Beyond Human-Level Accuracy: Computational Challenges in Deep Learning

Joel Hestness, Newsha Ardalani, Gregory Diamos

Baidu Research
Sunnyvale, California, USA
{joel,newsha,gregdiamos}@baidu.com

Abstract

Deep learning (DL) research yields accuracy and product improvements from both model architecture changes and scale: larger data sets and models, and more computation. For hardware design, it is difficult to predict DL model changes. However, recent prior work shows that as dataset sizes grow, DL model accuracy and model size grow predictably. This paper leverages the prior work to project the dataset and model size growth required to advance DL accuracy beyond human-level, to frontier targets defined by machine learning experts. Datasets will need to grow 33–971×, while models will need to grow 6.6–456× to achieve target accuracies.

We further characterize and project the computational requirements to train these applications at scale. Our characterization reveals an important segmentation of DL training challenges for recurrent neural networks (RNNs) that contrasts with prior studies of deep convolutional networks. RNNs will have comparatively moderate operational intensities and very large memory footprint requirements. In contrast to emerging accelerator designs, large-scale RNN training characteristics suggest designs with significantly larger memory capacity and on-chip caches.

CCS Concepts • **Computing methodologies** → **Model development and analysis**; *Machine learning*; • **Theory of computation** → Machine learning theory;

Keywords deep learning, neural networks, compute graph, compute requirements, data parallelism, model parallelism

1 Introduction

Deep learning (DL) has emerged as a primary driver of recent artificial intelligence (AI) breakthroughs. As DL-enabled

products grow, it becomes more important to satisfy the future hardware requirements of DL model training. We aim to develop systems that meet these future DL requirements.

DL accuracy advances can come from two different drivers. First, DL researchers study model architecture changes to better fit data sets and improve accuracy. Model changes tend to be highly non-trivial—often requiring problem reframing—and can substantially change their computational structure. As a result, it is very difficult to predict the model structures that will be important for future DL applications.

However, DL advances can also come from growing data sets, model size, and computation—an approach that has received more attention in recent research. The DL community commonly accepts that model accuracy improves as training dataset size grows (e.g., [6, 7, 36]). Further, Hestness et al. characterize accuracy and model size growth, showing they are particular power-law functions of dataset size [18].

This paper leverages the prior work to project the data and model size scaling required to advance DL accuracy beyond human-level, to frontier targets defined by machine learning experts. We collect these accuracy targets for five DL domains—word and character language modeling, machine translation, speech recognition, and image classification.

These domains will require substantial increases in dataset and model size to achieve target accuracy. Datasets will need to grow in size 33–971× larger than the datasets used to train current state-of-the-art (SOTA) models. Models must also grow in parameter count by 6.6–456× larger. Based on these desired targets, simple estimates suggest that training time would take decades to centuries on current systems.

Not shying away from the challenge, this paper characterizes and projects the growth in computational requirements to train these target applications. Although some DL applications are computationally well-understood, our broader analysis reveals surprisingly predictable compute and memory scaling across a range of very different DL architectures, including deep convolutional networks (CNNs), recurrent sequence-to-sequence models, and recurrent encoder-decoder models with attention.

Our characterization reveals an important segmentation of DL training challenges. While prior works have focused heavily on CNNs, their compute requirements differ significantly from recurrent neural networks (RNNs), which are likely to demand far more compute and memory resources.

Image processing applications with deep CNNs desire relatively small growth in dataset and model size, and they show more potential to leverage emerging compute accelerators with high compute-to-memory throughput ratios. Even small batch sizes can expose sufficient operational intensity for high compute throughput.

On the other hand, RNNs, especially in language domains, will require upwards of 100× more training time to achieve target accuracy. They have moderate operational intensities, and very large memory footprints that exceed current accelerator memory capacity by 8–100×. These characteristics make it difficult to efficiently parallelize large-scale training, as we demonstrate in a language modeling case study.

We recommend the hardware community place more focus on supporting RNNs. Systems for RNN training could be substantially different than emerging hardware. For example, a possible approach to better support large-scale RNN training parallelism would be to significantly increase accelerator memory capacity. We could also better leverage growing accelerator compute throughput by building larger on-chip caches to avoid excessive memory data streaming for large matrix multiply operations. These approaches run counter to emerging accelerator designs.

2 Deep Learning Applications

The deep learning research community has developed a large set of important DL applications. The initial release of MLPerf [24] identifies seven domains critical for industry DL training: image classification and object detection, recommendations, reinforcement learning in games, language understanding sentiment analysis, speech recognition, and translation. This paper focus on some MLPerf applications and a similar breadth: image classification, language modeling, speech recognition, and translation. This section describes the general algorithmic structure of DL applications, and describes the particular applications for which we study scaling behaviors.

2.1 Compute Graphs of DL Applications

Deep learning applications are usually structured algorithmically as compute graphs. These compute graphs include nodes, or "ops", that perform a mathematical computation—e.g., matrix-vector multiplication, convolution, or pointwise operations—on input data. Boxes in the network diagrams below represent ops or groups of ops. Data is passed between ops using "tensors" (like data arrays) that encode the data's structure and dependencies between ops.

To project future hardware needs, we define three properties of the compute graphs that allow us to characterize their compute and memory requirements. In practice, when executing a compute graph on hardware, numerous hardware factors affect performance and are difficult or impossible to model (e.g., memory/cache hierarchy, addressing modes, kernel optimization). Rather than trying to model each of these factors for all kinds of hardware, we choose to define **algorithmic** compute requirements, which are independent from particular choices of hardware:

Algorithmic FLOPs are the number of FLOPs required to perform the mathematical calculation of a compute graph op (note: either floating point or integer arithmetic). For example, algorithmic FLOPs include the multiplies and accumulations in a matrix multiply op. Algorithmic FLOPs do not include other instructions executed by hardware to perform the computation, such as address, loop invariant, or branch target calculations. Hardware instructions that are not counted in algorithmic FLOPs are likely to account for at most constant overhead per algorithmic FLOP.

Unlike more general applications, DL compute graphs also perform backward propagation ("backprop") of gradients from the model's predictions. Ops in a DL compute graph are differentiable, so that the gradient of each input can be calculated when given gradients of the outputs. After backprop, accumulated gradients are used to update weights and improve the model's predictions. A compute graph's backprop has highly analogous ops to the forward graph traversal, but it splits gradients to flow to model weights and to activations. The backprop for matrix operations usually has twice the algorithmic FLOPs as the forward traversal.

Analogously, we define an op's **algorithmic bytes accessed** as the total memory bytes that an op must read as inputs and write as outputs to perform the operation. Algorithmic op bytes do not include intermediate data or other memory that might be used to perform the operations, and ignores hardware effects such as caching.

We also define **algorithmic memory footprint** as the minimum number of memory bytes that must be allocated to execute a training step. More precisely, it is the minimum—over all correct topological compute graph traversals—of the maximum memory capacity required to accommodate all active tensors during any step of the traversal. Active tensors are those produced by an op in a previous traversal step, but not yet consumed by each of its downstream ops.

Finally, **algorithmic IO** counts the amount of data accessed for input to and output from a model. Training data is often stored on disks, read from the disk, and placed into the model's input memory allocations. Algorithmic IO is proportional to the batch size, but stays fixed as model size and training step compute requirements grow. We do not investigate algorithmic IO further in this work, because we expect IO will grow very slowly relative to compute.

2.2 Image Classification

ResNets are recognized as high-accuracy convolutional networks (CNNs) for image classification and processing [17]. The ResNet bottleneck architecture, diagrammed in Figure 1, shows the generic structure of residual groups that allow the model grow in depth. Each residual group contains blocks
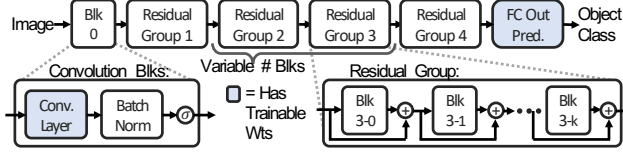
Figure 1. ResNet Bottleneck Abstract Architecture.

of layers, as well as skip connections that permit activations to bypass the blocks. Blocks contain convolutions (with trainable weights designated in blue), batch normalization, and nonlinearities. The final layer is a fully-connected (FC) projection that maps its input to the object classes. These networks tend to be compute intensive due to their depth (50+ convolutions with 64–2048 filters each). However, as we show later, the following recurrent networks can also require significant compute.

2.3 Language Modeling

Language models (LMs) predict the next word or character given a previous sequence of input text. Most computation in these RNNs occurs in recurrent layer matrix multiplies.

Word Language Models:
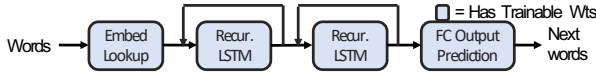
Figure 2. Word LM Recurrent LSTM Abstract Architecture.

LSTM-based RNNs are the SOTA architecture for word LMs [19]. Figure 2 shows the word LM LSTM generic architecture: an embedding layer followed by recurrent layers that feed a FC output layer. The embedding layer is a table lookup operation with no algorithmic FLOPs, but it accounts for a large portion of overall weight memory footprint. The LSTM layers are moderately compute-intensive due to their many matrix multiplications in separate recurrent steps. Finally the FC output layer is compute-intensive and responsible for a large portion of activation memory footprint.

Character Language Models:
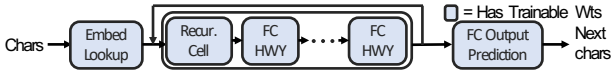
Figure 3. Character LM RHN Abstract Architecture.

Recurrent-highway networks (RHN) have been shown to provide low character perplexities for character LMs [41]. This character LM architecture, as depicted in Figure 3, is a sequence of embedding layer, followed by a deep RHN layer, followed by an output layer. Unlike word LMs, embedding layer and output layer account for a small portion of run time and memory footprint as the vocabulary size (number of characters) is significantly smaller. Each RHN layer contains a sequence of feed-forward sublayers, and the last sublayer output feeds into the next time-step. These layers tend to be compute-intensive and responsible for a large portion of activation memory footprint, especially given their many recurrent steps (100–300 per sample).

2.4 Neural Machine Translation (NMT)
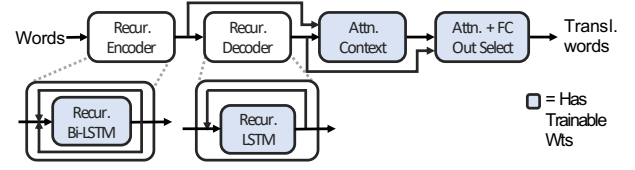
Figure 4. NMT Attention Abstract Architecture.

The SOTA models for neural machine translation use encoder-decoder architectures with an attention mechanism to identify important recent time steps [22]. Figure 4 diagrams such an architecture, which uses a recurrent bi-directional LSTM in the encoder and a standard LSTM for the decoder cell. The encoder and decoder feed the attention context and selection layers that choose the best decoder outputs to predict translated words. Most compute and memory access comes from the recurrent cells in this model.

2.5 Speech Recognition
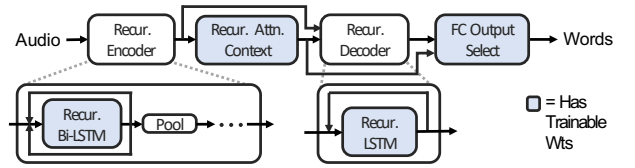
Figure 5. Speech Attention Abstract Architecture.

We investigate the hybrid attention speech model diagrammed in Figure 5 [8]. Like NMT, this model is an encoder-decoder model with attention, though the encoder is a multi-layer bi-directional LSTM with intermediate pooling layers. Most computation occurs in these encoder layers. This model contains convolutions in its attention context layer, but they are very small relative to recurrent portions of network.

3 Application Accuracy Scaling

The DL community has progressively increased dataset and model sizes, and the future system demands of DL training will continue to grow. We would like to project the computational requirements of future DL applications based on the way we expect applications to grow. Recent prior work allows us to project application-level characteristics using analytical models that show the relationships between DL dataset size, model size, and model accuracy. We collect desirable accuracy targets and use the analytical models to predict the dataset and model sizes required to achieve the target accuracy. Compared to current SOTA, DL domains would like 33−971× as much data and 6.6−456× larger models!

3.1 Motivation to Grow Data and Models

The DL community has continually grown datasets, with open-source sets larger than 10s of GBs, to increase modeling task difficulty and model accuracy. Industry is already using significantly larger datasets. Google Research recently showed the importance of training image classifiers with 300× more images than prior datasets [36]. Baidu's prior

work uses speech recognition datasets of multiple terabytes [15]. Google has also stated they want to train language models on a trillion word dataset [32]. Such datasets of interest to DL industry are upwards of 5TB, or about 50×+ larger than current publicly available datasets.

As datasets grow, DL models must also grow to fit the larger datasets, and industry is aiming for very large models. Google states they would like to train a trillion parameter model on a trillion word dataset [32]. That same work proposes a method to compose numerous small models to reach the trillion parameter size. Our projections indicate models will easily reach into the 100s of billions of parameters. Such models would be 10–500× larger than DL models described in current research.

3.2 Accuracy Scaling with Training Data Growth

Recent work indicates why industry wants to increase dataset and model sizes. Hestness et al. show that on real datasets, DL model accuracy improves predictably with training dataset size [18]. They further show that the model size required to fit the data grows predictably with data size. Industry can use these empirical models to estimate the amount of training data and model sizes required to achieve particular accuracy.

Figure 6 shows a sketch of a model's learning curve—the reduction in prediction error as datasets grow[1]. The curve begins in the **small data region**, where models can only perform as well as "best" guessing for the output data distribution. The **power-law region** is where each new training sample offers information to help models improve predictions on previously unseen samples. Error declines predictably. Finally, for real applications, curves are likely to end in an **irreducible region** where models cannot further improve due to the stochastic nature of the data.
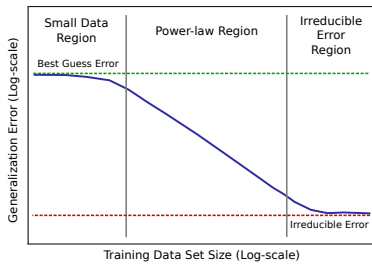
Figure 6. Sketch of power-law learning curves.

In particular, we project learning curves starting from the power-law region, where most existing large-scale data applications are currently. In this region, model generalization error scales roughly as a power law:

$$\varepsilon(m) \approx \alpha m^{\beta_g} \qquad (1)$$

Here, m is the number of samples in the training dataset, and α and $\beta_g \in [-0.5, 0]$ are constants that depend on the

---

[1]Copied with author permission [18]

structure of the modeling task, possibly including the data distribution and model architecture. α represents aspects of the input data space and the DL model architecture. β_g is the power-law exponent and indicates the difficulty for models to learn more information from each additional training example. β_g closer to -0.5 means models can learn quickly from smaller datasets. Table 1 lists estimates of α and β_g for the different modeling tasks as found in the prior work [18].

To extend the work of Hestness et al. to predict the required data and model size from these models, we need to define accuracy targets that would be desirable for DL-enabled products. We collect feedback from DL experts and refer to prior studies that estimate the irreducible error to select desirable accuracy targets for each domain. For example, word and character LM desired SOTA are near estimated lower bounds on the entropy of English text [31]. The "Desired SOTA" column of Table 1 reflects these projections.

Finally, given these analytical learning curves and target error rates, we solve the analytical models for the required data size to realize the target. The "Projected Scale" columns in Table 1 show the relative data size projections. Desired SOTA values are 1.4× to 3.9× better than current SOTA values. However, the amount of data required to achieve these values range from 33× more for speech recognition to 971× more for character LMs. Language domains require the most data due to their poorer power-law exponents, β_g.

3.3 Model Size Scaling with Training Data Growth

As datasets grow in size, models must also grow in size to represent the data. Hestness et al. also collect and characterize model sizes required to fit varying training set sizes. Model parameters (roughly capacity) are expected to grow sublinearly in the training set size with the following form:

$$p(m) \approx \sigma m^{\beta_p} \qquad (2)$$

Here, m is the the number of samples in the training set, and σ and $\beta_p \in [0.5, 1)$ are constants that depend on the problem structure, possibly including data distribution and model architecture. Models should grow parameter count more slowly than the training set (i.e., $\beta_p \leq 1$), or we could just store the dataset rather than training a model. Recent prior work shows that deep neural network model capacity—the volume of concepts (data) it can learn—grows with $O(lp \log p)$, where l is a measure of the model's depth [16]. Loosening this bound slightly, model size should grow at least with a square root of the dataset size (i.e., $\beta_p \geq 0.5$).

Table 1 shows empirically collected σ and β_p for the DL domains [18]. Given the target data size determined in the last subsection, we project the model sizes required to fit the target dataset sizes. The model scale column shows the relative required increase in model size. For example, current SOTA word LMs use roughly $1B$ parameters to fit roughly $1B$ word datasets. Thus, to fit a 100× larger dataset, a model would require ~$23B$ parameters (23–92GB, depending on weight precision).

Table 1. Learning Curve and Model Size Scaling Relationships for DL Domains

Domain (model)	Current SOTA	Desired SOTA	Current Data Size Samples	GB	Learn Curve α	β_g	Model Size σ	β_p	Projected Scale Data	Model
Word LMs (LSTM)	3.37 nat/word	2.48 [31]	768M word	3.9	13.0	−0.066	9.4e−4	0.68	100×	23×
Character LMs (RHN)	1.30 bit/char	0.70 [31]	3.48B char.	3.9	9.39	−0.092	1.2e−5	0.89	971×	456×
NMT (enc/dec+attn)	28% WPER	12%	130M WP	2.6	3.06	−0.128	6.4e−4	0.68	750×	90×
Speech Recogn. (enc/dec+attn)	9.5% CER	4% [39]	425M char.	1674	30.5	−0.291	2.4e−3	0.54	33×	6.6×
Image Classification (ResNet)	19.4% Top-1	5% [29]	1.3M image	152	15.0	−0.309	2.0e−2	0.57	81×	12×

4 Characterizing Compute Requirements

Now that we have an idea of desirable data and model sizes, we turn our attention to characterizing the computational requirements to train these very large models. This section characterizes DL application compute FLOP, memory access, and memory footprint growth. Although the structure of DL applications is intricate, their training requirements scale mostly predictably. Compute and memory usage grow asymptotically linearly with model size and batch size. We provide these accessible first-order models of compute requirements not characterized in prior work.

4.1 Methodology

We estimate model training compute requirements by collecting statistics from training runs and assembling analytical models to project growth. We train with Tensorflow 1.5.0 [4] running on NVIDIA GPUs and using a modified version of TFprof. TFprof annotates compute graph ops to calculate their algorithmic FLOPs and bytes, and collect run time as they execute. At the end of a training step (i.e., a compute graph traversal), TFprof returns this profile for all ops executed during the step, ensuring that we profile even fine details of an end-to-end training step. We also query Tensorflow's memory allocators for the maximum amount of training step memory allocated—the memory footprint.

We collect profiles from 100-500 randomly-chosen training steps to account for per-training-step differences in FLOPs and memory accessed for different models. For instance, character LMs, NMT, and speech models unroll their recurrent layers for the time-steps required for the longest batch sample. This unrolling results in variable computation and memory access in separate training steps, so we average the profiled results over the training steps.

The most complicated variable to control for is training batch size—the number of data parallel samples to observe in a single training step. Batch size can be set arbitrarily, but particular batch sizes result in best model accuracy depending on data set size [35]. For tested domains in this study, SOTA models have been trained using data parallelism across GPUs to increase batch size beyond the maximum memory capacity of a single GPU. It is likely that future DL training will also be constrained by per-compute-unit memory capacity, suggesting that ML researchers will choose per-compute-unit batch sizes (henceforth, "**subbatch size**") that

can provide near-peak utilization of compute unit resources. We profile with the smallest such subbatch size.

To grow models, we change hyperparameters that have the largest effect on the ability for the model to fit larger data sets as measured by generalization error. For ResNets, increasing depth and convolution channels, rather than filter sizes, improves accuracy the most, so we collect profiles for deeper and wider image classification networks. Most recurrent models have already grown to a depth such that increased depth results in no accuracy improvement. Instead, we increase the number of hidden weights per layer.

Finally, we aim to project forward the compute requirements for models as we scale up data set and model size. The analytical models of application characteristics below use first-order approximations to provide the community with a concise set of formulas for projections. However, we also use high-fidelity modeling to verify these results (Appendix A).

4.2 Estimating Training Step Algorithmic FLOPs

For DL models, the number of FLOPs per training step grows roughly linearly in the number of parameters of the model, suggesting that each model parameter is used roughly the same number of times in a single training step. We demonstrate this observation analytically for word LMs next.

Again, let p be the number of model parameters for a LSTM word LM, and let p_{em}, p_{re}, and p_o be the parameters in embedding, recurrent, and output layers, respectively. We approximate the total model parameters as:

$$p = p_{em} + p_{re} + p_o \approx hv + 8h^2l + hv = 8h^2l + 2hv$$

Here, v is the LM's vocabulary size, h is number of hidden weights per recurrent layer, and l is the number of layers.

Next, we show the roughly linear relationship between parameters and FLOPs per step. Since backward propagation adds ~2× the number of FLOPs, regardless of the model, we consider only the forward propagation. For this first-order model, we assume that most compute FLOPs come from the subset of ops that perform vector or matrix operations. We estimate forward propagation algorithmic FLOPs:

$$c_{fwd} = c_{em} + c_{re} + c_o \approx 0 + 16lqh^2 + 2qvh = q(16h^2l + 2hv)$$

Here, q is the sequence length for the training step (we ignore subbatch size to normalize per training sample). These models indicate that $\lim_{h\to\infty} \frac{c_{fwd}}{p} \to k$, a constant. Thus, we expect that for word LMs and similarly structured recurrent

models, compute FLOPs should grow roughly linearly in the increase in number of model parameters.

We confirm this linear relationship between model parameters and algorithmic FLOPs per training step empirically across our set of applications. Figure 7 plots the TFprof-profiled growth in algorithmic FLOPs (note: batched training roughly multiplies these values by the subbatch size). Each domain's algorithmic FLOPs grow linearly with model size above 30–100M parameters—moderately large models. FLOPs per parameter ranges from 149 for NMT to 1111 for ResNets. For recurrent networks, as sequence length grows, the FLOPs/parameter also grows, approaching ResNet requirements. Character LMs and speech networks unroll layers for 150 and 300 time-steps, respectively.
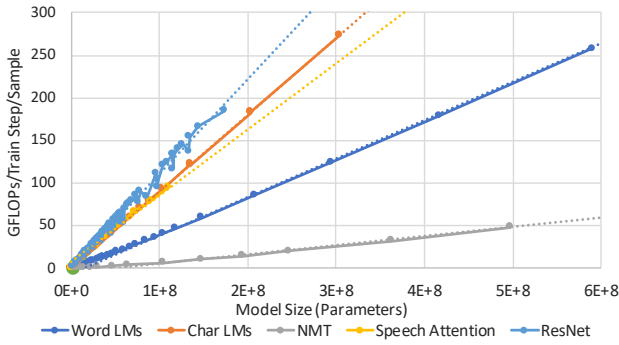
Figure 7. Per-training sample FLOPs growth with number of model parameters, all models (dotted lines are trends).

Table 2 records the asymptotic hardware requirements for each DL domain as models grow. Given the clear linear relationships between FLOPs and parameter counts, we use the following linear trend to project the compute FLOPs per training sample ("c_t") for models with p parameters:

$$c_t(p) \approx \gamma p$$

Here, γ is a constant that depends on the input data shape, recurrent sequence length, and model architecture.

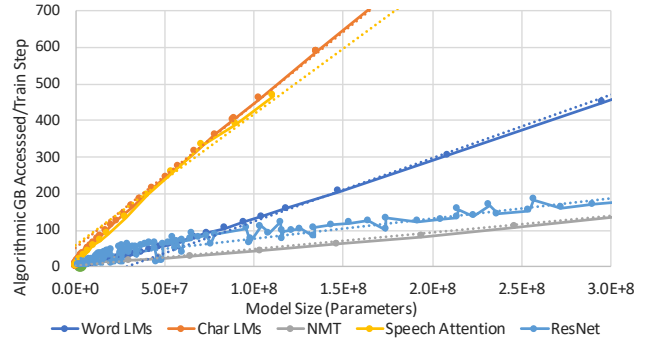### 4.3 Estimating Algorithmic Memory Bytes Accessed

Like algorithmic FLOP counts, algorithmic memory accesses also scale linearly with model parameters across the DL applications. However, since a significant portion of training step memory accesses are from reading or updating model weights—which do not depend on the subbatch size—memory access counts depend, to first-order, on both model size and subbatch size. This section describes an analytical model and verifies that it fits empirical results.

A training step must access two types of tensors: the DL model and the activation tensors that flow through the model. Hardware loads from and stores to the model parameters roughly a constant number of times each for the forward and backward propagation, and to update the weights at the end of a training step. Similarly, activation memory, with dimensions proportional to the batch size and model dimensions,

is accessed roughly a constant number of times. As above, denote s as the model parameter count. Then total memory accesses for a training step ("a_t") takes this first-order form[2]:

$$a_t(p, b) \approx \lambda p + \mu b \sqrt{p}$$

Here, λ and μ are constants that depend on input data shape, recurrent sequence length, and model architecture. The $\sqrt{p}$ term approximates the model's hidden layer weight or channel counts—one dimension of the compute graph's large linear algebra ops. We find $\sqrt{p}$ is a good approximation for all domains, with a small caveat: For models with many parameters to embed input data (e.g., the larger vocabularies of word LMs and NMT), $\sqrt{p}$ over-estimates hidden dimension until the hidden dimension is large relative to the embedding dimension. Figure 8 curves show nearly linear asymptotes.

Figure 8. TFprof algorithmic memory access as model size grows (note: for particular subbatch sizes).

4.4 Estimating Training Operational Intensity

Conveniently, although model training steps are composed of many ops, their algorithmic FLOPs and memory access characteristics are strikingly similar to those of a single large linear algebra operation. As a result, operational intensity—the ratio of FLOPs to memory bytes accessed—takes form familiar in linear algebra kernel optimization.

Algorithmic operational intensity for each DL model is listed in Table 2. A model's ops that contribute the most to FLOPs and memory accesses are often matrix operations with dimensions related to the hidden dimension ($\sim\sqrt{p}$) and subbatch size. The operational intensity of a matrix multiplication with dimensions $(b \times \sqrt{p})(\sqrt{p} \times \sqrt{p})$ is $b\sqrt{p}/(2\sqrt{p} + 4b)$, the same form as the end-to-end training step operational intensities listed in Table 2.

As a result of its form, operational intensity will approach some fixed upper bound unless both a model's hidden dimension and the subbatch size grow. When either model size or subbatch size is fixed, it will asymptotically approach the ratio of the slopes of algorithmic FLOPs and bytes growth. Figure 9 shows the leveling of operational intensity for fixed subbatch size as model size grows for each application.

---

[2]Supplemental material shows detailed calculation for word LMs

Table 2. Asymptotic Application-level Compute Requirements

Domain (model)	Alg. compute (FLOPs/Param)	Alg. memory access (Bytes/Param)	Alg. operational intensity (FLOPs/Byte)	Minimal Mem. Foot (Bytes/Param)
Word LMs (LSTM)	$481\,b$	$1755 + 30784\,b/\sqrt{p}$	$b\sqrt{p}/(3.65\sqrt{p} + 64\,b)$	11.94
Character LMs (RHN)	$900\,b$	$3510 + 102980\,b/\sqrt{p}$	$b\sqrt{p}/(3.9\sqrt{p} + 114\,b)$	12.47
NMT (enc/dec+attn)	$149\,b$	$533 + 22653\,b/\sqrt{p}$	$b\sqrt{p}/(3.6\sqrt{p} + 151\,b)$	10.32
Speech Recogn. (enc/dec+attn)	$775\,b$	$3100 + 162750\,b/\sqrt{p}$	$b\sqrt{p}/(4.0\sqrt{p} + 210\,b)$	32.94
Image Classification (ResNet)	$1111\,b$	$66.7 + 268862\,b/\sqrt{p}$	$b\sqrt{p}/(0.06\sqrt{p} + 242\,b)$	42.57
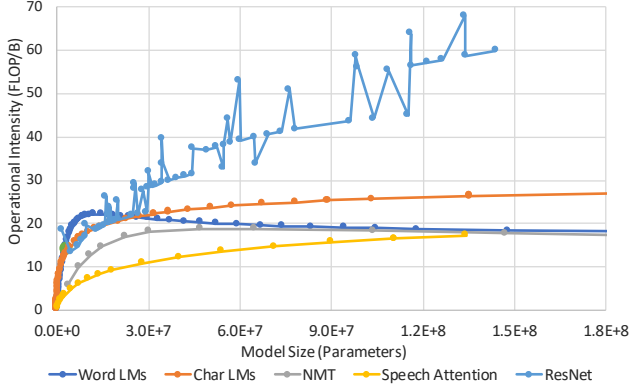
Figure 9. TFprof algorithmic operational intensity as model size grows (note: fixed subbatch size).

Figure 10. Empirical and asymptotic minimal memory footprint as model size grows (note: fixed subbatch size).

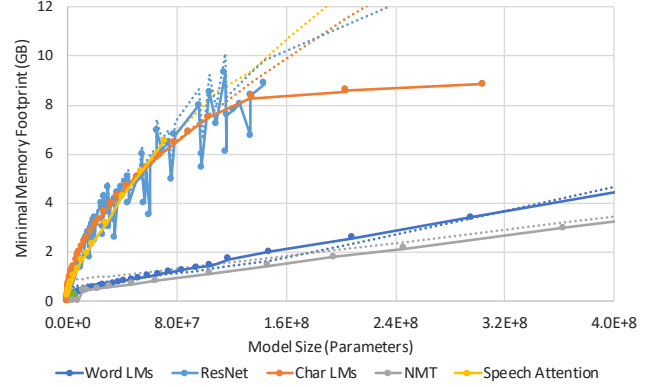### 4.5 Estimating Training Step Memory Footprint

Memory footprint is the measure of the memory capacity required to execute an algorithm. Tensorflow's memory allocator provides a footprint estimate, but we also estimate minimal memory footprint by tracking it through a topological traversal of the compute graph. For each op, DL frameworks can allocate memory for the op's output tensors, and after executing the op, free the op's input tensors if all the tensor's consumer ops have executed.

Figure 10 plots the Tensorflow allocator memory footprints for each model and our topological estimates. These values agree up to the point that Tensorflow runs out of GPU memory capacity (80% of 12GB). At that point, the allocator starts swapping GPU memory to the CPU's memory space, where it no longer counts the memory as part of the footprint. When Tensorflow does not swap memory, our models tend to slightly overestimate minimal memory footprint; Tensorflow optimizes to perform some ops on tensors in-place rather than allocating separate output tensors.

Minimal memory footprint grows asymptotically linearly with model size for larger models. This trend is expected given that the model's parameters dominate memory and are persistent, while activation tensors can be freed and reused by the framework. We model minimal footprint linearly:

$$f_t(p) \approx \delta p$$

Here, δ is a constant dependent on the input data shape, recurrent sequence length, and model architecture. This first-order approximation fits well for parameter counts above

~$500M$, but for our projections in the next section, we opt to use more accurate topological traversal estimates.

Language model footprint growth is similar across the domains; character LM footprint growth slows significantly for large models (not depicted in the figure). Speech and image domains show faster memory footprint growth with model size. However, as the next section shows, speech and image domains need much smaller networks to achieve accuracy targets, so their footprint requirements are modest.

5 Projecting the Accuracy Frontier

Here, we project the compute resources required to train models to target accuracy levels. We also project a hypothetical Roofline estimate of model training time and discuss implications of the resource requirements. Improving speech recognition and image classification should be feasible with existing parallelism strategies. Language domains, however, are likely to require 100× more compute, suggesting the need for both improved algorithmic and parallelism strategies.

5.1 Projecting Target Compute Requirements

Using the analytical models from the last two sections, we project compute resource requirements to reach target accuracy levels. Table 3 lists the projected data and model size, our choice of subbatch sizes (Section 5.2.1), and projected training requirements.

We expect that image processing networks will require the least growth in algorithmic FLOPs and memory access per training step to achieve aggressive accuracy targets. Their required model growth is small relative to recurrent networks, and their convolutional layers offer high operational

Table 3. Application-level Training Requirements Projected to Target Accuracy

| Domain (model) | Projected | | Sub-batch Size | Training Step | | | Accel. Time | |
	Data Size	Model Params.		TFLOPs/ Step	Mem Acc. TB/Step	Min Mem Foot (GB)	Step (secs)	Epoch (days)
Word LMs (LSTM)	77B word	23.8B	128	1444	41.5	272	**115**	**31K**
Character LMs (RHN)	3.4T char.	146B	96	12618	488.1	1703	**1007**	**3.5M**
NMT (enc/dec+attn)	97.4B WP	18.9B	96	499	18.4	185	**39.8**	**16K**
Speech Recogn. (enc/dec+attn)	14B char.	727M	128	72	2.8	30	5.8	93
Image Classification (ResNet)	103M image	732M	32	28	0.4	34	2.3	84

intensity to utilize compute resources with smaller subbatch sizes. The clearest contrast is with speech recognition, which would require similar model size as image classification, but its larger subbatch size means more FLOPs and memory access per training step. These results suggest it may be easier to parallelize very large image network training by sharding full batches across many accelerators.

The projected compute requirements also witness the challenges of scaling language domains specifically, and recurrent networks in general. To reach target accuracy on language and speech domains will require 2.5–1200× more FLOPs and memory access per training step than image classification. In language domains, these increases are largely due to the model size growth required to fit larger data sets.

Finally, we note that all domains are likely to require significantly more memory capacity than available with current accelerators. Current GPUs and Google's TPU v2 have 16 or 32GB of memory per accelerator chip [13]. Running any of these models on such accelerators will require either model-level parallelism to split portions of the models across multiple accelerator's memories, or migrating model parts into and out of accelerator memory—an expensive operation.

5.2 Projecting Run Time on Hardware

Next, we estimate hypothetical best-case run times for each of the target applications running on an accelerator. We configure a target accelerator, describe our process for choosing the training step subbatch size, and then estimate run time. The estimates use the Roofline model to predict the overall system throughput given the full-graph algorithmic FLOPs and memory accesses [38].

Table 4. Target Accelerator Configuration

Component	Configuration
Compute Throughput, 32-bit (x_c^{32b})	15.67 TFLOP/s
On-chip Cache	6 MB
Memory Bandwidth (x_a)	898 GB/s
Memory Capacity (off-chip)	32 GB
Inter-device Bandwidth	56 GB/s

Table 4 shows the configuration for a target accelerator similar to NVIDIA's V100 version 2. We assume maximum achievable throughput of 80% of peak FLOPs and 70% of peak memory bandwidth, consistent with existing hardware. The accelerator's compute intensity inflection point between
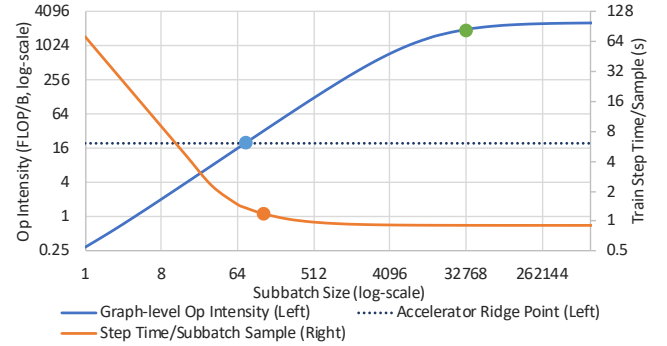
Figure 11. Subbatch size effect on word LM operational intensity, training step time per sample. Marginal run time gains for op intensity higher than accelerator ridge point.

memory-bound and bandwidth-bound (its Roofline "**ridge point**") is 17.4 FLOP/B, but given peak achievable throughput, rises to 19.9 FLOP/B. We start by assuming that the accelerator has infinite memory capacity and is able to fit the memory footprint for a training step of any model.

5.2.1 Subbatch Size: Minimize Per-Sample Time

Choosing an appropriate subbatch size for model training is a difficult process that depends on many aspects of the DL application. Here, we focus on the hardware trade-offs: we want to ensure good utilization of the accelerator while keeping a small memory footprint. We identify three subbatch size points-of-interest and show that the smallest size that minimizes per-sample latency (i.e., maximizes throughput) provides the best trade-offs.

Figure 11 shows the effect of subbatch size on the graph-level operational intensity and the training step time per-subbatch-sample. We could choose subbatch size such that the graph-level operational intensity nears saturation (green marker), giving the most opportunity to utilize the accelerator's compute throughput. However, this point also requires a very large memory footprint, often 5–20× more than a small subbatch. Another option is subbatch size such that the graph-level operational intensity matches the accelerator's ridge point (blue marker). In practice, however, this point does not optimize the accelerator's compute throughput—many ops are still memory-bound. The training step time-per-sample curve (orange) shows 40% throughput loss.

Instead, we prefer subbatch size that minimizes the training step time normalized per-sample. The orange point in

Figure 11 is a subbatch size that keeps memory footprint small while achieving 79% of the peak compute throughput. We use this approach to estimate best subbatch sizes for each domain in Table 3. For recurrent networks, subbatch size settles at about 1.5× larger than the point where graph-level operational intensity matches the accelerator's ridge point.

5.2.2 Per-epoch Run Time

Finally, we estimate best-case run time using a Roofline model—performance is bounded either by the accelerator's compute (x_c) or memory access (x_a) throughput:

$$r_t(x_c, x_a) = max\left(\frac{c_t}{80\% \cdot x_c}, \frac{a_t}{70\% \cdot x_a}\right)$$

We list training step time in Table 3, and project these out to the training time for one epoch. These estimates were also used for selecting subbatch sizes.

Although optimistic, these training time projections show that the target accuracies for image classification and speech recognition may not be far out of reach. A single epoch would take ~3 months on a single accelerator. Reducing epoch time to less than a day would require parallelizing training over ~100 accelerators—activity becoming more common in recent data parallelism work.

However, major challenges exist in language domains with epoch times of 40–9600 years on a single accelerator. To achieve target accuracies for these domains will require significant innovation beyond existing parallelization strategies.

6 Case Study: Word LMs at the Frontier

Language and translation domains are among the most challenging problems we have tested. Our results show they may require models of ~20–200B parameters and ~100× more compute than other domains. This section works through a case study of word LMs to consider the challenges and potential approaches to scale to their frontier accuracy. A combination of algorithmic and parallelism optimizations is required to train a target word LM in 7 days per epoch.

6.1 Setting the Baseline Train Time

We begin by setting the baseline training time. Since the prior word LM would take ~84 years per epoch to train, we start by choosing an algorithmic optimization used in recent word models: LSTM projection [30]. The projected LSTM reduces the inner dimension of the last hidden layer before feeding it to the output layer. We also increase the vocabulary size to match prior work [19]. These changes reduce the per-training-step FLOPs, memory accesses, reducing the roofline time by a factor of 11.7× to ~9.89s on the target accelerator. Table 5 records our process of parallelizing the word LM, starting with this best-case 2707 days per epoch.

We also make our target application more realistic by adding simple modeling for memory accesses in matrix arithmetic. Unfortunately, algorithmic memory accesses underestimate total memory accesses for ops that perform large
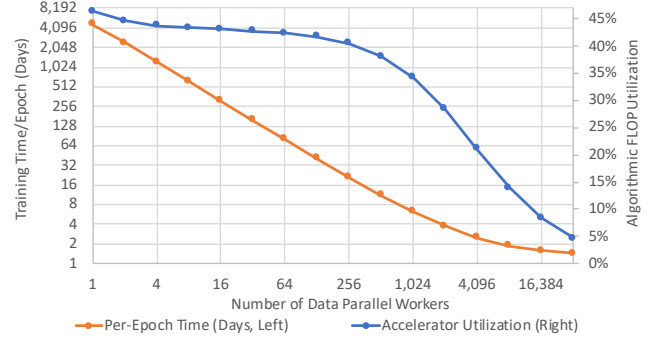
Figure 12. Data parallelism effect on overall run time and utilization at subbatch = 128.

matrix multiplications; portions of the input tensors can be stored in on-chip caches of the accelerator, but large-tensor multiplies will require re-streaming significant portions of the inputs from off-chip memory multiple times. We capture the impact of cache hierarchy on performance by modeling these extra memory accesses, assuming a common, tiled matrix multiply implementation [12]. This cache-hierarchy-aware model predicts per-epoch time would take 4671 days, reducing to 46% algorithmic FLOP utilization. We validate that this model is still optimistic, but reduces maximum prediction error from 42% down to 15% on tested hardware.

6.2 Step-by-Step Parallelism Strategy

There are three major challenges to scale this word LM's training time. First, we will need to reduce training time by 667× (4671 days/7), requiring parallelism across at least this many accelerators. Second, the required memory footprint is too large to fit in a single accelerator's memory, so each data-parallel worker will need to parallelize across at least 4 accelerators (113.8GB per step/32GB capacity per accelerator). Finally, we aim for effective use of resources, and describe a parallelism scheme that keeps accelerator algorithmic FLOPs utilization above 14.5%.

6.2.1 Data Parallelism

We first scale out using data parallelism—the process of dividing batch elements across multiple workers, and then collecting their results to update model weights. The baseline subbatch size is 128, found using the method in Section 5.2.1. We model a synchronous stochastic gradient descent (SGD) approach implemented using a ring-allreduce [25].

SGD communication overheads eventually dominate per-training-step time. As listed in Table 4, we assume accelerators can communicate using high bandwidth inter-device links at 56GB/s, consistent with future intra-node and Infiniband 400Gb inter-node interconnects. Figure 12 shows training time per epoch improves while utilization declines as we increase the number of data-parallel workers.

Modeling results show that word LMs would require at least 1024 accelerators to reduce epoch time to 6.2 days. Utilization declines slightly to 34% at 1024 accelerators due to

Table 5. Step-by-Step Process of Training Word LM to Target Accuracy.

Optimization Stage	Num. Accel.	Batch Size	Accel. Mem. Required (GB)	L2 Cache Capacity	Train Time days/epoch	Alg. FLOP Utilization
Best-case (Roofline) Baseline	1	128	**113.8**	—	2707	80%
Cache-hierarchy-aware Baseline	1	128	**113.8**	6MB	4071	46%
w/ Data Parallelism (Option 1)	1024	131072	**125.7**	6MB	6.2	34%
w/ Data Parallelism (Option 2)	512	65536	**125.7**	6MB	11.1	38%
+ Layer Parallelism (4×)	2048	65536	{**60**, 17, 17, 32}	6MB	7.2	14.5%
+ Shard the Embedding Layer	2048	65536	{32, 31, 31, 32}	6MB	7.2	14.5%

communication overheads for reducing gradients. Recent prior work uses batch sizes up to 32K samples to train image classification [40] and character LMs [26], so we believe batch size 32–128K may be feasible with future techniques. We also list a configuration for 512 accelerator data parallelism as the basis for the next stage.

6.2.2 Model Parallelism

Although data parallelism gets close to our desired training time, it does not address the problem of per-accelerator required memory footprint. At the current optimization stage, each accelerator would require roughly 126GB of memory capacity, so we must divide the model to parallelize training steps across more accelerators.

We consider "layer-wise parallelism", an approach that places separate layers of the model across neighboring accelerators. Since the word LM has 4 layers, we allocate 4 accelerators per data-parallel worker. Starting from the 512 data parallel worker option, we require 2048 total accelerators to add layer parallelism. This approach would reduce epoch time to just over 7 days, and cut the required memory footprint per accelerator by half.

Although layer parallelism reduces the footprint required per accelerator, the embedding layer of the word LM (59.5GB) will not fit in a single accelerator's memory. Prior techniques move the embedding layer to locations with more memory capacity, such as host memory, which will require moving embedded data to the accelerator's memory. Instead, we propose to split the embedding layer into 3 pieces and locate two smaller parts in the memories of accelerators that perform recurrent layer computations. This split evens out the per-accelerator footprints with trivial run time overhead, and results in final algorithmic FLOP utilization of 14.5%.

6.2.3 Discussion and Related Work

This word LM case study highlights the challenges that will exist for scaling RNN model training to frontier-level accuracy. Major opportunities to optimize RNN training all result in the need for more cache or memory capacity.

Memory Capacity: Although emerging accelerators use high-bandwidth memories (HBM), their capacities are currently just 16 or 32GB. Existing CNN applications can utilize compute FLOPs of these accelerators with small memory footprints, so there is little pressure from these applications to increase memory capacity. On the other hand, each of the

language domains show extreme per-accelerator training-step memory footprint that exceeds current memory capacities by 8–100×.

We started with algorithmic optimization to reduce the word LM's memory (and compute) requirements. Accelerator memory capacities are orders of magnitude short of LM requirements, but some algorithmic optimizations may be promising to chip away at this gap. Model compression or distillation, and low-precision or sparse computation may reduce model or activation tensor size [9], and reduce memory requirements by 1.5–10×. Currently, many challenges exist to use these techniques during model training.

Parallelism Techniques: DL frameworks could also provide parallelization techniques to enable more effective use of cache and memory capacity. Researchers have shown how to train image and language applications quickly using data-parallel scaling to reduce per-accelerator activation memory requirements [14, 26, 40]. Prior work also explores other forms of data parallelism [20, 23, 28], and there are opportunities to reduce data communication overheads of parallelism [5, 21, 37]. Layer parallelism can also reduce the memory requirements for model weights [33, 34]. Improved model parallelism techniques could recover some of the ~23% algorithmic FLOP utilization lost to layer parallelism in the case study. Frameworks should aim to automatically and dynamically subdivide the computation, automatically map appropriate compute graph portions to compute resources, and prefetch data between host and accelerator memories.

Operational Intensity: Our modeling also shows that RNN networks suffer from moderate operational intensity in large matrix multiplications. This medium operational intensity is caused by the need to stream inputs from memory multiple times during tiled multiplies, decreasing the algorithmic FLOP utilization by ~40% compared to an ideal system that would not need to restream inputs. In this setting, increasing on-chip cache size may hinder compute throughput growth, but is likely to proportionally reduce input restreaming from memory. Better cache tiling, kernel optimization and fusion techniques might also help [10, 11].

Hardware techniques to better support large-scale RNN training—larger memories and on-chip caches—run counter to emerging accelerator designs. Emerging designs aim to support very high compute-to-memory ratios by optimizing

for compute throughput. This design philosophy is unlikely to trade die area for memory channels (capacity) or caches.

7 Other Related Work

DL Application Characterization: Many benchmark suites have been developed that aim to analyze DL applications by focusing on particular ops/kernels [3] or quantifying the performance of end-to-end DL applications [2, 24]. OpenAI recently characterized the trend in DL FLOP growth over time using coarse approximations of FLOPs for a range of applications ranging from AlexNet to Alpha Go [1].

Performance Modeling Paleo is an analytical performance model which explores parallelism for CNN networks [27].

8 Conclusion

This paper leverages the prior work to project the dataset and model size growth required to advance DL accuracy beyond human-level, to "frontier" targets defined by machine learning experts. Datasets will need to grow by 33–971×, while models will need to grow by 6.6–456× to achieve target accuracies. We project the computational requirements to train these applications at scale. Our results reveal an important segmentation of DL training challenges for recurrent neural networks (RNNs) that contrasts with prior studies of deep convolutional networks. RNNs will have comparatively moderate operational intensities and very large memory footprint requirements. In contrast to emerging accelerators, large-scale RNN training characteristics suggest designs with significantly larger memory capacity and on-chip caches.

References

[1] 2018. AI and Compute. https://blog.openai.com/ai-and-compute

[2] 2018. DAWNBench. https://dawn.cs.stanford.edu/benchmark/

[3] 2018. DeepBench. https://github.com/baidu-research/DeepBench

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[5] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *Advances in Neural Information Processing Systems (NIPS)*. 1709–1720.

[6] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, JingDong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. 2016. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In *The International Conference on Machine Learning (ICML)*. 173–182.

[7] Michele Banko and Eric Brill. 2001. Scaling to Very Very Large Corpora for Natural Language Disambiguation. In *Association of Computational Linguistics (ACL)*.

[8] Eric Battenberg, Jitong Chen, Rewon Child, Adam Coates, Yashesh Gaur, Yi Li, Hairong Liu, Sanjeev Satheesh, David Seetapun, Anuroop

Sriram, and Zhenyao Zhu. 2017. Exploring Neural Transducers for End-to-end Speech Recognition. In *IEEE Automatic Speech Recognition and Understanding Workshop*. 206–213.

[9] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2018. Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges. In *IEEE Signal Processing Magazine*, Vol. 35.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014).

[11] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Andrew Ng. 2013. Deep Learning with COTS HPC Systems. In *International Conference on Machine Learning (ICML)*. 1337–1345.

[12] Stephanie Coleman and Kathryn S. McKinley. 1995. Tile Size Selection Using Cache Organization and Data Layout. In *ACM SIGPLAN Notices*, Vol. 30. ACM, 279–290.

[13] Jeff Dean et al. 2017. Machine Learning for Systems and Systems for Machine Learning. Presentation at ML Systems Workshop with Neural Information Processing Systems (NIPS) Conference. http://learningsys.org/nips17/assets/slides/dean-nips17.pdf

[14] Priya Goyal, Piotr Dollàr, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *Facebook AI Research Publications* (2017).

[15] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep Speech: Scaling Up End-to-End Speech Recognition. *arXiv preprint arXiv:1412.5567* (2014).

[16] Nick Harvey, Peter L. Bartlett, Christopher Liaw, and Abbas Mehrabian. 2017. Nearly-tight VC-dimension and Pseudodimension Bounds for Piecewise Linear Neural Networks. In *The Conference on Learning Theory (COLT)*, Vol. 65. 1064–1068.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[18] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. Deep Learning Scaling is Predictable, Empirically. *arXiv preprint arXiv:1712.00409* (2017).

[19] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the Limits of Language Modeling. *arXiv preprint arXiv:1602.02410v2* (2016).

[20] Alex Krizhevsky. 2014. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997* (2014).

[21] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. 2017. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *The International Conference on Learning Representations (ICLR)*.

[22] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *The Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1412–1421.

[23] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2018. Semantics-Preserving Parallelization of Stochastic Gradient Descent. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 224–233.

[24] MLPerf. 2018. MLPerf: A Broad ML Benchmark Suite for Measuring Performance of ML Software Frameworks, ML Hardware Accelerators, and ML Cloud Platforms. https://mlperf.org/

[25] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal on Parallel and Distributed Computing* 69, 2 (February 2009), 117–124.

[26] Raul Puri, Robert Kirby, Nikolai Yakovenko, and Bryan Catanzaro. 2018. Large Scale Language Modeling: Converging on 40GB of Text in

Four Hours. *arXiv preprint arXiv:1808.01371* (2018).

[27] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *The International Conference on Learning Representations (ICLR)*.

[28] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NIPS)*. 693–701.

[29] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *arXiv preprint arXiv:1409.0575* (January 2015).

[30] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. 2014. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *arXiv preprint arXiv:1402.1128* (2014).

[31] Claude E. Shannon. 1951. Prediction and Entropy of Printed English. *Bell System Technical Journal* 30, 47–51.

[32] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *arXiv preprint arXiv:1701.06538v1* (2017).

[33] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE Computer Society, Los Alamitos, CA, USA*.

[34] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *The International Symposium on Computer Architecture (ISCA)*. ACM, 535–547.

[35] Samuel L. Smith and Quoc V. Le. 2017. A Bayesian Perspective on Generalization and Stochastic Gradient Descent. *arXiv preprint arXiv:1710.06451v2* (October 2017).

[36] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. 2017. Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. In *The International Conference on Computer Vision (ICCV)*.

[37] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *Advances in Neural Information Processing Systems (NIPS)*.

[38] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Communications of the Association of Computing Machinery* 52, 65–76.

[39] Wayne Xiong, Lingfeng Wu, Fil Alleva, Jasha Droppo, Xuedong Huang, and Andreas Stolcke. 2017. *The Microsoft 2017 Conversational Speech Recognition System*. Technical Report. https://www.microsoft.com/en-us/research/publication/microsoft-2017-conversational-speech-recognition-system/

[40] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet Training in Minutes. In *The International Conference on Parallel Processing (ICPP)*. 1:1–1:10.

[41] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. 2017. Recurrent Highway Networks. In *The International Conference on Machine Learning (ICML)*.

A Artifact Appendix

A.1 Abstract

*The artifact contains the latest version of our codebase, called "Catamount", which is a compute graph analysis tool to load, construct, and modify deep learning (DL) models and to symbolically analyze their compute requirements. Catamount can read DL model checkpoints saved from DL frameworks (e.g., from Tensorflow). This artifact includes (A) Tensorflow checkpoints for each of the models (compute graphs) analyzed in the PPoPP 2019 paper, **Beyond Human Level Accuracy: Computational Challenges in Deep Learning** and (B) shell scripts to run graph analytics and generate results for Figures 7 to 10 of the paper. To validate the results, run the test script, which generates and collects the corresponding outputs:*

```
~$ bash catamount/frameworks/example_graphs/tensorflow/full_models/generate_results.sh
```

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

- **Algorithm:** Catamount can construct or load compute graphs and using various graph traversal algorithms, propagate symbolic graph dimensions and calculate various compute requirements, including compute Flops, memory accesses, and memory footprint.
- **Data set:** Consists of compute graph checkpoints from neural network models trained in Tensorflow. The models are a word language model (LSTM), character language model (RHN), neural machine translation (encoder/decoder+attention), speech recognition (encoder/decoder+attention), and image classification (ResNet).
- **Operating system:** Linux
- **Hardware:** No special hardware is required. CPU-based system. Recommended 8+ GB memory.
- **Program requirements:** Python 3.6
- **Python packages:** Catamount requires Python packages to run. First, it requires a virtual environment created with Virtualenv (https://virtualenv.pypa.io/en/latest/, commonly included with Python 3.6), or another virtual environment package. Second, Catamount depends on three other Python packages, *numpy*, *sympy*, and *tensorflow>=1.7*. See instructions below to install these dependencies.
- **Input:** Each Catamount test takes as input the Tensorflow model definition (compute graph) for the problem domain. Model descriptions are in Section 2.
- **Output:** Catamount tests output all analytics about the models (compute graphs) they load, including symbolic model parameters, algorithmic Flops, memory accesses, and memory footprint. By binding these symbolic functions to particular values, the tests also output the numerical values.
- **How much disk space required (approximately)?:** 1 GB
- **How much time is needed to set up experiments (approximately)?:** Less than 5 minutes
- **How much time is needed to complete experiments (approximately)?:** Less than 2 hours, depending on system CPU, memory performance
- **Publicly available?:** Yes
- **Artifact DOI:** https://doi.org/10.5281/zenodo.2259280
- **Repository location:** https://github.com/baidu-research/catamount
- **Code/data licenses (if publicly available)?:** Apache 2.0

A.3 Description

A.3.1 How delivered

Catamount is an open source Python package under Apache 2.0 license and is hosted with code and example DL compute graphs on GitHub (https://github.com/baidu-research/catamount).

A.3.2 Software dependencies

Catamount depends on recent versions of *numpy*, *sympy*, and *tensorflow>=1.7*, which work most stably with recent versions of Python. It is strongly recommended that users begin with Python 3.6 to install and run Catamount tests. Catamount may not work smoothly with prior versions of Python.

A.3.3 Data sets

The model definitions (compute graphs) analyzed in this paper are included as end-to-end tests of Catamount functionality and are distributed in the Catamount Github repository. These model definitions are in the form of Tensorflow checkpoints that were saved using the standard Tensorflow saver as follows:

```
# ... Construct a TF model ...
# Set output directory
```

```
outdir = ...
# Start TF session, create saver, and save model
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.Saver()
    saver.save(sess, os.path.join(outdir, 'tf_graph'))
```

This process saves the graph definition as a Tensorflow MetaGraphDef file, `tf_graph.meta`, along with saved parameters. Catamount can load the MetaGraphDef (`.meta`) files as graph-like Python objects for analysis. An example saving process can be found in the Catamount repo at `catamount/frameworks/example_graphs/tensorflow/rnn/tf_dyanmic_rnn.py`.

The `.meta` files used to analyze compute requirements for the different applications in this paper can be found in the following locations in the repo:

- Machine translation, word and character language models:
  `catamount/frameworks/example_graphs/tensorflow/full_models/language_models/`
- Image classification ResNet models:
  `catamount/frameworks/example_graphs/tensorflow/full_models/image_classification/`
- Speech recognition attention model:
  `catamount/frameworks/example_graphs/tensorflow/full_models/speech_attention/`

Finally, the tests that load these graphs and generate analytical models and numerical outputs can be found in the Catamount full-graph tests directory:

- Language models: `catamount/tests/full/tf_language_models.py`. Pass the parameter `--domain <domain>`, where `<domain>` can be one of `charlm`, `nmt`, or `wordlm`, for character, machine translation, or word models, respectively.
- Image classification: `catamount/tests/full/tf_image_resnet.py`. Pass the model depth as a parameter `--depth`. Supported depths currently include ResNet 18, 34, 50, 101, or 152.
- Speech recognition: `catamount/tests/full/tf_speech_attention.py`

A.4 Regenerating experiments from this paper

To download Catamount and regenerate results for this paper, run the following commands. These commands clone the public repository, check out the commit known to work for validating PPoPP paper results, and run the tests that generate results:

```
~$ git clone https://github.com/baidu-research/catamount
~$ cd catamount
~$ git checkout -b ppopp-artifact-validation ppopp-2019-artifact
~$ bash catamount/frameworks/example_graphs/tensorflow/full_models/generate_results.sh
```

A.5 Evaluation and expected result

With the commands in the prior subsection, Catamount should create an output file for each of the 9 analyzed compute graphs. These files will be named `ppopp_2019_tests/output_*.txt`. To regather results from these files after running the `generate_results.sh` script, you can run the following command inside the top-level Catamount directory:

```
~$ bash catamount/frameworks/example_graphs/tensorflow/full_models/gather_results.sh
```

A.6 Experiment customization

To customize Catamount tests or experimental results from this paper, users can modify the appropriate Python script in the tests directory, `catamount/tests/full/`. By changing the the `bind_subs` dictionary, users can bind symbolic dimensions of the compute graphs to different values. Users can also create their own compute graph definitions manually using the Catamount API (see `catamount/api/`) or by checkpointing Tensorflow models and loading them into Catamount as shown in the tests.

Catamount can calculate the algorithmic compute requirements for a loaded model. In particular, it calculates algorithmic FLOPs, memory bytes accessed, and minimal memory footprint for a pass through the compute graph. In addition to requirements presented in the paper, Catamount can calculate **algorithmic IO**, which is the amount of data accessed for input to and output from a model. Training data is often stored on and read from disk, and placed into the model's input memory allocations. Algorithmic IO is proportional to the batch size, but stays fixed as model size and training step compute requirements grow. We do not investigate algorithmic IO in this work, because we expect IO will grow very slowly relative to compute.