gradients in early training phases can hurt the final model accuracy (Lin et al., 2017). Similarly, factorizing the vanilla model weights in the very beginning of the training procedure can also lead to accuracy loss, which may be impossible to mitigate in later training epochs. It has also been shown that good initialization strate-
gies play a significant role in the final model accuracy (Zhou et al., 2020).

In this work, to mitigate the accuracy loss, we propose to use the partially trained vanilla, full-rank model weights to initialize the low-rank factorized network. We refer to this as “vanilla warm-up training”. We train the vanilla model for a few epochs ($E_{wu}$) first. Then, we conduct truncated matrix factorization (via truncated SVD) over the partially trained model weights to initialize the low-rank factors. For instance, given a partially trained FC layer $W^{(l)}$, we deploy SVD on it such that we get $U\Sigma V^T$. After that the $U$ and $V^T$ weights introduced in the previous sections can be found by $U = \tilde{U}\Sigma^2, V^T = \Sigma^2 V^T$. For convolution layer $W \in \mathbb{R}^{c_{in} \times c_{out} \times k \times k}$, we conduct SVD over the unrolled 2D matrix $W_{unrolled} \in \mathbb{R}^{c_{in} k^2 \times c_{out}}$, which leads to $U \in \mathbb{R}^{c_{in} k^2 \times r}, V^T \in \mathbb{R}^{r \times c_{out}}$ where reshaping $U, V$ back to 4D leads to the desired initial weights for the low-rank layer, i.e., $U \in \mathbb{R}^{c_{in} \times c_{out} \times k \times k}, V^T \in \mathbb{R}^{c_{in} \times c_{out} \times 1 \times 1}$. For the Batch Normalization layers (BNs) (Ioffe & Szegedy, 2015) we simply extract the weight vectors and the collected running statistics, e.g., the running mean and variance, for initializing the low-rank training. We also directly take the bias vector of the last FC layer. PUFFERFISH then finishes the remaining training epochs over the factorized hybrid network initialized with vanilla warm-up training.

Figure 3(b) provides an experimental justification on the effectiveness of vanilla warm-up training where we study a hybrid ResNet-50 trained on the ImageNet dataset. The results indicate that vanilla warm-up training helps to improve the accuracy of the factorized model. Moreover, a carefully tuned warm-up period of $E_{wu}$ also plays an important role in the final model accuracy. Though SVD is computationally heavy, PUFFERFISH only requires to conduct the SVD once throughout the entire training. We benchmark the SVD cost for all experimented models, which indicate the SVD runtime is comparatively small, e.g., on average, it only costs 2.29 seconds for ResNet-50. A complete study on the SVD factorization overheads can be found in the Appendix.

**Last FC layer.** The very last FC layer in a neural network can be viewed as a linear classifier over the features extracted by the previous layers. In general, its rank is equal to the number of classes in predictive task at hand. Factorizing it below the number of classes, will increase linear dependencies, and may further increase the approximation error. Thus, PUFFERFISH does not factorize it.

Putting all the techniques we discussed in this section together, the training procedure of PUFFERFISH is summarized in Algorithm 1.

### 4 Experiments

We conduct extensive experiments to study the effectiveness and scalability of PUFFERFISH over various computer vision and natural language processing tasks, across real distributed environments. We also compare PUFFERFISH against a wide range of baselines including: (i) P Gow-ERSGD, a low-rank based, gradient compression method that achieves high compression ratios (Vogels et al., 2019); (ii) SIGNUM, a gradient compression method that only communicates the sign of the local momentum (Bernstein et al., 2018a;b); (iii) The “early bird” structured pruning method EB Train (You et al., 2019); and (iv) The LTH sparsification method (referred to as LTH for simplicity) (Frankle & Carbin, 2018).

Our experimental results indicate that PUFFERFISH allows to train a model that is up to $3.35 \times$ smaller than other methods, with only marginal accuracy loss. Compared to Pow-ERSGD, SIGNUM, and vanilla SGD, PUFFERFISH achieves $1.22 \times, 1.52 \times$, and $1.74 \times$ end-to-end speedups respectively for ResNet-18 trained on CIFAR-10 while reaching to the same accuracy as vanilla SGD. PUFFERFISH leads to a model with $1.3M$ fewer parameters while reaching $1.76\%$ higher top-1 test accuracy than EB Train on the ImageNet dataset. Compared to LTH, PUFFERFISH leads to $5.67 \times$ end-to-end speedup for achieving the same model compression ratio for VGG-19 on CIFAR-10. We also demonstrate that the performance of PUFFERFISH is stable under the “mixed-precision training” implemented by PyTorch AMP.

Our code is publicly available for reproducing our results\(^1\).

#### 4.1 Experimental setup and implementation details

**Setup.** PUFFERFISH is implemented in PyTorch (Paszke et al., 2019). We experiment using two implementations. The first implementation we consider is a data-parallel model training API, i.e., DDP in PyTorch. However, as the gradient computation and communication are overlapped in DDP\(^2\), it is challenging to conduct a breakdown runtime analysis in DDP. We thus also come up with a prototype allreduce-based distributed implementation that decouples the computation and communication to benchmark the breakdown runtime of PUFFERFISH and other baselines. Our prototype distributed implementation is based on allreduce in PyTorch and the NCCL backend. All our experiments are deployed on a distributed cluster consisting of up to 16 p3.2xlarge (Tesla V100 GPU equipped) instances on Amazon EC2.

**Models and Datasets.** The datasets considered in our experiments are CIFAR-10 (Krizhevsky et al., 2009), ImageNet (ILSVRC2012) (Deng et al., 2009), the WikiText-2 datasets (Merity et al., 2016), and the WMT 2016 German-English translation task data (Elliott et al., 2016). For the image classification tasks on CIFAR-10, we considered VGG-

---

\(^1\)https://github.com/hwang595/Pufferfish

\(^2\)the computed gradients are buffered and communicated immediately when hitting a certain buffer size, e.g., 25MB.
19-BN (which we refer to as VGG-19) (Simonyan & Zisserman, 2014) and ResNet-18 (He et al., 2016). For ImageNet, we run experiments with ResNet-50 and WideResNet-50-2 (Zagoruyko & Komodakis, 2016). For the WikiText-2 dataset, we considered a 2-layer stacked LSTM model. For the language translation task, we consider a 6-layer Transformer architecture (Vaswani et al., 2017). More details about the datasets and models can be found in the Appendix.

**Implementation details and optimizations.** In our prototype distributed implementation, the allreduce operation starts right after all compute nodes finish computing the gradient. An important implementation-level optimization we conduct is that we pack all gradient tensors into one flat buffer, and only call the allreduce operation once per iteration. The motivation for such an optimization is that PUFFERFISH factorizes the full-rank layer $W_l$ to two smaller layers, i.e., $U_l, V_l^T$. Though the communication cost of the allreduce on each smaller layer is reduced, the total number of allreduce calls is doubled (typically an allreduce is required per layer to synchronize the gradients across the distributed cluster). According to the run-time cost model of the ring-allreduce (Thakur et al., 2005), each allreduce call introduces a network latency proportional to the product of the number of compute nodes and average network latency. This is not a negligible cost. Our optimization strategy aims at minimizing the additional latency overhead and leads to good performance improvement based on our tests. For a fair comparison, we conduct the same communication optimization for all considered baselines.

| Model archs. | Vanilla LSTM | PUFFERFISH LSTM |
|--------------|--------------|-----------------|
| # Params.    | 85, 962, 278 | 67, 962, 278    |
| Train Ppl.   | 52.87 ± 2.43 | 62.2 ± 0.74     |
| Val Ppl.     | 92.49 ± 0.41 | 93.62 ± 0.36    |
| Test Ppl.    | 88.16 ± 0.39 | 88.72 ± 0.24    |
| MACs         | 18M          | 9M              |

**Hyper-parameters for PUFFERFISH.** For all considered model architectures, we use a global rank ratio of 0.25, e.g., for a convolution layer with an initial rank of 64, PUFFERFISH sets $r = 64 \times 0.25 = 16$. For the LSTM on WikiText-2 experiment, we only factorize the LSTM layers and leave the tied embedding layer as is. Allocating the optimal rank for each layer can lead to better final model accuracy and smaller model sizes as discussed in (Idelbayev & Carreira-Perpinán, 2020). However, the search space for the rank allocation problem is large. One potential way to solve that problem is to borrow ideas from the literature of neural architectural search (NAS), which we leave as future work. We tune the initial low-rank layer index, i.e., $K$ and the vanilla warm-up training period to balance the hybrid model size and the final model accuracy. More details of the hyper-parameters of PUFFERFISH can be found in the Appendix.

| Model Archs. | # Params. | Test Acc. (%) | MACs (G) |
|--------------|-----------|---------------|----------|
| Vanilla VGG-19 (FP32) | 20, 560, 330 | 93.91 ± 0.01 | 0.4 |
| PUFFERFISH VGG-19 (FP32) | 8, 370, 634 | 93.89 ± 0.14 | 0.29 |
| Vanilla VGG-19 (AMP) | 20, 560, 330 | 94.12 ± 0.08 | N/A |
| PUFFERFISH VGG-19 (AMP) | 8, 370, 634 | 93.98 ± 0.06 | N/A |
| Vanilla ResNet-18 (FP32) | 11, 173, 834 | 95.09 ± 0.01 | 0.56 |
| PUFFERFISH ResNet-18 (FP32) | 3, 336, 138 | 94.87 ± 0.21 | 0.22 |
| Vanilla ResNet-18 (AMP) | 11, 173, 834 | 95.02 ± 0.1 | N/A |
| PUFFERFISH ResNet-18 (AMP) | 3, 336, 138 | 94.70 ± 0.37 | N/A |

**Table 3.** The results (averaged across 3 independent trials with different random seeds) of PUFFERFISH and vanilla 6-layer Transformers trained over the WMT 2016 German to English Translation Task.

**Table 4.** The results (averaged across 3 independent trials with different random seeds) of PUFFERFISH and vanilla VGG-19 and ResNet-18 trained over the CIFAR-10 dataset. Both full-precision training (FP32) and “mixed-precision training” (AMP) results are reported.

**4.2 Results**

**Parameter reduction and model accuracy.** We extensively study the effectiveness of PUFFERFISH, and the comprehensive numerical results are shown in Table 2, 3, 4, and 5. The main observation is that PUFFERFISH effectively reduces the number of parameters while introducing only marginal accuracy loss. In particular, PUFFERFISH ResNet-18 is $3.35 \times$ smaller than vanilla ResNet-18 with only 0.22% accuracy loss. Surprisingly, the PUFFERFISH Transformer leads to even better validation perplexity and test BLEU scores than the vanilla Transformer. One potential reason behind that is that factorizing the Transformer introduces some implicit regularization, leading to better generalization. Apart from the full precision training over FP32, we also conduct mixed-precision experiments over PyTorch AMP on both CIFAR-10 and ImageNet. Our results generally demonstrate that the performance of PUFFERFISH remains stable under mixed-precision training. We measure
Table 5. The results of the vanilla and PUFFERFISH ResNet-50 and WideResNet-50-2 models trained on the ImageNet dataset. For the ResNet-50 results, both full precision training (FP32) and mixed-precision training (AMP) are provided. For the AMP training, MACs are not calculated.

| Model Archs. | Number of Parameters | Final Test Acc. (Top-1) | Final Test Acc. (Top-5) | MACs (G) |
|--------------|----------------------|-------------------------|-------------------------|---------|
| Vanilla WideResNet-50-2 (FP32) | 68,883,240 | 78.09% | 94.00% | 11.44 |
| PUFFERFISH WideResNet-50-2 (FP32) | 40,047,400 | 77.84% | 93.88% | 9.99 |
| Vanilla ResNet-50 (FP32) | 25,557,032 | 76.93% | 93.41% | 4.12 |
| PUFFERFISH ResNet-50 (FP32) | 15,202,344 | 76.43% | 93.10% | 3.6 |
| Vanilla ResNet-50 (AMP) | 25,557,032 | 76.97% | 93.35% | N/A |
| PUFFERFISH ResNet-50 (AMP) | 15,202,344 | 76.35% | 93.22% | N/A |

Figure 4. (a) Breakdown per-epoch runtime analysis (top) and end-to-end convergence (bottom) results for vanilla SGD, PUFFERFISH, and SIGNUM over ResNet-50 trained on the ImageNet dataset. Where Comp. and Comm. stands for computation and communication costs; (b) Breakdown per-epoch runtime analysis (top) and end-to-end convergence (bottom) results for vanilla SGD, PUFFERFISH, SIGNUM, and PowerSGD over ResNet-50 trained on CIFAR-10; (c) The scalability of PUFFERFISH compared to vanilla SGD for ResNet-50 training on ImageNet using PyTorch DDP over the distributed clusters that consist of 2, 4, 8, 16 nodes (top); End-to-end convergence for vanilla SGD and PUFFERFISH with PyTorch DDP under the cluster with 8 nodes (bottom).

Table 6. The runtime mini-benchmark results of PUFFERFISH and vanilla VGG-19 and ResNet-18 networks training on the CIFAR-10 dataset. Experiment running on a single V100 GPU with batch size at 128, results averaged over 10 epochs; under the reproducible cuDNN setup with cudnn.benchmark disabled and cudnn.deterministic enabled; Speedup calculated based on the averaged runtime.

| Model Archs. | Epoch Time (sec.) | Speedup | MACs (G) |
|--------------|-------------------|---------|----------|
| Vanilla VGG-19 | 13.51 ± 0.02 | – | 0.4 |
| PUFFERFISH VGG-19 | 11.02 ± 0.01 | 1.23× | 0.29 |
| Vanilla ResNet-18 | 18.89 ± 0.07 | – | 0.56 |
| PUFFERFISH ResNet-18 | 12.78 ± 0.03 | 1.48× | 0.22 |

the computational complexity using “multiply–accumulate operations” (MACs)\(^3\). The MAC results are shown in Table 2, 4, and 5. The computation complexity is estimated by passing a single input through the entire network, e.g.,

\(^3\)https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation

for the CIFAR-10 dataset, we simulate a color image with size $32 \times 32 \times 3$ and pass it to the networks. For the LSTM network, we assume a single input token is with batch size at 1. We only report the MACs of forward pass. PUFFERFISH reduces the MACs of the vanilla model to up to $2.55\times$ over ResNet-18 on CIFAR-10.

Runtime mini-benchmark. It is of interest to investigate the actual speedup of the factorized networks as they are dense and compact. We thus provide mini-benchmark runtime results over VGG-19 and ResNet-18 on the CIFAR-10 dataset. We measure the per-epoch training speed of the factorized networks used in PUFFERFISH and the vanilla networks on a single V100 GPU with batch size at 128. The results are shown in Table 6. We report the results (averaged over 10 epochs) under the reproducibility optimized cuDNN environment, i.e., cudnn.benchmark disabled and cudnn.deterministic enabled. The results indicate that the factorized networks enjoy promising runtime speedups, i.e., $1.23\times$ and $1.48\times$ over the vanilla VGG-19 and ResNet-18 respectively. We also study the
runtime of the factorized networks under the speed optimized cuDNN setting, i.e., cudnn.benchmark enabled and cudnn.deterministic disabled, the results can be found in the Appendix.

**Computation and communication efficiency.** To benchmark the computation and communication costs of PUFFERFISH under a distributed environment, we conduct a per-epoch breakdown runtime analysis and compare it to vanilla SGD and SIGNUM on ResNet-50, trained over ImageNet. The experiment is conducted over 16 p3.2xlarge EC2 instances. We set the global batch size at 256 (16 per node). We use tuned hyper-parameters for all considered baselines.

The result is shown in Figure 4(a) where we observe that the PUFFERFISH ResNet-50 achieves 1.35× and 1.28× per-epoch speedups compared to vanilla SGD and SIGNUM respectively. Note that though SIGNUM achieves high compression ratio, it is not compatible with allreduce, thus allgather is used instead in our SIGNUM implementation. However, allgather is less efficient than allreduce, which hurts the communication efficiency of SIGNUM. The effect has also been observed in the previous literature (Vogels et al., 2019). We extend the per-epoch breakdown runtime analysis to ResNet-18 training on CIFAR-10 where we compare PUFFERFISH to POWERSGD, SIGNUM, and vanilla SGD. The experiments are conducted over 8 p3.2xlarge EC2 instances with the global batch size at 2048 (256 per node). We use a linear learning rate warm-up for 5 epochs from 0.1 to 1.6, which follows the setting in (Vogels et al., 2019; Goyal et al., 2017). For POWERSGD, we set the rank at 2, as it matches the same accuracy compared to vanilla SGD (Vogels et al., 2019). The results are shown in Figure 4(b), from which we observe that PUFFERFISH achieves 1.33×, 1.67×, 1.92× per-epoch speedups over POWERSGD, SIGNUM, and vanilla SGD respectively. Note that PUFFERFISH is slower than POWERSGD in the communication stage since POWERSGD massively compresses gradient and is also compatible with allreduce. However, PUFFERFISH is faster for gradient computing and bypasses the gradient encoding and decoding steps. Thus, the overall epoch time cost of PUFFERFISH is faster than POWERSGD. Other model training overheads, e.g., data loading and pre-processing, gradient flattening, and etc are not included in the “computation” stage but in the overall per-epoch time.

Since PUFFERFISH only requires to modify the model architectures instead of gradients, it is directly compatible with current data-parallel training APIs, e.g., DDP in PyTorch. Other gradient compression methods achieve high compression ratio, but they are not directly compatible with DDP without significant engineering effort. For PyTorch DDP, we study the speedup of PUFFERFISH over vanilla distributed training, measuring the per-epoch runtime on ResNet-50 and ImageNet over distributed clusters of size 2, 4, 8, and 16. We fix the per-node batch size at 32 following the setup in (Goyal et al., 2017). The results are shown in Figure 4(c). We observe that PUFFERFISH consistently outperforms vanilla ResNet-50. In particular, on the cluster with 16 nodes, PUFFERFISH achieves 1.52× per epoch speedup.

**End-to-end speedup.** We study the end-to-end speedup of PUFFERFISH against other baselines under both our prototype implementation and PyTorch DDP. The experimental setups for the end-to-end experiment are identical to our per-epoch breakdown runtime analysis setups. All reported runtimes include the overheads of the SVD factorization and vanilla warm-up training. The ResNet-50 on ImageNet convergence results with our prototype implementation are shown in Figure 4(a). We observe that to finish the entire 90 training epochs, PUFFERFISH attains 1.3× and 1.23× end-to-end speedups compared to vanilla SGD and SIGNUM respectively. The ResNet-18 on CIFAR-10 convergence results are shown in Figure 4(b). For faster vanilla warm-up training in PUFFERFISH, we deploy POWERSGD to compress the gradients. We observe that it is generally better to use a slightly higher rank for POWERSGD in the vanilla warm-up training period of PUFFERFISH. In our experiments, we use POWERSGD with rank 4 to warm up PUFFERFISH. We observe that to finish the entire 300 training epochs, PUFFERFISH attains 1.74×, 1.52×, 1.22× end-to-end speedup compared to vanilla SGD, SIGNUM, and POWERSGD respectively. PUFFERFISH reaches to the same accuracy compared to vanilla SGD. Moreover, we extend the end-to-end speedup study under PyTorch DDP where we compare PUFFERFISH with vanilla SGD under 8 EC2 p3.2xlarge instances. The global batch size is 256 (32 per node). The results are shown in Figure 4(c) where we observe that to train the model for 90 epochs, PUFFERFISH achieves 1.64× end-to-end speedup compared to vanilla SGD. We do not study the performance of SIGNUM and POWERSGD under DDP since they are not directly compatible with DDP.

**Comparison with structured pruning.** We compare PUFFERFISH with the EB Train method where structured pruning is conducted over the channel dimensions based on the activation values during the early training phase (You et al., 2019). EB Train finds compact and dense models. The result is shown in Table 7. We observe that compared to EB Train with prune ratio (pr) = 30%, PUFFERFISH returns a model with 1.3M fewer parameters while reaching 1.76% higher top-1 test accuracy. The EB Train experimental results are taken directly from the original paper (You et al., 2019). To make a fair comparison, we train PUFFERFISH with the same hyper-parameters that EB Train uses, e.g., removing label smoothing and only decaying the learning rate at the 30-th and the 60-th epochs with the factor 0.1.
PUFFERFISH: Communication-efficient Models At No Extra Cost

Table 7. Comparison of Hybrid ResNet-50 model compared to the Early-Bird Ticket structure pruned (EB Train) ResNet-50 model results with prune ratio \( pr \) at 30\%, 50\%, 70\% over the ImageNet dataset

| Model architectures | # Parameters | Final Test Acc. (Top-1) | Final Test Acc. (Top-5) | MACs (G) |
|---------------------|--------------|-------------------------|-------------------------|----------|
| vanilla ResNet-50   | 25, 610, 205 | 75.99\%                 | 92.98\%                 | 4.12     |
| PUFFERFISH ResNet-50| 15, 202, 344 | 75.62\%                 | 92.53\%                 | 3.6      |
| EB Train (\( pr = 30\% \)) | 16, 466, 787 | 73.86\%                 | 91.52\%                 | 2.8      |
| EB Train (\( pr = 50\% \)) | 15, 081, 947 | 73.30\%                 | 91.36\%                 | 2.37     |
| EB Train (\( pr = 70\% \)) | 7, 882, 503  | 70.16\%                 | 89.55\%                 | 1.03     |

Figure 5. The performance comparison between PUFFERFISH and LTH over a VGG-19 model trained over the CIFAR-10 dataset: (a) the number of parameters vs. wall-clock runtime; (b) the number of parameters pruned vs. the test accuracy.

Comparison with LTH. The recent LTH literature initiated by Frankle et al. (Frankle & Carbin, 2018), indicates that dense, randomly-initialized networks contain sparse subnetworks (referred to as “winning tickets”) that—when trained in isolation—reach test accuracy comparable to the original network (Frankle & Carbin, 2018). To find the winning tickets, an iterative pruning algorithm is conducted, which trains, prunes, and rewinds the remaining unpruned elements to their original random values repeatedly. Though LTH can compress the model massively without significant accuracy loss, the iterative pruning is computationally heavy. We compare PUFFERFISH to LTH across model sizes and computational costs on VGG-19 trained with CIFAR-10. The results are shown in Figure 5(a), 5(b) where we observe that to prune the same number of parameters, LTH costs 5.67\times more time than PUFFERFISH.

Ablation study. We conduct an ablation study on the accuracy loss mitigation methods in PUFFERFISH, i.e., hybrid network and vanilla warm-up training. The results on ResNet-18+CIFAR-10 and LSTM+WikiText-2 are shown in Table 8 and Table 9, which indicate that the hybrid network and vanilla warm-up training methods help to mitigate the accuracy loss effectively. Results on the other datasets can be found in the Appendix.

Limitations of PUFFERFISH. One limitation of PUFFERFISH is that it introduces two extra hyper-parameters, i.e., the initial low-rank layer index \( K \) and the vanilla warm-up epoch \( E_{wu} \), hence hyperparameter tuning requires extra effort. Another limitation is that although PUFFERFISH reduces the parameters in ResNet-18 and VGG-19 models effectively for the CIFAR-10 dataset, it only finds 1.68\times and 1.72\times smaller models for ResNet-50 and WideResNet-50-2 in order to preserve good final model accuracy.

5 Conclusion

We propose PUFFERFISH, a communication and computation efficient distributed training framework. Instead of gradient compression, PUFFERFISH trains low-rank networks initialized by factorizing a partially trained full-rank model. The use of a hybrid low-rank model and warm-up training, allows PUFFERFISH to preserve the accuracy of the fully dense SGD trained model, while effectively reducing its size. PUFFERFISH achieves high computation and communication efficiency and completely bypasses the gradient encoding and decoding, while yielding smaller and more accurate models compared pruning methods such the LTH and EB Train, while avoiding the burden of “winning the lottery”.

Acknowledgments This research is supported by an NSF CAREER Award #1844951, two SONY Faculty Innovation Awards, an AFOSR & AFRL Center of Excellence Awards, and an NSF TRIPODS Award #1740707. The authors also thank Yoshiki Tanaka, Hisahiro Suganuma, Pongsakorn U-chupala, Yuji Nishimaki, and Tomoki Sato from SONY for invaluable discussions and feedback.
REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 265–283, 2016.

Achille, A., Rovere, M., and Soatto, S. Critical learning periods in deep networks. In International Conference on Learning Representations, 2018.

Agarwal, S., Wang, H., Lee, K., Venkataraman, S., and Papailiopoulos, D. Accord: Adaptive gradient communication via critical learning regime identification. arXiv preprint arXiv:2010.16248, 2020.

Aji, A. F. and Heafield, K. Sparse communication for distributed gradient descent. arXiv preprint arXiv:1704.05021, 2017.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient SGD via gradient quantization and encoding. In Advances in Neural Information Processing Systems, pp. 1707–1718, 2017.

Bernstein, J., Wang, Y.-X., Azizzadenesheli, K., and Anandkumar, A. Signsgd: Compressed optimisation for non-convex problems. In International Conference on Machine Learning, pp. 560–569, 2018a.

Bernstein, J., Zhao, J., Azizzadenesheli, K., and Anandkumar, A. Signsgd with majority vote is communication efficient and fault tolerant. In International Conference on Learning Representations, 2018b.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.

Chen, C.-Y., Choi, J., Brand, D., Agrawal, A., Zhang, W., and Gopalakrishnan, K. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. arXiv preprint arXiv:1712.02679, 2017.

Chen, J., Pan, X., Monga, R., Bengio, S., and Jouzel, R. Revisiting distributed synchronous SGD. arXiv preprint arXiv:1604.00981, 2016a.

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, 2016b.

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.

De Sa, C., Feldman, M., Ré, C., and Olukotun, K. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 561–574. ACM, 2017.

De Sa, C., Leszczynski, M., Zhang, J., Marzoev, A., Aberger, C. R., Olukotun, K., and Ré, C. High-accuracy low-precision training. arXiv preprint arXiv:1803.03383, 2018.

De Sa, C. M., Zhang, C., Olukotun, K., and Ré, C. Taming the wild: A unified analysis of hogwild-style algorithms. In Advances in neural information processing systems, pp. 2674–2682, 2015.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. Large scale distributed deep networks. In Advances in neural information processing systems, pp. 1223–1231, 2012.

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.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171–4186, 2019.

Elliott, D., Frank, S., Sima’an, K., and Specia, L. Multitok: Multilingual english-german image descriptions. arXiv preprint arXiv:1605.00459, 2016.

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

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.

Grubic, D., Tam, L., Alistarh, D., and Zhang, C. Synchronous multi-GPU deep learning with low-precision communication: An experimental study. 2018.
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*, 2015.

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.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

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, 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.

Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, pp. 3, 2017.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binaryized 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.

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.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

Jastrzebski, S., Szymczak, M., Fort, S., Arpit, D., Tabor, J., Cho, K., and Geras, K. The break-even point on optimization trajectories of deep neural networks. *arXiv preprint arXiv:2002.09572*, 2020.

Jiang, Y., Zhu, Y., Lan, C., Yi, B., Cui, Y., and Guo, C. A unified architecture for accelerating distributed DNN training in heterogeneous gpus/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 463–479, 2020.

Karimireddy, S. P., Rebjock, Q., Stich, S., and Jaggi, M. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*, pp. 3252–3261, 2019.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Konečný, J. and Richtárik, P. Randomized distributed mean estimation: Accuracy vs communication. *arXiv preprint arXiv:1611.07555*, 2016.

Konečný, 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. 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.

Leblond, R., Pedregosa, F., and Lacoste-Julien, S. ASAGA: asynchronous parallel SAGA. *arXiv preprint arXiv:1606.04809*, 2016.

Leclerc, G. and Madry, A. The two regimes of deep network training. *arXiv preprint arXiv:2002.10376*, 2020.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.
Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. *OSDI*, 1(10.4):3, 2014.

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

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

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

Luo, J.-H., Wu, J., and Lin, W. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

Mania, H., Pan, X., Papailiopoulos, D., Recht, B., Ramchandran, K., and Jordan, M. I. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv preprint arXiv:1507.06970*, 2015.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

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.

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

Qi, H., Sparks, E. R., and Talwalkar, A. Paleo: A performance model for deep neural networks. In *Proceedings of the International Conference on Learning Representations*, 2017.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pp. 525–542. Springer, 2016.

Renggli, C., Alistarh, D., and Hoefler, T. SparCML: high-performance sparse communication for machine learning. *arXiv preprint arXiv:1802.08021*, 2018.

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.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

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

Stich, S. U., Cordonnier, J.-B., and Jaggi, M. Sparsified sgd with memory. In *Advances in Neural Information Processing Systems*, pp. 4447–4458, 2018.

Strom, N. Scalable distributed DNN training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

Suresh, A. T., Yu, F. X., Kumar, S., and McMahan, H. B. Distributed mean estimation with limited communication. *arXiv preprint arXiv:1611.00429*, 2016.

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

Tang, H., Yu, C., Lian, X., Zhang, T., and Liu, J. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International Conference on Machine Learning*, pp. 6155–6165. PMLR, 2019.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

Touvron, H., Vedaldi, A., Douze, M., and Jégou, H. Fixing the train-test resolution discrepancy: Fixefficientnet. *arXiv preprint arXiv:2003.08237*, 2020.

Tsuzuku, Y., Imachi, H., and Akiba, T. Variance-based gradient compression for efficient distributed deep learning. *arXiv preprint arXiv:1802.06058*, 2018.

Tucker, L. R. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.
Vogels, T., Karimireddy, S. P., and Jaggi, M. Powersgd: Practical low-rank gradient compression for distributed optimization. In Advances in Neural Information Processing Systems, pp. 14259–14268, 2019.

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

Wang, H., Sievert, S., Liu, S., Charles, Z., Papailiopoulos, D., and Wright, S. Atomo: Communication-efficient learning via atomic sparsification. In Advances in Neural Information Processing Systems, pp. 9850–9861, 2018.

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.

Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In Advances in Neural Information Processing Systems, pp. 1508–1518, 2017.

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.

Wu, J., Huang, W., Huang, J., and Zhang, T. Error compensated quantized sgd and its applications to large-scale distributed optimization. In International Conference on Machine Learning, pp. 5325–5333, 2018.

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.

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.

Yu, J., Yang, L., Xu, N., Yang, J., and Huang, T. Slimmable neural networks. arXiv preprint arXiv:1812.08928, 2018a.
We provide the complexity and number of parameters in the LSTM layer. The computation complexity is simply $O(mn)$ for $xW$ and $O(nr + rn)$ for $(xU)V^T$. For the number of parameters, the vanilla LSTM layer contains $mn$ parameters in total while the low-rank FC layer contains $r(m + n)$ parameters in total.

**Convolution layer.** For a convolution layer, assuming the input is with dimension $x \in \mathbb{R}^{c_{in} \times H \times W}$ (the input “image” has $c_{in}$ color channels and with size $H \times W$), the computation complexity of a vanilla convolution layer with weight $W \in \mathbb{R}^{c_{out} \times k \times k \times k}$ is $O(c_{in}c_{out}k^2HW)$ for computing $W \ast x$ where $\ast$ is the linear convolution operation. And the low-rank factorized convolution layer with dimension $U \in \mathbb{R}^{c_{in} \times r \times k \times k}$, $V \in \mathbb{R}^{c_{out} \times 1 \times 1 \times 1}$ has the computation complexity at $O(rc_{out}k^2HW)$ for $U \ast x$ and $O(rHWc_{out})$ for convolving the output of $U \ast x$ with $V$. For the number of parameters, the vanilla convolution layer contains $c_{in}c_{out}k^2$ parameters in total while the low-rank convolution layer contains $c_{in}rk^2 + rc_{out}$ parameters in total.

**LSTM layer.** For the LSTM layer, the computation complexity is similar to the computation complexity of the FC layer. Assuming the tokenized input is with dimension $x \in \mathbb{R}^d$, and the concatenated input-hidden and hidden-hidden weights $W_i \in \mathbb{R}^{d \times 4h}$, $W_h \in \mathbb{R}^{4h \times h}$, thus the computation complexity of the forward propagation of a LSTM layer is $O(4dh + 4h^2)$. And for the low-rank LSTM layer, the computation complexity becomes $O(dr + 4rh + 4hr + rh)$ (as mentioned in Section 2, we assume that the same rank $r$ is used for both the input-hidden weight and hidden-hidden weight). For the number of parameters, the vanilla LSTM layer contains $4dh + 4h^2$ parameters in total while the low-rank convolution layer contains $4(dr + rh) + 4(hr + rh) = 4dr + 12hr$ parameters in total.

**Transformer.** For the encoder layer in the Transformer architecture, there are two main components, i.e., the multi-head attention layer and the FFN layer. Note that, for the multi-head attention layer, the dimensions of the projection matrices are: The dimensions of the matrices are $Q, K, V \in \mathbb{R}^{H \times d}$, $WQ, WK, WV \in \mathbb{R}^{pd \times d}$, $W \in \mathbb{R}^{pd \times pd}$. And the dimensions of the two FC layers in the FFN are with dimensions $W_1 \in \mathbb{R}^{pd \times 4pd}$, $W_2 \in \mathbb{R}^{4pd \times pd}$. And we assume a sequence of input tokens with length $N$ is batched to process together. Since the computation for each attention head is computed independently, we only analyze the computation complexity of a single head attention, which is $O(d \cdot pd \cdot N + 2N^2 \cdot d + pd \cdot pd \cdot N)$ for the attention layer and $O((p + p^2)N^2d^2 + N^2d)$ for the low-rank FFN layer, the computation complexity becomes $O((p + 1)drN + 2Ndpd + 2N^2d) = O(pdrN + N^2d)$ and the computation complexity for FFN is $O((p \cdot d \cdot r + 4r \cdot h \cdot d) \cdot N + (p \cdot d \cdot r + 4r \cdot p \cdot d) \cdot N)$.

For the number of parameters, the vanilla multi-head attention layer contains $3pd^2 \cdot p + p^2d^2 = 4p^2d^2$ parameters in total while the low-rank multi-head attention layer contains $3pd^2 \cdot dr + (pd^2 \cdot rd) = pd(3pd + 5)$ parameters in total. The vanilla FFN layer contains $4p^2d^2 + 4p^2d^2 = 8p^2d^2$ parameters in total while the low-rank FFN layer contains $(pdr + 4pd) + (4pdr + rp) = 10pdr$ parameters in total.

**Details on the dataset and models used for the experiment**

The details of the datasets used in the experiments are summarized in Table 10.

**Details on the hybrid networks in the experiments**

The hybrid VGG-19-BN architecture. we generally found that using $K = 10$ in the VGG-19-BN architecture leads to good test accuracy and moderate model compression ratio.
Table 10. The datasets used and their associated learning models.

| Method                  | CIFAR-10 | ImageNet | WikiText-2 | WMT16' Gen-Eng |
|-------------------------|----------|----------|------------|---------------|
| # Data points           | 60,000   | 1,281,167| 29,000     | 1,017,981     |
| Data Dimension          | 32 × 32 × 3 | 224 × 224 × 3 | 1,500       | 9,521         |
| Model                   | VGG-19-BN:ResNet-18 | ResNet-50:WideResNet-50-2 | 2 layer LSTM | Transformer (p = 8, N = 6) |
| Optimizer               | SGD      | SGD      | ADAM       |               |
| Hyper-params            | lr: 0.01 | lr: 20(decay with 0.25 when val. loss not decreasing) | Init lr: 0.001 |                |
|                         | momentum: 0.9, $\ell_2$ weight decay: $10^{-4}$ |                      |              |

Table 11. Detailed information of the hybrid VGG-19-BN architecture used in our experiments, all non-linear activation function in this architecture is ReLU after each convolution layer (omitted in the Table). The shapes for convolution layers follows $(c_i, c_o, k, k)$. There is a BatchNorm layer after each convolution layer with number of neurons the same as $c_o$ (also omitted in the Table).

| Parameter   | Shape                  | Layer hyper-parameter |
|-------------|------------------------|-----------------------|
| layer 1. conv1.weight | $3 \times 64 \times 3 \times 3$ | stride:1; padding:1 |
| layer 2. conv2.weight | $64 \times 64 \times 3 \times 3$ | stride:1; padding:1 |
| pooling.max             | N/A                    | kernel size:2; stride:2 |
| layer 3. conv3.weight | $64 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 4. conv4.weight | $128 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| pooling.max             | N/A                    | kernel size:2; stride:2 |
| layer 5. conv5.weight | $128 \times 256 \times 3 \times 3$ | stride:1; padding:1 |
| layer 6. conv6.weight | $256 \times 256 \times 3 \times 3$ | stride:1; padding:1 |
| layer 7. conv7.weight | $256 \times 256 \times 3 \times 3$ | stride:1; padding:1 |
| layer 8. conv8.weight | $256 \times 256 \times 3 \times 3$ | stride:1; padding:1 |
| pooling.max             | N/A                    | kernel size:2; stride:2 |
| layer 9. conv9.weight | $256 \times 512 \times 3 \times 3$ | stride:1; padding:1 |
| layer 10. conv10.weight | $512 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 10. pool.weight   | $128 \times 128 \times 1 \times 1$ | stride:1 |
| layer 11. conv11.weight | $128 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 11. pool.weight   | $128 \times 128 \times 1 \times 1$ | stride:1 |
| layer 12. conv12.weight | $128 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 12. pool.weight   | $128 \times 128 \times 1 \times 1$ | stride:1 |
| layer 13. conv13.weight | $512 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 13. pool.weight   | $512 \times 128 \times 1 \times 1$ | stride:1 |
| layer 14. conv14.weight | $512 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 14. pool.weight   | $512 \times 128 \times 1 \times 1$ | stride:1 |
| layer 15. conv15.weight | $512 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 15. pool.weight   | $512 \times 128 \times 1 \times 1$ | stride:1 |
| layer 16. conv16.weight | $512 \times 128 \times 3 \times 3$ | stride:1; padding:1 |
| layer 16. pool.weight   | $512 \times 128 \times 1 \times 1$ | stride:1 |
| layer 17. fc17.weight   | $512 \times 512$        | N/A                   |
| layer 17. fc17.bias     | $512$                   | N/A                   |
| layer 18. fc18.weight   | $512 \times 512$        | N/A                   |
| layer 18. fc18.bias     | $512$                   | N/A                   |
| layer 19. fc19.weight   | $512 \times 10$         | N/A                   |
| layer 19. fc19.bias     | $10$                    | N/A                   |

The low-rank LSTM architecture. Note that we only use a 2-layer stacked LSTM as the model in the WikiText-2 next word prediction task. Our implementation is directly modified from the PyTorch original example⁴. We used the tied version of LSTM, i.e., enabling weight sharing for the encoder and decoder layers.

Table 12. Detailed information on the low-rank LSTM architecture in our experiment.

| Parameter                  | Shape                  | Hyper-param. |
|----------------------------|------------------------|--------------|
| encoder.weight             | $33278 \times 1500$    | N/A          |
| dropout                    | N/A                    | $p = 0.65$   |
| lstm0.weight.ii/f/g/o_u    | $1500 \times 375$      | N/A          |
| lstm0.weight.ii/f/g/o_v    | $375 \times 375$       | N/A          |
| lstm0.weight.hi/f/g/o_u    | $1500 \times 375$      | N/A          |
| lstm0.weight.hi/f/g/o_v    | $375 \times 375$       | N/A          |
| dropout                    | N/A                    | $p = 0.65$   |
| lstm1.weight.ii/f/g/o_u    | $1500 \times 375$      | N/A          |
| lstm1.weight.ii/f/g/o_v    | $375 \times 375$       | N/A          |
| lstm1.weight.hi/f/g/o_u    | $1500 \times 375$      | N/A          |
| lstm1.weight.hi/f/g/o_v    | $375 \times 375$       | N/A          |
| decoder.weight(shared)     | $1500 \times 33278$    | N/A          |

The hybrid ResNet-18, ResNet-50, WideResNet-50-2 architectures. For the CIFAR-10 dataset, we modified the original ResNet-50 architecture described in the original ResNet paper (He et al., 2016). The details about the modified ResNet-18 architecture for the CIFAR-10 dataset are shown in Table 13. The network architecture is modified from the public code repository⁵. For the first 2 convolution block, i.e., conv2_x, we used stride at 1 and padding at 1 for all the convolution layers. For conv3_x, conv4_x, and conv5_x we used the stride at 2 and padding at 1. We also note that there is a BatchNorm layer after each convolution layer with the number of elements equals the number of

---

⁴https://github.com/pytorch/examples/tree/master/word_language_model
⁵https://github.com/kuangliu/pytorch-cifar
convolution filters. As shown in Table 13, our hybrid architecture starts from the 2nd convolution block, i.e., $K = 4$. Our experimental study generally shows that this choice of hybrid ResNet-18 architecture leads to a good balance between the final model accuracy and the number of parameters. Moreover, we did not handle the downsample weights in the convolution blocks.

| Layer Name | ResNet-18 Rank Information       |
|------------|----------------------------------|
| conv1      | 3×3, 64, stride 1, padding 1     |
| conv2_x    | 3×3, 64 × 2 1st block full-rank  |
| conv3_x    | 3×3, 128 × 2 low-rank            |
| conv4_x    | 3×3, 256 × 2 low-rank            |
| conv5_x    | 3×3, 512 × 2 low-rank            |
| Avg Pool   | 10-dim FC, SoftMax               |

Table 13. The ResNet-18 architecture for the CIFAR-10 dataset used in the experiments.

For the ResNet-50 architecture, the detailed information is shown in the Table 14. As we observed that the last three convolution blocks, i.e., conv5.x contains around 60% of the total number of parameters in the entire network, thus we just put the last three convolution blocks as low-rank blocks and all other convolution blocks are full-rank blocks. Note that, different from the ResNet-18 architecture for the CIFAR-10 dataset described above. We also handle the downsample weight inside the ResNet-50 network, which only contains in the very first convolution block of conv5.x. The original dimension of the downsample weight is with shape (1024, 2048, 1, 1). Our factorization strategy leads to the shape of conv_u: (1024, 256, 1, 1) and conv_v: (256, 2048, 1, 1).

| Layer Name | output size | WideResNet-50-2 Rank Information       |
|------------|-------------|----------------------------------------|
| conv1      | 3×3 max pool, stride 2     |
| conv2_y    | 1×1, 128     |
| conv3_y    | 1×1, 512     |
| conv4_y    | 1×1, 1024    |
| conv5_y    | 1×1, 2048    |
| Avg Pool   | 1024-dim FC, SoftMax      |

Table 15. The WideResNet-50-2 architecture for the ImageNet dataset used in the experiments.

The hybrid Transformer architecture. The Transformer architecture used in the experiment follows from the original Transformer paper (Vaswani et al., 2017). Our implementation is modified from the public code repository.

https://github.com/jadore801120/attention-is-all-you-need-pytorch
The hybrid VGG-19-BN architecture used for the LTH comparison. To compare PUFFERFISH with LTH, we use the open-source LTH implementation, i.e., https://github.com/facebookresearch/openlth. The VGG-19-BN model used in the open-source LTH repository is slightly different from the VGG-19-BN architecture described above. We thus use the VGG-19-BN architecture in the LTH code and deploy PUFFERFISH on top of it for fairer comparison. Detailed information about the hybrid VGG-19-BN architecture we used in PUFFERFISH for the comparison with LTH is shown in Table 18.

### Table 16. Detailed information of the encoder layer in the Transformer architecture in our experiment

| Parameter                      | Shape | Hyper-param. |
|-------------------------------|-------|---------------|
| embedding                     | 9521 × 512 | padding index: 1 |
| positional encoding           | N/A   | N/A           |
| dropout                       | N/A   | p = 0.1       |
| encoder.self-attention.wq(Wq) | 512 × 512 | N/A           |
| encoder.self-attention.wk(Wk) | 512 × 512 | N/A           |
| encoder.self-attention.wv(Wv) | 512 × 512 | N/A           |
| encoder.self-attention.wo(Wo) | 512 × 512 | N/A           |
| decoder.self-attention.dropout| N/A   | p = 0.1       |
| encoder.ffn.layer1            | 512 × 2048 | N/A           |
| encoder.ffn.layer2            | 2048 × 512 | N/A           |
| encoder.layernorm             | 512   | € = 10⁻⁶      |

### Table 17. Detailed information of the decoder layer in the Transformer architecture in our experiment

| Parameter                      | Shape | Hyper-param. |
|-------------------------------|-------|---------------|
| embedding                     | 9521 × 512 | padding index: 1 |
| positional encoding           | N/A   | N/A           |
| dropout                       | N/A   | p = 0.1       |
| decoder.self-attention.wq(Wq) | 512 × 512 | N/A           |
| decoder.self-attention.wk(Wk) | 512 × 512 | N/A           |
| decoder.self-attention.wv(Wv) | 512 × 512 | N/A           |
| decoder.self-attention.wo(Wo) | 512 × 512 | N/A           |
| decoder.self-attention.dropout| N/A   | p = 0.1       |
| decoder.self-attention.layernorm | 512 | € = 10⁻⁶      |

### Table 18. Detailed information of the hybrid VGG-19-BN architecture used in our LTH comparison experiments, all non-linear activation function in this architecture is ReLU after each convolution layer (omitted in the Table). The shapes for convolution layers follows (Cin, Cout, k, k). There is a BatchNorm layer after each convolution layer with number of neurons the same as Cout (also omitted in the Table).

| Parameter                      | Shape | Layer hyper-parameter |
|-------------------------------|-------|-----------------------|
| layer1.conv1.weight           | 3 × 64 × 3 × 3 | stride:1; padding:1 |
| layer2.conv2.weight           | 64 × 64 × 3 × 3 | stride:1; padding:1 |
| pooling.max                   | N/A   | kernel size:2; stride:2 |
| layer3.conv3.weight           | 64 × 128 × 3 × 3 | stride:1; padding:1 |
| pooling.max                   | N/A   | kernel size:2; stride:2 |
| layer4.conv4.weight           | 128 × 128 × 3 × 3 | stride:1; padding:1 |
| pooling.max                   | N/A   | kernel size:2; stride:2 |
| layer5.conv5.weight           | 128 × 256 × 3 × 3 | stride:1; padding:1 |
| layer6.conv6.weight           | 256 × 256 × 3 × 3 | stride:1; padding:1 |
| layer7.conv7.weight           | 256 × 256 × 3 × 3 | stride:1; padding:1 |
| layer8.conv8.weight           | 256 × 256 × 3 × 3 | stride:1; padding:1 |
| pooling.max                   | N/A   | kernel size:2; stride:2 |
| layer9.conv9.weight           | 256 × 512 × 3 × 3 | stride:1; padding:1 |
| layer10.conv10_u.weight       | 512 × 128 × 3 × 3 | stride:1; padding:1 |
| layer10.conv10_v.weight       | 128 × 128 × 1 × 1 | stride:1 |
| layer11.conv11_u.weight       | 512 × 128 × 3 × 3 | stride:1; padding:1 |
| layer11.conv11_v.weight       | 128 × 128 × 1 × 1 | stride:1 |
| layer12.conv12_u.weight       | 512 × 128 × 3 × 3 | stride:1; padding:1 |
| layer12.conv12_v.weight       | 128 × 128 × 1 × 1 | stride:1 |
| pooling.max                   | N/A   | kernel size:2; stride:2 |
| layer13.conv13_u.weight       | 512 × 128 × 3 × 3 | stride:1; padding:1 |
| layer13.conv13_v.weight       | 128 × 128 × 1 × 1 | stride:1 |
| layer14.conv14_u.weight       | 512 × 128 × 3 × 3 | stride:1; padding:1 |
| layer14.conv14_v.weight       | 128 × 128 × 1 × 1 | stride:1 |
| layer15.conv15_u.weight       | 512 × 128 × 3 × 3 | stride:1; padding:1 |
| layer15.conv15_v.weight       | 128 × 128 × 1 × 1 | stride:1 |
| layer16.conv16_u.weight       | 512 × 128 × 3 × 3 | stride:1; padding:1 |
| layer16.conv16_v.weight       | 128 × 512 × 1 × 1 | stride:1 |
| pooling.max                   | N/A   | kernel size:2; stride:2 |
| layer17.flat17.weight         | 512 × 10 | N/A |
| layer17.flat17.bias           | 10    | N/A |
E  THE COMPATIBILITY OF PUFFERFISH WITH OTHER GRADIENT COMPRESSION METHODS

As PUFFERFISH is a training time parameter reduction method, the gradient of the factorized networks can be compressed further with any gradient compression methods. As PowerSGD is the state-of-the-art gradient compression method and is compatible with allreduce, we consider another baseline, i.e., “UFFERFISH+PowerSGD”, we conduct an experimental study over this baseline on ResNet-18 trained on CIFAR-10 (results shown in Figure 6). The experiment is running over 8 p3.2xlarge EC2 nodes with batch size at 256 per node (2048 in total). The experimental results indicate that combining PUFFERFISH with PowerSGD can effectively reduce the gradient size of PUFFERFISH, making PUFFERFISH enjoys high computation efficiency and the communication efficiency as high as PowerSGD. However, as PowerSGD conducts layer-wise gradient encoding and decoding on both $U_i$ and $V_i$ layers, the gradient encoding and decoding cost in the “UFFERFISH+PowerSGD” baseline is higher compared to PowerSGD. We observe that a slightly higher rank is desired when combining PUFFERFISH with PowerSGD since both model weights and gradients are approximated in this case. In the experimental results shown in Figure 6, we use PowerSGD with rank 4 when combining with PUFFERFISH for both the vanilla warm-up training epochs and the consecutive low-rank training epochs. Moreover, we also found that under the large-batch setting, it is always helpful to re-warmup the learning rate for the “UFFERFISH+PowerSGD” baseline, i.e., in the first 5 epochs, we warm-up the learning rate linearly from 0.1 to 1.6, then at the 80-th epoch when we switch from the vanilla warm-up training to low-rank training, we repeat the learning rate warm-up again with 5 epochs (from 0.1 to 1.6). Our experimental results suggest that PUFFERFISH can be combined with the gradient compression methods to attain better communication efficiency, but it is desirable to combine PUFFERFISH with the gradient compression methods that can be deployed on the fattened gradients, e.g., Top-$k$.

F  DISCUSSION ON THE COMMUNICATION EFFICIENCY OF PUFFERFISH

It is natural to ask the question that “Why are the previously proposed light weight gradient compression methods slow in practice, e.g., the ones proposed in (Suresh et al., 2016)” We agree that there are lots of gradient compression methods, which are computationally cheap. However, other important factors can affect the gradient compression efficiency in practice (taking the gradient compression method in (Suresh et al., 2016) as an example):

(i) After the binary sign rounding, extra encoding and decoding steps e.g. binary encoding are required to aggregate the quantized bits to bytes for attaining real communication speedup. That is optimizing the data structures to support low-communication for quantized gradients is necessary for any benefit to the surface, and also quite non trivial. (ii) For most gradient compression schemes, the encoded gradients are not compatible with all-reduce. Thus, all-gather has to be used instead. Unfortunately, in terms of comm. costs all-gather suffers a performance gap that increases with the number of nodes. (iii) In all-reduce, each worker receives a pre-aggregated gradient, making the cost of decompression independent to the number of workers. In all-gather, a worker receives the number of workers compressed gradients that need to be individually decompressed and aggregated. The time for decompression with all-gather therefore scales linearly with the number of workers.

In fact we did run a test for the “Stochastic binary quantization” method in (Suresh et al., 2016) on ResNet-50+ImageNet over 16 EC2 p3.2xlarge nodes (per node batch size 32) as it is the computationally cheapest methods proposed in the paper. Though it is showed that conducting random rotation over the gradients can improve the compression error, we only care about the computational and communication efficiencies of the method in this particular experiment. Per epoch runtime results are shown in Figure 7.
Figure 7. Breakdown per-epoch runtime comparison between PUFFERFISH, vanilla SGD, and stochastic binary quantization. Note that in the “compress.+decompress.” stage, stochastic binary quantization takes 12.1 ± 0.6 seconds for gradient compression and 118.4 ± 0.1 for gradient decompression. We observe that although the stochastic binary quantization is efficient in the compression stage, its gradient decompression cost is expensive. Moreover, all-gather is less efficient compared to all-reduce at the scale of 16 nodes.

G THE EFFECTIVENESS OF USING SVD TO FIND THE LOW-RANK FACTORIZATION

In the vanilla warm-up training strategy proposed in PUFFERFISH, we decompose the network weights using SVD to find the initialization weights for the hybrid network. Though SVD is a computationally expensive method, PUFFERFISH only requires to conduct the factorization over the network weights once during the entire training process. We explicitly test the overhead incurred by conducting SVD over the model weights here. All the runtimes are measured over the p3.2xlarge instance of Amazon EC2 (equipped with Tesla V100 GPU). The results are shown in Figure 19. From the results, it can be observed that the run time on using SVD to factorize the partially trained vanilla full-rank network is quite fast, e.g., on average it only costs 2.2972 seconds over the ResNet-50 trained over the ImageNet dataset, which only takes 0.17% of the per epoch training time.

Table 19. The time costs on conducting SVD over the partially trained vanilla full-rank network to find the initialization model for the hybrid network. The run time results are averaged from 5 independent trials.

| Method                | Time Cost (in sec.) |
|-----------------------|---------------------|
| ResNet-50 on ImageNet | 2.2972 ± 0.0519     |
| WideResNet-50-2 on ImageNet | 4.8700 ± 0.0859 |
| VGG-19-BN on CIFAR-10 | 1.5198 ± 0.0113     |
| ResNet-18 on CIFAR-10 | 1.3244 ± 0.0201     |
| LSTM on WikiText-2    | 6.5791 ± 0.0445     |
| Transformer on WMT16  | 5.4104 ± 0.0542     |

H DETAILS OF DATA PREPROCESSING

The CIFAR-10 dataset. In preprocessing the images in CIFAR-10 dataset, we follow the standard data augmentation and normalization process. For data augmentation, random cropping and horizontal random flipping are used. Each color channels are normalized with mean and standard deviation by \( \mu_r = 0.491, \mu_g = 0.482, \mu_b = 0.447, \sigma_r = 0.247, \sigma_g = 0.244, \sigma_b = 0.262 \). Each channel pixel is normalized by subtracting the mean value in this color channel and then divided by the standard deviation of this color channel.

The ImageNet dataset. For ImageNet, we follow the data augmentation process of (Goyal et al., 2017), i.e., we use scale and aspect ratio data augmentation. The network input image is a 224 × 224 pixels, randomly cropped from an augmented image or its horizontal flip. The input image is normalized in the same way as we normalize the CIFAR-10 images using the following means and standard deviations: \( \mu_r = 0.485, \mu_g = 0.456, \mu_b = 0.406; \sigma_r = 0.229, \sigma_g = 0.224, \sigma_b = 0.225 \).

I DETAILED HYPER-PARAMETERS USED IN OUR EXPERIMENTS

ResNet-50 and WideResNet-50-2 over the ImageNet dataset. For ResNet-50 and WideResNet-50-2 models, we follow the model training hyper-parameters reported in (Goyal et al., 2017). We train the model using the optimizer SGD with momentum with momentum over the model weights instead of the BatchNorm layers with the regularization coefficient \( 10^{-4} \). The entire training process takes 90 epochs. For both of the ResNet-50 and WideResNet-50-2 models, we start from the learning rate at 0.1 and decay the learning rate by a factor of 0.1 at the 30-th, 60-th, and the 80-th epochs. For the vanilla warm-up training, we use warm-up epoch \( E = 10 \). Note that at the 10-th epoch we switch from the vanilla ResNet-50/WideResNet-50-2 models to the hybrid architecture, but we still use the same learning rate, i.e., 0.1 until the 30-th epoch. Additional to the previously proposed work, we adopt the label smoothing technique with probability 0.1. The model initialization method follows directly from the implementation of PyTorch example 7.

ResNet-18 and VGG-19-BN over the CIFAR-10 dataset. For ResNet-18 and VGG-19-BN models. We train the model using the optimizer SGD with momentum with momentum value at 0.9 with batch size at 128. The entire training takes 300 epochs. We also conduct \( \ell_2 \) regularization over the

\[\text{https://github.com/pytorch/examples/tree/master/imagenet}\]
model weights with the regularization coefficient $10^{-4}$. For both of the ResNet-18 and VGG-19-BN models, we start from the learning rate at 0.1 and decay the learning rate by a factor of 0.1 at the 150-th, 250-th epochs. For the vanilla warm-up training, we use warm-up epoch $E = 80$. Note that at the 80-th epoch we switch from the vanilla ResNet-18/VGG-19-BN models to the hybrid architecture, but we still use the same learning rate, i.e., 0.1 until the 150-th epoch.

LSTM over the WikiText-2 dataset. For the LSTM model. We conduct training using the vanilla SGD optimizer with batch size at 20. We also conduct gradient norm clipping with norm bound at 0.5. The entire training takes 40 epochs. We start from the learning rate at 20 and decay the learning rate by a factor of 0.25 if the validation loss is not decreasing. For the vanilla warm-up training, we use warm-up epoch $E = 10$. Note that at the 10-th epoch we switch from the vanilla LSTM model to the hybrid architecture, we also decay the learning rate by a factor of 0.5. We also tie the word embedding and SoftMax weights (Press & Wolf, 2016).

The Transformer over the WMT16 dataset. For the Transformer model. We conduct training using the Adam optimizer with initial learning rate at 0.001, $\beta_s = (0.9, 0.98), \epsilon = 10^{-8}$ batch size at 256. We also conduct gradient norm clipping with norm bound at 0.25. The entire training takes 400 epochs. For the vanilla warm-up training, we use warm-up epoch $E = 10$. We enable label smoothing, weight sharing for the source and target word embedding, and weight sharing between target word embedding and the last dense layer.

J Detailed Information on the Runtime Mini-Benchmark

In the experiment section, we discussed that in the reproducibility optimized setting, factorized networks achieve promising runtime speedup over the vanilla networks. However, sometimes users prefer faster runtime to reproducibility where the speed optimized setting is used (with cudnn.benchmark enabled and cudnn.deterministic disabled). We also study the runtime of the factorized network under the speed optimized setting. The results are shown in Table 20, from which we observe that the speedup of the factorized network is less promising compared to the reproducibility optimized setting especially for the VGG-19-BN network. However, PUFFERFISH ResNet-18 still achieves $1.16 \times$ per-epoch speedup. We leave exploring the optimal model training speed of the factorized networks as the future work.

Table 20. The runtime mini-benchmark results of PUFFERFISH and vanilla VGG-19-BN and ResNet-18 networks training on the CIFAR-10 dataset, results averaged over 10 epochs. Experiment running on a single V100 GPU with batch size at 128; Over the optimized cudNN implementation with cudnn.benchmark enabled and cudnn.deterministic disabled; Speedup calculated based on the averaged per-epoch time.

| Model Archs. | Epoch Time (sec.) | Speedup | MACs (G) |
|--------------|-------------------|---------|----------|
| Vanilla VGG-19 | 8.27 ± 0.07 | – | 0.4 |
| PUFFERFISH VGG-19 | 8.16 ± 0.12 | $1.01 \times$ | 0.29 |
| Vanilla ResNet-18 | 11.15 ± 0.01 | – | 0.56 |
| PUFFERFISH ResNet-18 | 9.61 ± 0.08 | $1.16 \times$ | 0.22 |

K Time Cost Measurement on Amazon EC2

We use the p3.2xlarge instances for the distributed experiments, the bandwidth of the instance is “Up to 10 Gbps” as stated on the Amazon EC2 website, i.e., https://aws.amazon.com/ec2/instance-types/p3/. For some tasks (especially for the ResNet-50 and WideResNet-50-2), we observe that the bandwidth of the p3.2xlarge instance decays sharply in the middle of the experiment. The time costs for ResNet-50 trained on the ImageNet dataset under our prototype allreduce distributed implementation are collected when there is no bandwidth decay, e.g., under 10 Gbps. For the DDP time cost results, we run the experiments till the per-epoch time costs become stable, then measure the per-epoch time. For ResNet-18 trained on the CIFAR-10 dataset experiments under our prototype allreduce distributed implementation, we do not observe significant bandwidth decay for the p3.2xlarge instances. All distributed experiments are conducted under the us-west-2c availability zone of EC2.

L Additional Experimental Results

The ablation study on the accuracy mitigation strategy over CIFAR-10 and ImageNet. The ablation study results are shown in Table 21 for ResNet-50 trained on ImageNet and Table 22 for VGG-19-BN trained over CIFAR-10. For the vanilla low-rank ResNet-50 trained on ImageNet, we do not deploy the label smoothing and the extra learning rate decay (with a factor 0.1) at the 80-th epoch.
Table 21. The effect of vanilla warm-up training and hybrid network architectures of PUFFERFISH of the low-rank ResNet-50 trained over the ImageNet dataset.

| Model architectures       | Test Acc. Top1 | Test Acc. Top5 |
|---------------------------|----------------|----------------|
| Low-rank ResNet-50        | 71.03%         | 90.26%         |
| Hybrid ResNet-50 (wo. vanilla warm-up) | 75.85%         | 92.96%         |
| Hybrid ResNet-50 (w. vanilla warm-up) | **76.43%**     | **93.10%**     |

Table 22. The effect of vanilla warm-up training and hybrid network architectures of PUFFERFISH of the low rank VGG-19-BN trained over the CIFAR-10 dataset. Results are averaged across 3 independent trials with different random seeds.

| Model architectures       | Test Loss       | Test Accuracy   |
|---------------------------|-----------------|-----------------|
| Low-rank VGG-19-BN        | 0.355 ± 0.012   | 93.34 ± 0.08%   |
| Hybrid VGG-19-BN (wo. vanilla warm-up) | 0.407 ± 0.008   | 93.53 ± 0.13%   |
| Hybrid VGG-19-BN (w. vanilla warm-up) | 0.375 ± 0.019   | **93.89 ± 0.14%** |