Computer Architecture for Deep Learning

Optimizing the Tradeoff Between Performance and Model Accuracy in Model Distributed Machine Learning Systems

Yu Chen
ychen244@syr.edu

College of Engineering & Computer Science, Syracuse University

March 1, 2019

Abstract

The era of Big Data, Internet of Things, and accessible cloud computing has allowed machine learning researchers to implement neural network algorithms and architectures that were once only theoretically possible. At first glance, the inherent abundance of linear algebra computations within deep learning architectures means it is well suited for commodity GPU and vector computational architectures. However, neural networks also bring along several key tradeoffs between hardware flexibility (CPUs) and customization (ASICs), between balancing hardware optimizations with algorithmic redesign, and between power consumption and model scalability. We provide a survey of industry-established best practices - at both the software algorithm and hardware layer - for optimizing the performance of deep learning networks, and propose, in the spirit of Amdahl's Law, that the bottleneck in terms of further significant progress now rests at the level of algorithmic innovation, rather than increasingly customized hardware design.

1 Neural Network Architecture

Neural networks, at their heart, consist of layers of parallel graphical nodes stacked together in a logical sequence. The edges in this computational graph are referred to as *weights*, and computation typically proceeds sequentially, from layer to layer. Each node in a neural network can be represented as

$$x_{n,i} = \psi(\sum_j w_{n,i,j} x_{n-1,j}) \tag{1}$$

[4] Here n represents the nth layer of the network, i represents the ith node of layer n, j represents the jth node of the previous $(n-1)$th layer, and ψ represents an activation or mapping function. Typically, researchers shorten this notation to vector form:

$$x_n = \psi(w_n x_{n-1}) \tag{2}$$

The allure of neural networks is in the relative simplicity of this feed-forward computation, paired with the **Universal Approximation Theorem**, which states that

> standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonconstant activation function are universal approximators with respect to $L_{p(\mu)}$ performance criteria, for arbitrary finite input environment measures μ, provided only that sufficiently many hidden units are available. [3]

This theorem has been warped into the fundamentally incorrect notion that *neural networks can learning anything* but the does indeed underscore that the vast potential of deep learning shifts away from algorithmic design to computational efficiency and hardware constraints - with an arbitrarily large number of hidden nodes and layers, neural networks are indeed capable of learning extremely intricate, high-dimensional patterns (such as those pervasive in natural language processing/generation and computer vision). As a result, the core promise of a neural network is encapsulated in this activation function ψ - which must typically be a nonlinear transformation for neural networks to correctly model inherently complex feature spaces. Indeed, consider a linear transformation of an arbitrary vector of data, x:

$$A^k = W^k X^k \tag{3}$$

In a neural network architecture, A^k becomes the input X^{k+1} for the next layer. Therefore, even if multi-

ple layers of transformations are stacked on top of each other, the overall effect is a linear transformation:

$$A_{k+1} = W^{k+1} W^k X^k \tag{4}$$

$$A_{k+1} = W' X^k \tag{5}$$

Reagan et. al. have noted two diametrically opposing trends in neural network architectures:

1.1 Deeper, Wider Hidden Layers

Wider, shallower networks tend to suffer from overfitting, where it learns to tightly fit its model parameters (weights) to the exposed training data but is unable to generalize on unseen test data.

1.2 Fewer Fully Connected Layers (Pruning)

`vgg`, the famous convolutional neural network first conceived at Oxford University by Simonyan and Zisserman that achieved 92.7% accuracy on a 1000-class ImageNet dataset, features only three fully connected layers that consume 7% of total runtime. In particular many deep learning models, at the algorithmic level, are taking advantage of a technique first introduced by Yann LeCun in 1989, and refined by Song Han and his team at Stanford University and NVIDIA at the Neural Information Processing Systems Conference (NIPS) 2015:

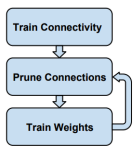


Figure 2: Three-Step Training Pipeline.

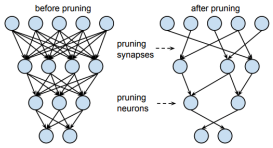


Figure 3: Synapses and neurons before and after pruning.

Figure 1: **Han and his team proposed an iterative pruning process, where certain weights are successfully removed, then the model retrained, and then more weights are removed.** This resampling and retraining technique allowed the Stanford and NVIDIA research team to achieve no drop in model accuracy while reducing the number of parameters by factors of **9X** to **13X** (on VGG, AlexNet, and LeNet). [2]

Neural network weights that are close to 0 are pruned, with the effect of converting dense weight matrices into sparse matrices. Aside from compressing model parameters for hardware acceleration, pruning also conveniently mitigates the effect of **vanishing gradients**, a well-known problem with deep neural networks featuring significant numbers of hidden layers.

The bulk of these operations are a strong match for hardware acceleration and parallelization techniques:

- **For loops in both the training and inference steps are statically defined**. For instance, the number of data points N, the feature dimensionality K, the number of hidden layers, the width of each layer, etc. can all be known at compile time, virtually eliminating most common control hazards.

- If batch sizes are optimally selected and data is stored contiguously in memory, then **hit rates across the cache hierarchies are maximized** - the lack of branching and conditionals within the lower-level logic of a deep learning network means that memory accesses can be parallelized and pipelined, edge the average cycles per instruction towards the pipelined workflow's ideal CPI.

Despite the strong matches between neural network workflows and potential for hardware acceleration, researchers have encountered barriers inhibiting significant incremental progress.

1.3 Challenge 1: The Power Wall

As models have been increasingly complex and deeper, with larger weight matrices and higher dimensional feature spaces, they require more memory references, which in turn, requires more energy. Google's famous AlphaGo neural network, which made news headlines for being the first computer program to defeat a professional Go player without handicaps in 2015, features an astonishingly high electricity bill of $3000 per game. [1]

While AlphaGo and other eponymous neural network programs' achievements are impressive, this power consumption demand prevents deployment on either a embedded or cloud distributed systems level (since as the model would quickly drain stored battery power) and would increase the maintainance costs, downtime, and carbon footprint of the large cloud providers' datacenters. Indeed, particularly for lower-intensity (with intensity defined as the number of floating point operations per weight parameter), the bottleneck is memory access both in terms of execution time and in power consumption:

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

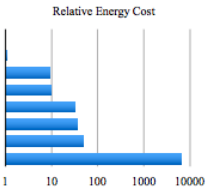


Figure 2: **Each memory access to DRAM is thousands of times more costly than a 32-bit `ADD` operation.** This further underscores need to reduce the number of parameters in neural networks, as being able to fit model parameters in the SRAM cache reduces power consumption and access times by a factor of over 100X. [2]

Indeed, deep learning hardware acceleration efforts are encountering another version of **Dennard's Scaling**: we have traditionally optimized computation as a whole by packing more and more transistors / gates into a smaller surface area, but are facing an asymptotic barrier where power scales quadratically with a linear increase in transistors/gates. During the initial golden age of deep learning, we have optimized model performance by increasing the density of weight matrices (either by making hidden layers wider, or by increasing the number of hidden layers) relative to the size of our features. Mark Horowitz has noted ASIC architectures already have impressive data locality, and are already designed to use low-energy integer data:

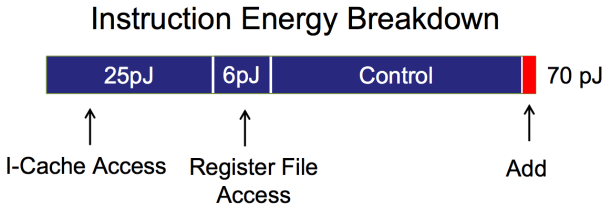


Figure 3: **Energy breakdown of an average ASIC `ADD` instruction.** Notice the extremely small portion of energy attributed to the actual computation versus control and cache index addressing, even within an ASIC. [8]

1.4 Challenge 2: The Simplicity Wall

Since a large portion of neural network operations are either matrix multiplication or convolution, **the easiest optimizations have long since been implemented**. By the very nature of their inherent simplicity, neural network performance acceleration has hit what is termed the **Simplicity Wall**, where the lowest risk optimizations - termed **safe optimizations**, have already been applied. A safe optimization is defined as an optimization that improves hardware performance but does not reduce the model's performance metrics (for classification, its accuracy, F1, or AUROC scores, for regression - mean absolute error, mean squared error, and mean absolute percentage error). The boundary between safe and unsafe optimizations, however, is not as clear as researchers would hope. Indeed, neural network models, by their very nature, tend to be stochastic:

- **the ordering of data points**: if data points are not fed into the model sampled randomly, then the model will overcorrect initially for certain biased trends and require much longer to complete training

- **randomness of weight initialization**: two of the most common modern techniques for weight matrix initializations are **Xavier initialization** and **He initialization**. Both initialization types draw from Gaussian distributions with parameters defined by the activation function and sample size:

$$w_i \sim N(0, \sqrt{\frac{2}{D_{i-1}}}) \tag{6}$$

- **optimization algorithm selected**: it is often intractable to compute the loss function for every single data point; therefore, many neural networks use a variant of gradient descent called **stochastic gradient descient** which computes loss for only a small subset of training samples each iteration

Moreover, model performance is not a single scalar value, but rather a time-series vector of error metrics at each iteration of the model. Thus, researchers have devised a method of **bounding** to classify optimizations as either safe or unsafe. The underlying assumption behind this performance methodology is the presence of an **Iso-Training Noise**[1], which measures the latent noise in prediction errors that are part of neural networks.

[1] coined by Reagen et. al. [4]

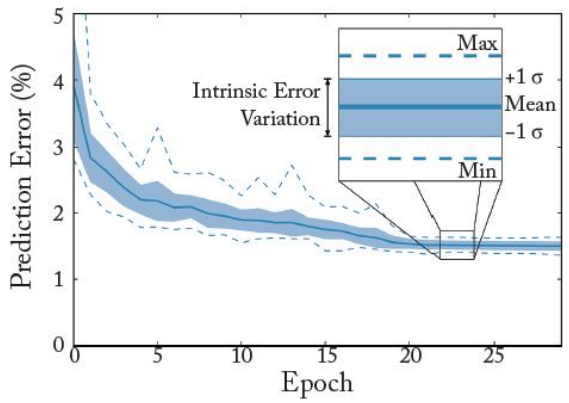


Figure 4: **Researchers will iteratively train models using random initialization to derive an intrinsic error variation**. After arriving at some threshold for acceptance (ie., +/- σ or 2σ) from the mean, we can classify any optimizations that return prediction errors outside of these bounds for a given iteration as unsafe. [4]

By repeatedly initializing the model parameters at different states, a distribution for inherent noise can be constructed. If a hardware optimization technique's prediction error stays within an acceptable interval (as defined by the distribution's variance - usually one or two standard deviations from the mean), then it is assumed that **any observable effect in prediction error from a hardware optimization cannot be distinguished from the neural network's inherent training noise**.

A convenient way of comparison profiling performance across architectures across a high number of dimensions (such as operation type, hardware architecture, data type bit sizes, etc.) is with a well-established machine learning metric called **cosine distance (ϕ)**:

$$\phi(A, B) = 1 - \frac{A \cdot B}{|A||B|} \tag{7}$$

In this case, consider two neural network architectures A and B: Architecture A spends 55% runtime on computation, 15% on data loads, 30% on data transfer and manipulation, while Architecture B spends 65% on computation, 10% on data loads, and 25% on transfers:

$$A = \begin{bmatrix} 0.55 & 0.15 & 0.30 \end{bmatrix} \tag{8}$$

$$B = \begin{bmatrix} 0.65 & 0.10 & 0.25 \end{bmatrix} \tag{9}$$

$$\phi(A, B) = 1 - \frac{0.4475}{0.6442 \times 0.70356} \tag{10}$$

$$\phi(A, B) = 0.012 \tag{11}$$

This indicates a high degree of similarity between the two networks in terms of its distribution of runtime operations, and conveniently can be used to cluster different architectures that may benefit from ASIC or FPGA optimized hardware.

1.5 Backpropagation and Model Learning

A neural network actually learns by iteratively making predictions about a value, comparing that predicted value to the true value, and propagating back across its network changes to the weights to account for this difference. For many supervised learning classification problems (such as identifying a character in the English alphabet, recognizing a person's voice, or detecting a fraudulent financial transaction), the *cross-entropy loss* function is used to model the difference between true and predicted outputs:

$$C(y_{true}, y_{pred}) = -\frac{1}{n} \sum_i ln(\frac{e^{y_i^{true}}}{\sum_j e^{y_j^{pred}}}) \tag{12}$$

If we considered a neural network with only one neuron, then the cost function could be expressed as

$$C = A(X \times W + B, y_{true}) \tag{13}$$

Here, A is the activation function, X is the input feature matrix, W is the weight matrix (the neural network's parameters). and B is the bias vector.[2] Since we are looking to update our weight parameters, we care about the derivative of C with respect to W:

$$C'(W) = C'(R(X \times W + B)) \tag{14}$$

We'll use $Z = X \times W + B$ for shorthand. We can apply the **chain rule of derivatives** to calculate the ultimately derivative of cost:

$$C'(W) = C'(R) \times R'(Z) \times Z'(W) \tag{15}$$

Intuitively, this states the the degree to which we can alter our cost function by adjusting weights (which is the whole purpose of supervised machine learning as a whole) is determined by, and more specifically, a product of three items:

- the degree to which we can alter our cost function by adjusting the output we get our from our activation function
- the degree to which we can alter the output of our activation function by adjusting the output of $X \times W + B$
- the degree to which we can alter the output of $X \times W + B$ by adjusting the values of W

[2]The mathematical notations used in the following section is taken from The ML Cheatsheet: Backpropagation.

What is noteworthy here is that the entire "learning" operation of a neural network (the updates to weights based upon feedback from the training target outputs y_{true}) can be broken down into modular components: $R'(Z)$, can be easily substituted from ReLU, to tanh, to sigmoid activation functions. The weighted output $Z'(W)$ can likewise be altered to include custom regularization parameters, so long as the formula itself is differentiable, with minimal effect on the rest of the learning pipeline.

1.6 Memoization to Optimize Backpropagation

There is clearly a fair amount of recursive work involved in this calculation which can be optimized at the algorithmic level. For instance, assume we have 3 hidden layers (and therefore 4 weight matrices) in our neural network architecture. The derivatives of the cost functions with respect to each layer's weights are recursively computed, and optimized:

$$C'(W_3) = (O - y) \cdot R'(Z_3) \cdot H_2$$
$$C'(W_2) = (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot H_1$$
$$C'(W_1) = (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot H_0$$
$$C'(W_0) = (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot W_1 \cdot R'(Z_0) \cdot X$$

Figure 5: **Derivatives of cost functions with respect to each layer's weights.** Notice the familiar pattern of computation at each layer. The use of dynamic programming techniques such as memoization can convert a task that is quadratic by nature (if there are N layers, you must compute $\frac{N^2}{2} \rightarrow O(N^2)$) derivatives to one that is linear (given N layers, compute one new derivative for each layer. [11]

Indeed, Google has found that in the computationally inefficient method of learning for a recurrent neural network (RNN), in order to access the state of the model at time t, each time period's state must be recomputed through forward-propagation beginning from time 0 up until time t, ultimately resulting in $O(1)$ memory complexity but $O(t^2)$ time complexity as a result of $\frac{t \times (t+1)}{2}$ forward passes. [5] Instead of this inefficiency, Google's DeepMind researchers have proposed an adapted implementation of Chen's recursive segmentation, by storing hidden RNN states across time points. This allows backpropagation across time to follow immediately after a forward propagation calculation.

1.7 Training and Inference

Supervised machine learning models incorporate two major workflows: **training** and **inference**. During the training portion, the model is fed in both the features usually in the form of a high-dimensional tensor X as well as the targets (usually $N \times K$ vector, where N is the number of training data points, and K is the number of prediction classes - for a regression model, it is simply a $N \times 1$ vector of real number values). The model iteratively updates its parameters (for a deep learning neural network, these are the weight matrices and bias vectors) as it computes the error between its predicted value y_{pred} and the actual observed result y_{true}.

On the other hand, during the inference stage of a machine learning model, parameters are fixed, and data flows forward only. Thus, inference execution time is almost always faster than training time, although the training to inference execution time ratio varies from model to model and amongst CPU and GPU architectures. Indeed, certain neural network architectures tend to be especially well suited for GPU computations:

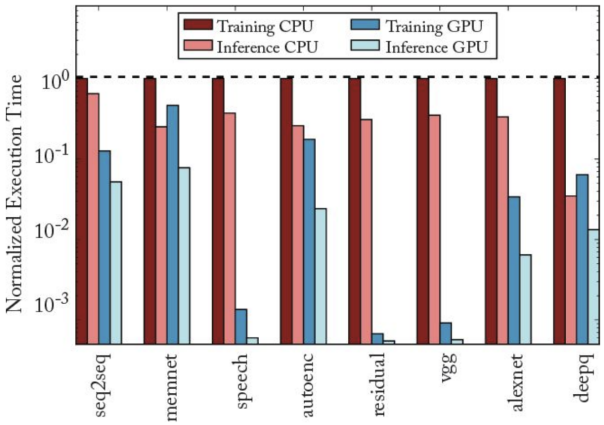


Figure 6: **The runtimes of inference and training workflows on popular neural network architectures.** Notice the dramatic speedup gained from GPU compute on `speech`, `vgg`, and `residual`. Given the lower memory capacity and overall slower clock speeds of GPUs compared to CPUs, the architectures that require storing massive long-term memory units, such as `memnet` and `seq2seq`, unsurprisingly experience the smallest speedup benefit from CPU. Moreover, architectures that feature significant sequential computations (where no data parallelization is available) like `autoenc`, which contains an extremely small "chokepoint" hidden layer, also appear to perform suboptimally for GPUs. [4]

2 Case Study: Hardware Customization and Model Compression for Google's TPU

Traditional neural network architectures utilize floating point data types for weights. In Tensorflow, for instance, weight matrices and bias vectors are frequently initialized by sampling from a random Gaussian distribution, which, by default utilizes `float32` data types [7]. A common approach towards optimizing model performance is to reduce the size of the data types involved in the fitting and prediction steps of a neural network. A few guidelines are followed:

- **Can the data fit into on-chip memory?** Access times for off-chip DRAM is significantly lower; therefore, if architects can reduce data type sizes for neural network matrices to fit entirely onto on-chip memory, the simple reduction in cache miss rates and/or page faults may be well worth the optimization itself.
- **What is the expected reduction in power consumption?** Studies have found that 8-bit fixed point multiplication yields a 13x reduction in power and 38x reduction in area required compared to a traditional 32-bit floating point.[6]

Google has recently deployed a custom ASIC (application-specific integrated circuit) called **TPU (Tensor Processing Unit)** for accelerating deep learning inference for its data centers. The TPU architecture centers around a 64,000 (256 x 256) 8-bit MAC (Multiply-accumulate operation) units, fed by an on-chip SRAM memory cache. Each MAC operation modifiers an accumulator register `a`:

$$a \leftarrow a + (b * c) \tag{16}$$

The TPU utilizes a CISC-style architecture, with an average of 10-20 cycles per instruction. Common instructions are

- `Read_Host_Memory`: read data from the CPU host memory into the Unified Buffer
- `Read_Weights`: read weights from the Weight Memory.
- `Write_Host_Memory`: write data from the Unified Buffer back out to the CPU host memory.
- `Matrix_Multiply/Convolve`: perform a matrix multiply using a N x 256 input vector and the 256 x 256 constant weight vector (supplied from `Read_Weights`). This matrix multiply computation is pipelined, and takes N clock cycles to complete in steady state.
- `Activate`: perform a fixed nonlinear transformation (typically either a Rectified Linear Unit, Sigmoid function, etc.).

The `Matrix_Multiply` instruction itself is composed of 12 bytes.

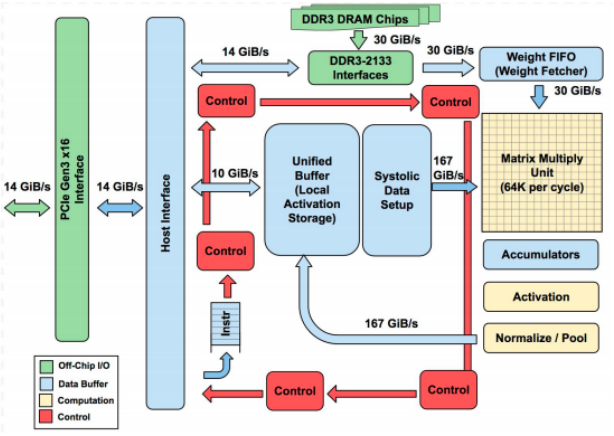


Figure 7: TPU Architecture. The **64K Matrix Multiply Unit** handles the computational work. The **Systolic Data Setup** feeds in data used for training the model, while the **Weight FIFO Fetcher** supplies the relevant weight matrices for the computation. [13]

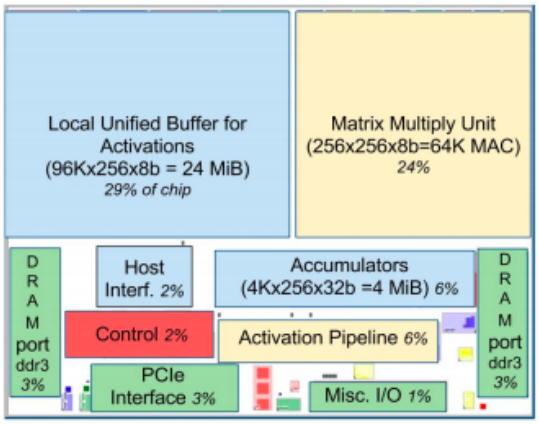


Figure 8: **TPU Die Floorplan**. Notice that the percentage of the die dedicated to Control is 2%- significantly less than in a CPU or GPU. Buffer space (37%) and computation (30% from the Matrix Multiply Unit and the Activation Pipeline) [13]

2.1 Compression and Quantization

A 32-bit floating point multiply operation requires about 3X as much energy (3.7 pJ versus 1.1 pJ) and 4-5X area accessed (7,700 μm^2 vs. 1,640 μm^2). [8] Thus, a reduction to an 8-bit integer representation of weight parameters leads to extremely significant reductions in energy and size consumptions. Given that the power consumption of high precision floating points is exorbitant, and the allure of fitting model parameters within L1, L2, and L3 caches, the TPU makes use of several significant compressions to reduce the overall size of its model. This effectively presents the hardware version of machine learning model regularization, whereby an algorithm designer purposely limits (or throttles) the available size of model parameters to prevent the model from overfitting to training data. By forcing weight parameters to lower levels of precision, machine learning models incur far less memory footprint while often building in intrinsic safeguards for model overfitting. The process of transforming floating-point numbers into smaller integers is called **quantization**.

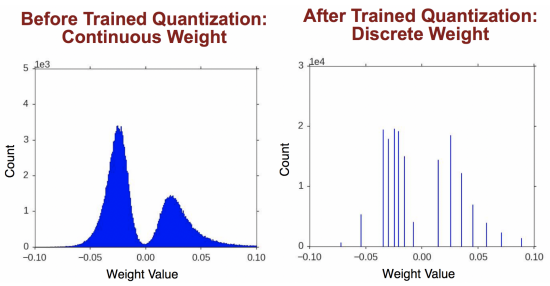


Figure 9: **Histogram showing distributions of weight parameters before and after trained quantization.**. Notice the amount of reduction in mass from the distribution - which reflects at runtime in significantly reduced memory and power footprint, and ultimately gains in model execution time. A researcher will iteratively quantize weight parameters, retrain the model, and quantize parameter further to retain model accuracy even as data type precision is incrementally decreased. [1]

Further compression can be achieved by applying **Huffman Encoding** to the weight parameters, selecting higher frequency weights to be represented with less bits, and less frequent weights with more bits. With a combination of pruning (completely zero-ing out certain weights), trained quantization, and Huffman Encoding, Han and Dally have achieved up to 49X reduction in model size without sacrificing accuracy:

Network	Original Size	Compressed Size	Compression Ratio	Original Accuracy	Compressed Accuracy
LeNet-300	1070KB	27KB	**40x**	98.36%	98.42%
LeNet-5	1720KB	44KB	**39x**	99.20%	99.26%
AlexNet	240MB	6.9MB	**35x**	80.27%	80.30%
VGGNet	550MB	11.3MB	**49x**	88.68%	89.09%
GoogleNet	28MB	2.8MB	**10x**	88.90%	88.92%
ResNet-18	44.6MB	4.0MB	**11x**	89.24%	89.28%

Figure 10: **Final compression ratios and model accuracies from prototypical modern neural network architectures**. Note almost all of the compressed model sizes are capable of fitting entirely on a typically-sized 1MB L2 cache or an 8MB L3 cache. [1]

3 Data Hazards

Neural networks are subdivided into semi-dependent layers, each loosely connected to the layer before and after it via weight matrices. Recall that the output of a given layer is

$$h_k^l = \sigma(\sum_j w_{jk}^l h_j^{l-1} + b_k^l) \tag{17}$$

Where σ is any valid activation function (ReLU, sigmoid, tanh), h_k^l represents the output of the k th neuron of layer l, and w_{jk}^l is the weight that connects neuron j of layer $l-1$ to neuron k of layer l. Clearly, any matrix multiply operation on h_k^l must wait until the activation function computation has completed. When this dependency is encountered during operation, the TPU's matrix multiply unit must stall before it reads the next set of operands from the Unified Buffer.

4 Amdahl's Law for Deep Learning

Amdahl's Law is a cornerstone for computer architecture because it provides architects and researchers with feedback on which optimizations will yield the greatest overall speedups in performance. Thus, it is a convenient measuring stick of a computer scientist's **return on investment**. Amdahl's Law can be adapted within the context of deep learning to identify whether hardware architects and machine learning practitioners are allocating resources and efforts towards the areas of highest potential speedup. For instance, Gaurav Kaul of Intel presented his own redefinition of Amdahl's Law at the High Performance

Computing Conference 2017 in Switzerland [12]:

$$Speedup = (\frac{1}{Ser_{\%} + \frac{1-Ser\%}{|C|}}) \times (\frac{1}{Sca_{\%} + \frac{1-Sca\%}{|V|}}) \quad (18)$$

Here, $|C|$ represents the number of cores available, $|V|$ the vector length, $Ser_{\%}$ the percentage of the workflow that is serial (sequential) and therefore not parallelizable, $andSca_{\%}$ represents the percentage that is scalar computation (and unable to take advantage of vector compute architectures). Thus, the ideal deep learning workflow is one that minimizes both $Sca_{\%}$ and $Ser_{\%}$. If both of these are 0, then the speedup becomes

$$Speedup = |C| \times |V| \quad (19)$$

What is significant from this formulation of speedup is that with the exception of $|C|$ (the number of cores), all other variables to optimize are found at the algorithmic layer. It is becoming increasingly clear that optimizations from hardware are only a minor piece of the puzzle. Far more substantial speedups can be acquired by machine learning engineers who design their neural network architectures wisely, selecting optimal hyperparameters, identifying the optimal vector batch lengths, removing "chokepoint" operations from their algorithms (areas of their hidden layer architecture where computations must proceed sequentially), etc. Note that **only in the ideal case of zero serial code and completely vectorized computation** does hardware (in the form of the number of cores $|C|$, play as large a role as the size of the vectors themselves $|V|$, an algorithmic layer parameter.

4.1 Unwarranted Focus on Convolutional Neural Networks

Moreover, the spirit of Amdahl's Law can be applied to the industry as a whole to reveal stark mismatches between resource allocation and tangible business needs and problems. Despite the flourishing ecosystem of neural network architectures prevalent today, most business use cases revolve around three primary types:

- Multi-Layer Perceptrons
- Convolutional Neural Networks
- Recurrent Neural Networks

Indeed, Google reports that approximately 95% of its deep learning workflows encompass one of these three architectures.[6] Operational profiling conducted on the major deep learning architectures reveals that most network architectures spend the majority of their compute time on matrix multiplication or convolution operations.

Name	*LOC*	*Layers*					*Nonlinear function*	*Weights*	*TPU Ops / Weight Byte*	*TPU Batch Size*	*% of Deployed TPUs in July 2016*
		FC	*Conv*	*Vector*	*Pool*	*Total*					
MLP0	100	5				5	ReLU	20M	200	200	61%
MLP1	1000	4				4	ReLU	5M	168	168	
LSTM0	1000	24		34		58	sigmoid, tanh	52M	64	64	29%
LSTM1	1500	37		19		56	sigmoid, tanh	34M	96	96	
CNN0	1000		16			16	ReLU	8M	2888	8	5%
CNN1	1000	4	72		13	89	ReLU	100M	1750	32	

Figure 11: **Overview of main neural network architectures deployed by Google**. MLPs (Multi-Layer Perceptrons) and LSTMs (Long Short-Term Memory) occupy approximately 95% of all deep learning workloads. Note also that the computational intensity (as measured here by TPU operations per weight byte, is significantly higher for CNNs (convolutional neural networks). [13]

The author's of Google's TPU paper has noted that while 15% of all papers at ISCA 2016 were on the topic of hardware acceleration for neural networks, all nine papers focused upon CNNs (convolutional neural networks). CNNs are notoriously complex and form the basis of modern image recognition and computer vision models, but represent only about 5% of typical datacenter business demands. Therefore, there is a misalignment in terms of optimizing for the common case, at least at the level of research focus and efforts.

4.2 Over-Parameterization

Researchers have noticed for years that many of the weight parameters in a neural network are more or less redundant. For instance, many image recognition and computer vision models feature a first layer of weights that are globally smoothed representations of edges (usually defined as areas of high contrast - these edges provide the outlined shape of objects in the image frame). Denil et. al. have pointed out that since each pixel is likely to simply be a weighted average of its surrounding pixels, it is not necessary to retain all weights to all hidden nodes, as removing up to 95% of weights using low-rank approximation algorithms has been shown to not affect a model's predictive power. [10]

5 Recommendations for Future Work

At the end of the day, resources within the field of deep learning must be re-allocated away from hardware optimizations and towards the redesign of our machine learning algorithms. A survey of the state of hardware and software optimizations for deep learning reveals three fundamental observations:

1. From specialized ASIC architectures with super-sized memory bandwidths and massive computing surfaces like Google's TPU and other ASICS, the **domain of available hardware accelerations for deep learning has been thoroughly explored and implemented, perhaps arguably over-engineered**.

2. **It is far easier to scale performance improvements at the algorithmic, as opposed to the hardware, layer.** Dynamic programming, quantization, hashing and Huffman encoding - all of which are more or less prevalent in more established domains of computer science - have barely begun to be implemented within neural network architectures.

3. There are many tradeoffs involved in designing an optimal deep learning computational architecture, but **model complexity and model performance is not one of them**: we have cited several cases throughout where reductions in the sheer size of a neural network's parameters often corresponds with improvements in model accuracy by improving runtime execution speed (so that more training can occur in a reasonable window of time) and reducing the danger of model overfitting.

The building blocks for improving the state of machine learning algorithms is there - the most prevalent deep learning frameworks for machine learning engineers and data scientists available today share core patterns that enable modular reuse and multiple layers of abstraction to reduce the operational complexity of architecting a complex computational system:

- **Data and operation primitives**: Because of the sheer complexity in building a deep learning model from scratch, each framework provides a primitive data block or operation so that developers can reason about the logic of their neural network architectures at a much higher level. In Tensorflow, the `Operation` class instance represents a node in the computational graph that accepts and produces a variable number of `Tensor` instances. For instance, at a fundamental level, almost all computation within Tensorflow can be parsed down to `c = tf.operation(a,b)`, where `a` and `b` are `Tensor` instances and `tf.operation` is a `Operation` instance.

- **Declarative programming styles**: TensorFlow, Theano, Caffe all provide a declarative API for developers, with the framework itself constructing dependency graphs and optimizing the computations at a much lower level.

Indeed, arguably the entire design feedback structure of deep learning is inverted. Hardware acceleration has traditionally been viewed by data scientists or researchers as a black box constant, governed by advances in chip technology and Moore's Law. However, deep learning algorithms today are saturated with parameters (weights), leading to over-optimized hardware that does not reflect business and industry use cases. No longer can data scientists and software engineers simply rely on performance improvements from Moore's Law to solve all issues with computational performance. Deep learning researchers today must take advantage of all possible optimizations at the algorithmic level (compression, quantization, Huffman encoding, memoization, etc.) before passing the burden of responsibility down to the hardware level.

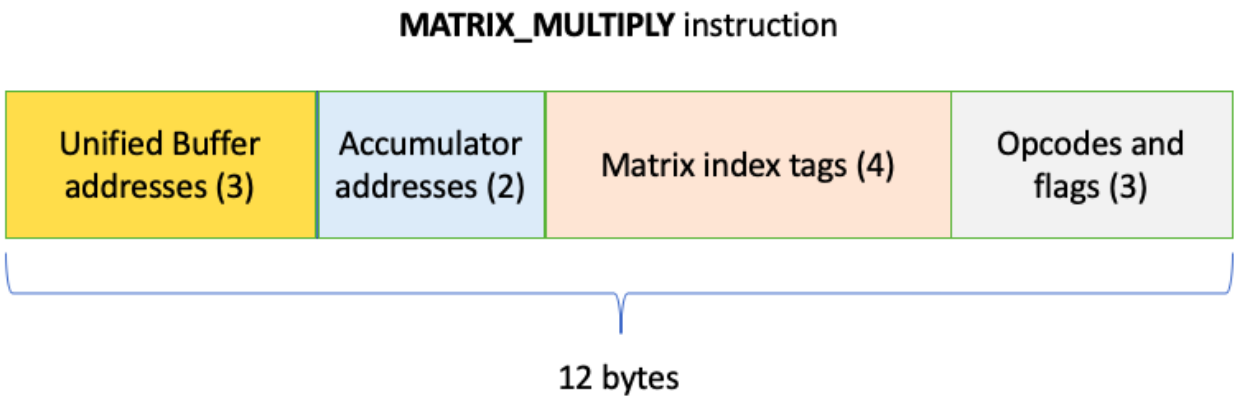
List of Figures

Appendix A Weight Initialization Example in Tensorflow

```
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'out': tf.Variable(tf.random_normal([n_hidden_1, n_classes]))
}

biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Appendix B Google TPU Instruction Bit Distribution



References

[1] Song Han. Lecture 15 - Efficient Methods and Hardware for Deep Learning. May 25th, 2017. Link. Accessed February 28th, 2019.

[2] Song Han, Jeff Pool, John Tran, and William Dally. *Learning both Weights and Connections for Efficient Neural Networks*. Presented for NIPS 2015. Link. Accessed February 28th, 2019.

[3] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*. Volume 4, Issue 2, 1991.Pages 251-257.

[4] Brandon Reagen et. al. Deep Learning for Computer Architects. *Synthesis Lectures on Computer Architecture*. Accessed February 26th, 2019.

[5] Andrunas Gruslys et. al.*Memory-Efficient Backpropagation Through Time*. Google DeepMind Research Group, presented for NIPS 2016. Submitted June 10th, 2016. Link. Accessed March 1st, 2019.

[6] Norman Jouppi et. al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, June 26th, 2017*. Accessed February 26th, 2019.

[7] Venelin Valkov. Building a Simple Neural Network - Tensorflow for Hackers (Part II). *Spark as the Gateway Drug to Typed Functional Programming*. Accessed February 26th, 2019.

[8] Mark Horowitz. *Computing's Energy Problem (and what we can do about it)*. 2014 International Solid-State Circuits Conference. Link. Accessed February 28th, 2019.

[9] Tianqi Chen, Bing Xu, Zhiyuan Zhang, and Carlos Guestrin. *Training deep nets with sublinear memory cost.* Link.

[10] Mishal Denil et. al. Predicting Parameters in Deep Learning. June 3rd, 2013. Link. Accessed February 28th, 2019.

[11] Backpropagation. *The ML Cheatsheet.* Link. Accessed March 1st, 2019.

[12] Gaurav Kaul. High Performance Hardware for Distributed Deep Learning. Switzerland HPC Conference 2017. Link. Accessed March 1st, 2019.

[13] David Patterson et. al. In-Datacenter Performance Analysis of a Tensor Processing Unit. Presented at the 44th International Symposium on Computer Architecture, Toronto, Canada. June 26th, 2017. Link. Accessed March 1st, 2019.