Hardware Systems for Neural Acceleration

Shelby Thomas
Computer Science and Engineering
University of California, San Diego
San Diego, CA 92122
sht005@eng.ucsd.edu

Abstract

For years the development of new neural networks and better training algorithms has been largely orthogonal to what kind of hardware might drive their usage. While neural networks have been around since the 1980s they were considered far too computationally intense and impractical compared to canonical computer vision techniques. In 2012 a neural net dubbed AlexNet handily broke the world record for image recognition tasks using a deep convolution neural network trained on commodity GPUs. This showed for the first time that deep convolutional neural networks can be trained on a low cost system in a reasonable time, one week, while producing state-of-the-art accuracy. Since then the exponential growth of even deeper neural network research has been enabled through high performance CPU/GPU devices and an awareness of the underlying hardware. This has opened a door for application specific integrated circuits, neural datacenters, and highly optimized compute kernels for putting them all together. This has ushered in a new era of hardware research that requires an appreciation of the complex neural learning problem coupled with hardware design experience to improve throughput, power, and in some cases accuracy in neural algorithms. This survey presents an overview of the hardware and software systems designed for neural networks with a focus in convolutional neural networks. A new taxonomy for research in neural acceleration is also presented that represent the three angels most commonly taken to optimize this algorithm - technology enabled acceleration (TEA), system enabled acceleration (SEA) and algorithm enabled acceleration (AEA).

1 Introduction

A neural network is a machine learning technique to generate a generalizable model for a certain problem domain. Broadly, machine learning techniques fall into two categories: Supervised learning and unsupervised learning. Supervised learning involves learning from training examples and target outputs while unsupervised learning involves extracting features from data without positive and negative examples. At a high level the goal of any machine learning algorithm is to generate a function that can model a certain phenomenon. This phenomenon can be the relationship between house features and house price, or pixels and output image. The model itself is completely dependent on training data but needs to generalize enough to perform well on data that it has never seen before. Imaging machine learning as a function generator the goal of training is to create a function with the correct set of constants or weights that minimize the error on the output. For most machine learning techniques this involves solving a convex optimization problem and finding a global minimum. In contrast, neural networks generate and model non-linear functions and require solve a significantly more involved non-convex optimization problem with several local minima and no guarantee that the global minima will be achieved. While there are many variants of neural networks, they are built on top of a set of common characteristics and techniques. In a basic architecture, a neural network will contain one input and output layer with a variable number of hidden layers. This basic architecture
can be seen in Figure 1 - the node at the ends of this network are the input and output and the layers in the middle are the hidden layers. The inputs of the network are propagated as neurons through the network and the connections in between each prorogation are the weights of the network being optimized.

2 Neural Network Overview

A typical net consists of a number of layers $l...L$ defined by a vector of activation units (neurons) and matrix of edges weights (synapses) between neurons. In this section we will discuss the most basic type of neural network, a multi-layer perceptron (MLP) shown in Figure 1 (left). In this kind of network all of the nodes are fully connected meaning that every neuron in a layer has an edge to every other neuron in the next layer. Note that in this fully connected architecture the number of edges between two layers is the product of the neurons in each layer, i.e. four neurons in layer one and five neurons in layer two will have 20 edge weights.

2.1 Computational Rules

We can distinguish a unique neuron $a_{lj}$ by index $j$ and the layer $l$ it is associated with i.e. neuron $a_{23}$ is the third neuron in layer two. Generally it is helpful to think of a group of neurons in one layer in terms of vectors in which case we omit the subscript and keep the layer superscript i.e. $a^2$ is the vector of activations in layer two.

Unique edges are defined similarly as $W_{lj}^1$ where the index of an edge denotes a "to-from" relationship - to the $i$ neuron in layer $l + 1$ from the $j$ neuron in layer $l$, an example of this is shown in Figure 1 (right). Similar to the vector of neurons it is useful to represent all the edges between neurons as a matrix of weights denoted with a layer superscript i.e. $W^2$ is the weight matrix of the second layer that contains edges from layer two to layer one.
The fundamental unit of computation in a neural network is a dense vector-matrix between the input activations vector $a^l$ and the weight matrix $W^{l+1}$. The vector-matrix multiply is a set of multiply-accumulate operations that provide a new vector $z^{l+1}$, also known as the weighted input vector. The weighted inputs are passed through an activation function $g$ which introduces a non-linearity into network and produces the output vector $a^{l+1}$. In the 3-layer MLP case this equation is defined as:

$$f(a^0; W) = W^3(g(W^2 g(W^1 a^0)))$$

(1)

3  The Learning Problem

Note that the goal of our learning algorithm is to generate a function that models our data. This is achieved by tuning the weights $W$ of our network such that we reduce error between the collected results or ground truth and the results that the network outputs. We quantify how well a network performs by using a loss or objective function $\mathcal{L}$ and tuning our weights $W$ such that error is reduced. We can pose this as the following optimization problem:

$$\min_W \mathcal{L}(f(a^0; W), y)$$

(2)

Once the weights have converged, the network has completed training and the weights will not change again unless retrained. This means that weights can be trained on an external machine and then exported to an embedded device that uses the same network architecture to perform inference. Training time takes significantly longer than inference as solving the non-convex optimization problem in Equation 2 requires several tens of thousands of iterations of an optimization technique such as stochastic gradient descent before the network converges. In contrast, making a prediction only requires us to solve Equation 1 using basic vector-matrix multiplies and take an order of magnitude less memory. Further explanation into the difference in computation and memory usage of inference and training is the topic of another section.

Formulating the Training Algorithm

Neural network training has three steps, the forward pass, backward pass, and weight updates. The forward evaluation runs through the network to the output and evaluates the error in the output layer relative to what the true output of the data should be. For clarity the layers are explicitly stated next to each weight $w$ and activation $a$. The activation for a single neuron is the following:

$$a^l_i = g\left(\sum_{j=1}^{k} W^l_{ij} \times a^{l-1}_j\right)$$

(3)

The above equation states that the activation of a certain neuron $a_i$ in layer $l$, is the product of every weight to neuron $i$ from neuron $j$ in layer $l-1$ with the activation of the previous layer $l-1$ of neuron $a_j$. This process is repeated for every neuron in $l-1$ layer until neuron $k$ and summed. The activation function $g(x)$ is then applied to the sum. Note that we have omitted an extra factor $b_i$ called the bias neuron that allows a shift of the activation function. Computationally, it is one extra neuron in each layer added to the total and for clarity it is omitted.

Error terms $\delta$ are computed for each neuron $i$ in the output layer $l_o$. Where $t(x)$ indicates the true label of the data and $g'(x)$ is the derivative of $G(x)$:

$$\delta^l_o = (t^l_o - a^l_o) \times g'(a^l_o)$$

(4)

To measure how much error each layer contributes to the total an error matrix is created that has the same dimensions as the weights between two layers. This repeated process of creating an identical error matrix is a recursive process that provides a measurement of how each each weight effects the output. This error matrix essentially doubles the memory requirement of network as we now need to
store memory for the weights and the errors. The equation to backpropagate per layer is shown below:

$$\delta^l_i = \left( \sum_{j=1}^{m} \delta^{l+1}_j \times W^l_{ji} \right) \times g'(a^l_i)$$  \hspace{1cm} (5)

After all the error for each layer are stored, a small change to the weight matrices are made based on a pre-defined learning rate $\alpha$. In the equation below note that we are performing almost the same calculation we do in the forward pass except our weight matrix is now the error matrix on which the learning rate is applied. This delta is then applied to all the weights int he network. For every layer from 1...k the change in weight is the following:

$$\Delta w_{ij}(l-1, l) = \alpha \times \delta(l) \times a_j(l-1)$$  \hspace{1cm} (6)

This final process is also known as gradient descent (GD) and is the optimization algorithm that we apply to the network. In practice an algorithm called stochastic gradient descent (SGD) with mini-batching is used. In GD one update is made after running through the entire dataset. For large datasets like ImageNet with millions of data points performing one update per million samples makes training impossible. SGD takes the opposite route where parameter updates are made after only one sample. The problem with this method is that since every example has an equal effect on the network, outliers and noisy points have the potential to swing the network away from the optima. Using SGD with mini-batches of example is a good middle ground between the two techniques and ensures that any noisy points are averaged out. For computation, this means that our memory size increases significantly as now we need to store a the gradient+error matrices for each example in the mini-batch.

Figure 3: The LeNet-5 architecture

4 Convolutional Neural Networks

Convolutional neural networks (CNN) have shown state-of-the-art performance in a variety of image, speech, and video recognition tasks beating long standing canonical methods of computer vision and language processing. Unlike the densely connected multi-layer perceptrons, the weights in a CNN are locally connected filters and the activations are output images. While this local connectivity means that the total weights in each layer are lesser the memory usage in each layer is greater. Figure 3 shows an example of a CNN. In this image the weights/filters are omitted and just the activation feature maps are shown. After every convolution process the image size is slightly reduced and in some layers an explicitly sub-sampling is performed to aggregate multiple low-level features.

4.1 Understanding the Convolution Operation

The primary computational kernel in a CNN is a convolution operation between an input image $f$ and a filter or kernel $g$. Figure 4 shows the resulting feature map after convolving an edge detection filter. In a continuous one dimensional domain, convolving two signals $f \ast g$ is equivalent to taking the integral of the pointwise multiplication of these two function as a function of the amount that one of the signals $g$ is translated. This is summarized in Equation 7.

$$h(x) = f \ast g = \int_{-\infty}^{\infty} f(x-u)g(u)du$$  \hspace{1cm} (7)
When working with images, we use a discrete 2D domain (pixel level rows and columns) formula. Note that the convolution theorem states that under suitable conditions the Fourier transform of a convolution is the pointwise product of Fourier transforms. Equation 8 shows the discrete convolution theorem for image data. In the first method, the spatial domain, a filter of size $x, y$ is passed through the input image of size $(a, b)$ as we iterate through the rows and columns of the image matrix to produce an output feature map. Alternatively, if we perform this operation in the Fourier domain, we can significantly reduce the amount of operations but at the cost of more memory (storing intermediate FFT matrices). Note that $\sqrt{2\pi}$ here is a normalization factor.

\[
\text{feature map}(a, b) = \text{input} \ast \text{kernel} = \sum_{y=0}^{\text{columns rows}} \sum_{x=0}^{\text{rows}} \text{input}(a - x, b - y) \text{kernel}(x, y) = \mathcal{F}^{-1} \left( \sqrt{2\pi} \mathcal{F}[f][g] \right)
\]

4.2 Algorithm Overview

In contrast to MLPs, the neurons in a CNN are arranged in three-dimensional volumes. For image recognition, the input of a CNN is a three-dimensional image object of size $H \times W \times D$, for height, width, and dimensions respectively. The algorithm learns a set of filters $K$ which are convolved or slide across the image to produce an output volume. The size of the filter is called the receptive field $F$ and the stride $S$ that the filter takes across the image determines the size of the output volume. Note that the filter has size $F \times F$ as filters are generally cubic volumes. To perform classification and generate a likelihood, the final layers in a CNN is a multi-layer perceptron.

Similar to the MLP, the goal of learning in a CNN is to learn filters (weights) that reduce error on the output by learning different features about a dataset (images). For example, filters may learn to become rotation, scale, and translation invariant or learn about different kinds of textures in an image. The total number of weights in a CNNs can be summarized as:

\[
\sum_{i=0}^{\text{Conv. Layers}} (F_i)^2 \times K_i + \sum_{j=0}^{\text{Clas. Matrix}} \hat{W}^j
\]

The left half of the equation is the sum of all the 3D filters in each conv layer $i$ with dimension $F \times F \times K$. The right half is the classification layers starting from layer $j$ and is a sum over all classification weights $\hat{W}$ from $j$ to the last classification layer.

Figure 5 shows an example of two filters convolving an input image of size $6 \times 6 \times 3$. The properties of this filter are $K = 2, F = 2, S = 1$. Each output filter map has size $5 \times 5$ and the total volume including the two filters is $5 \times 5 \times 2$.

5 Memory and Computation

There is a vast body of work involved in developing neural systems that can support next generation neural networks. One of the big driving factors for better neural hardware is the observation that the depth of a network correlates with higher accuracy. The set of networks from VGG family of benchmarks\cite{1} show that deeper networks produced high accuracy when experimenting from 16-19 layers on the ImageNet 1k dataset. Google’s winning entry into the Large Scale Visual Recognition Challenge (ILSVRC) in 2014, GoogLeNet\cite{2}, has a 22 layer CNN based Inception.
Network architecture [3]. In 2015, a 30-layer neural network was proposed that produced even higher accuracy on the same ImageNet dataset [4]. Finally, in late 2015 Microsoft edged out the competition with a 152 layer CNN based network [5] in ILSVRC-2015. The first time a CNN won the international image recognition challenge ILSVRC was in 2012 with AlexNet [6] which was a 9 layer network with an error rate of 16%. In 2015 competition, Microsoft’s Deep Residual Network achieved an error rate of 3.57% surpassing the rate of human accuracy.

5.1 Neural Network Memory Requirements

The memory requirements of a neural network are broken into two major parts, static memory and dynamic memory usage. Static memory refers to the total amount of weights or parameters in the network multiplied by 4, the number of bytes per weight. The static memory of the network define the entire architecture and these parameters are small enough to exported onto an embedded system while using less than half a gigabyte of memory for state of the art models. Dynamic memory is the memory usage for the activations that need to be stored during intermediate steps of inference or training. During training, the total dynamic memory usage of a network is an order of magnitude more than the static memory as images are batched and passed through the network in sets of 50 to 250 for better accuracy (stochastic gradient descent with mini-batching). This increases the memory usage significantly.

Figure 6: Memory usage for VGGNet. The network was trained on four Nvidia Titan Black GPUs and took three weeks to converge.
usage by approximately the factor of the mini-batch size. Additionally, gradient matrices need to be stored for every layer for backpropagation and this again increases the memory, this time by a factor of about 2x.

In a CNN the total amount of static memory is the size of each filter times the number of filters in a layer. The filter weights are added to the weights in the classification layer (MLP) to make up the total static memory. **In a CNN more than 90% of the total static memory is in classification layers.**

The total amount of dynamic memory is equal to the size of the resultant feature map after applying every filter through the input feature maps. This is added to the amount of dynamic memory of the activations of the fully connected layer, which is negligible compared to the memory needed for the resultant feature maps. **In a CNN more than 90% of the total dynamic memory is in feature maps.**

To explore this calculation further Figure 10 summarizes the parameters and memory usage of VGGNet[1], a deep 16-layer CNN and winner of ImageNet competition in 2014. Note that the number of parameters in the convolution layers are significantly less than the parameters in the fully connected layers. This is because the kernel weights themselves are small shared resources local to the entire image. In contrast, the number of parameters in the last three fully connected layers is an order of magnitude more. While this is a rough approximate and does not consider the diverse set of software optimizations that have been implemented, we see that compared to actual memory requirements for VGGNet-256 reported by Nvidia[7] in Figure 7 this approximation is acceptable (30GB calculated vs. 28GB actual).

![Figure 7](image-url)

**Figure 7:** Experimental results from Nvidia about the memory usage and breakdown for various CNNs. Most of the memory usage is in the feature maps[7].

### 5.2 Neural Network Computational Requirements

The naive way to perform convolutions involves a series of nested for loops which iterate through height, width, and depth for the input feature map and the filters. This methods takes \((n - k + 1)^2k^2\) operations and is not scalable to very deep networks. The preferred method is to decompose these image tensors in a way that we can leverage parallel compute kernels. In modern libraries, the operations performed in a CNN are matrix-matrix multiplies and vector-matrix multiplies based on the image to column decomposition[5]. For an image of size \(H \times W \times C\) a single receptive field will have size \(K \times K \times C\). This receptive field will be reshaped to a column of \(K^2 \times C\) elements and a matrix of \(K^2 \times C \times N\) where \(N\) is the number of receptive fields in the input feature map. We can perform a similar operation for the filters, reshaping the filters of size \(K \times K \times C\) to rows and create a matrix of \(K^2 \times C \times D\) where \(D\) is the number of filters being used. At this point, we perform a matrix multiply to obtain a \(D \times N\) result which we can reshape to produce an output tensor. In terms of complexity this method which was implemented in early version of Caffe[9] has a time complexity of \(O(N^2K^2)\) for the im2col operation and an additional \(O(N^2K^2F)\) for the dense matrix multiplies. This operation is summarized in Figure 9.

Another area of active research involves leveraging the convolution theorem and decompose the operation into a set of FFTs. Instead of using direct computation, an FFT is performed on the filters and feature maps and combined through a matrix multiply operations. To produce the output
In Feature map: \( I \times I \times C \) Filters: \((K \times K \times C) \times D\)

Matrices:
- \((K^2C) \times O\) matrix
- \(D \times (K^2C)\) matrix

Out Feature map: \( O \times O \times D \)

D \( \times O\) matrix

Reshape
Reshape
Reshape

Matrix Multiply

Figure 8: The im2col approach of convolution. The feature maps and filters are decomposed into matrices and a matrix multiply is performed to obtain the new output feature map.

In Feature map

Filters

FFT

FFT

Matrix Multiply

FFT-1

Figure 9: Taking an FFT approach to convolution. There is a high memory overhead associated with this especially if the size of the filters do not match the size of the input feature map. This is the case during the early layers of a CNN.

feature map, an inverse FFT then needs to be computed. The complexity of the FFT based method requires \( O(n^2 \log n) \) for each FFT and the pointwise products require \( 4n^2 \) calculations for a total of \( 6Cn^2 \log n + 4n^2 \) operations [10] where \( C \) is a constant. While this method can significantly reduce the amount of computation for convolutions it incurs a large memory overhead when the size of the kernels do not match the size of the feature map as the inputs need to be padded [11]. This method works the best during later stages of a CNN after the inputs have been downsampled several times and the size of the feature map matches the filter size. There has been interest in other methods for convolution that involve reducing the computation time of matrix multiplication given a specific kernel size (e.g., 3x3 filters). Using Strassen’s algorithm one can reduce the complexity of a matrix-matrix multiply from the naive \( O(n^3) \) to \( O(n^{2.81}) \) by reducing the amount of multiplies. This technique was implemented for 2x2 and 3x3 kernel sizes and shows massive speedup on VGG for small batch sizes [12].

6 The Communication Bottleneck

There are two main approaches to paralleizing a CPU/GPU cluster for neural computation. The first is to use GPUs in a **data parallel** mode where each compute unit has the entire model in memory and computes gradients on different parts of the training data. The second is to employ **model parallelism** to spread the model parameters over all of the compute units, such as GPUs, and synchronize the parameters and gradients after each layer. These methods are employed when the size of the network
itself or the size of each mini-batch does not fit in a compute unit. For both of these approaches, the cost of communication and bandwidth begin to limit scaling benefits when training very deep networks on large clusters resulting in marginal speedup and low accuracy\cite{13}.

6.1 Data Parallelism

In the data parallel approach the model is stored inside each GPU and a different mini-batch of the training data is fed through the GPUs. The forward pass is computed independently for each GPU and for the backward pass the entire gradient matrix need to be synchronized (averaged) and distributed across all GPUs. The pitfalls of using this method are clear when considering two weight matrices (3 fully connected layers) of size 4096x512 as in the Alexnet\cite{6} architecture. With a 6GB/s PCIe link and 2 TFLOP GPUs it takes 2.09\(\mu s\) to compute the gradient for each sample and 2.73\(\mu s\) for each GPU to receive 4096 floats. In a multi-node GPU cluster, this approach would involve having to aggregate gradients in each node and transferring large matrices across a network. This approach makes the most sense when the compute costs per parameter are high and total parameters are low\cite{14}.

6.2 Model Parallelism

Alternatively, a model can be paralleled by distributing the parameters of a layer on multiple GPUs. In contrast to the data parallel technique, each GPU is fed the same mini-batch at the same time. This approach requires frequent communication to synchronize and aggregate all the parameters in each layer and can work well a single GPU cluster that share a high speed communication bus. Problems arise when scaling up to several clusters the cost of synchronization and aggregation across machines at this rate can quickly overwhelm the communication costs. This approach is optimal for networks with many parameters (fully connected layers) as we can split the size of the network across the number of GPUs will only have to communicate small dense matrices between GPUs.

The COTS HPC system \cite{15} successfully uses this model parallel approach in a distributed GPU cluster with a highly tuned tuned CUDA kernel, custom MPI interface, and Infiniband interconnects. This technique obtains just under linear speedup (80% of maximum) of computation with a 64 GPU cluster running an 11 billion parameter sparse autoencoder. One of the insights that allowed them to obtain this speedup was the observation that this network had many sparse weight matrices. The authors noted that in generic sparse linear algebra code tends to be much slower than the highly optimized libraries for dense matrices and opted to decompose the computation of \(Y = WX\) as a large number of small dense matrix multiplies. This allowed them to saturate compute cores and utilize 800GFLOPS on a 1FLOP peak machine in contrast to 300 GFLOPS without this optimization. It is important to note that the sparse autoencoder used to benchmark this system is a fully connected neural network for unsupervised learning and has a different network architecture with respect to CNNs.

![Diagram of Data Parallelism and Model Parallelism](image-url)

Figure 10: Difference between the forward and backward pass for data and model parallelism. In the data parallel approach GPUs G1, G2, and G3 work independently and total parameters in the end. In model parallel, a small amount of communication is done every layer to synchronous activations.
6.3 Hybrid Approaches

The idea of splitting neural networks using either model and data parallelism has been explored in older systems [16][17] but were primarily tested on multi-layer perceptrons. With CNNs both types of parallelism can be utilized with the memory intensive convolution layers running on data parallel mode and the dense fully connected layers in model parallel mode. This distinction between approaches can also be made in terms of neuron activity - we utilize model parallelism for layers where the neuron activity is high (fully connected) and data parallelism for layers where the compute per weight is high (convolution layers). This technique, popularized by [14] shows that utilizing hybrid parallelism allow a cluster of 8 GPUs to utilize up to 80% of the total theoretical bandwidth. The only issue is that there is a slight drop in accuracy but this can be mitigated with larger datasets.

7 Taxonomy and Evaluation of Neural Systems

Hardware support for neural network falls primarily into three categories. The first category is Technology Enabled Acceleration (TEA). This category involves new or emerging technology that has shown promising results in neural net based algorithms. The family of DianNao papers[18][19] works primarily on the problem of the inference by designing accelerators for CNNs ranging from low power camera ASICS[20] to high throughput supercomputer accelerators[21]. Other recent work has explored using memristor based technologies for the repeated store/compute operation in neural networks[22][23][24]. In one example memristor crossbar arrays are utilized to store input weights and perform the dot-product in an analog manner[25]. While this has been explored before the ISAAC accelerator is one of the first that develops an entire crossbar based accelerator which performs 14x better than the state-of-the-art DaDianNao supercomputer accelerator. It is important to note that almost all TEA solutions focus exclusively on improving inference time computation.

<table>
<thead>
<tr>
<th>Technology Enabled Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DianNao</td>
</tr>
<tr>
<td>DaDianNao</td>
</tr>
<tr>
<td>ISSAC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System Enabled Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DistBelief/TensorFlow</td>
</tr>
<tr>
<td>Project Adam</td>
</tr>
<tr>
<td>COTS HPC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm Enabled Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryConnect</td>
</tr>
<tr>
<td>Batch Normalization</td>
</tr>
<tr>
<td>Gradient-less Methods</td>
</tr>
<tr>
<td>SqueezeNet</td>
</tr>
</tbody>
</table>

Table 1: Examples from each category of the neural system taxonomy

The second category is System Enabled Acceleration (SEA). SEA innovations focus on how to scale large scale GPU/CPU clusters for neural networks. For the most part these system level improvements are mostly at the software level with optimizations to minimize communication costs. One of the first modern era systems papers focused on this problem was the Dist Belief system from Google[13]. Although their benchmark was a sparse autoencoder which is very different from a CNN, they showed that the way the net was distributed across the cluster effects evaluation accuracy. To overcome this limitation they proposed new distributed training algorithms that scaled better but ultimately this scaling broke down at around 2000 CPU cores at which point accuracy began to drop. Shortly after, Stanford developed the COTS HPC system, a distributed GPU cluster with off the shelf commodity parts. Their system utilized model parallelism and transformed sparse operations to dense to maximize the GPU throughput. Using these techniques they obtained better scaling at less training time compared Google’s Dist Belief CPU cluster[15]. Microsoft’s distributed system, Project Adam [26] develops a cluster specifically for CNNs and partitions the network across the layer instead of across the network like DistBelief. Additionally, ADAM make a better case for how to load balance between the worker...
machines and the parameter server and places more computation on the parameter server to lower the communication bottleneck. All SEA innovations are focused exclusively on enabling deeper nets, speeding up training time, and improving training accuracy.

The final category is Algorithm Enabled Acceleration (AEA). This category focuses exclusively on innovations in neural network theory to improve training and inference time computation. There has been a large amount of effort in leveraging the error resiliency of neural networks for limited numerical precision computation[27]. While has phenomena has been observed before [28][29], more recently analysis on deep networks have shown that with only a few weights per feature it is possible to predict more than 95% of the weights in a network without loss of accuracy[30]. This has led to a branch of research looking to compress or hash networks to reduce the number of weights and memory footprint[31][32].

Further down the theoretical side, some work has been done to create new algorithms specifically for training that both improve accuracy and scale well to large distributed systems. One technique that has begun to show promise involves moving away from gradient based methods and toward other nonlinear optimization techniques such as the alternating direction method of multipliers (ADMM)[33][34] show that this method can scales well to large cluster sizes when tested on a 5000+ core cluster running a three layer MLP.

In the next section we explore each of these three areas with specific examples.

**Technology Enabled Acceleration**

The first paper in a series of neural network accelerators to show a synthesized (place & route) accelerator hardware was DianNao[18]. The DianNao accelerator which focuses primarily on fully connected layers has a pipelined architecture with three pipeline stages for the essential computations performed in a MLP - multiply, accumulate, apply transfer function. Data is passed from an off-chip DRAM into on chip scratchpad memory to store intermediate buffers of input and output neurons. These buffers can only process a small amount of neurons at a time. By placing data closer to the logic some of the off-chip access are reduced but the main problem with this implementation is the bandwidth required for the rest of the off-chip accesses to obtain the next set of inputs. It requires a link operating at at 470GB/s which is unrealistically high considering the best memory controller on the market primarily can only provide between 200-250GB/s. In addition, the authors make a claim that this accelerator can be applied to CNNs but this implementation actually scales very poorly to CNNs as DianNao would decompose images into 1D vectors before compute convolutions rather than exploiting locality in the the 2D feature maps.

ShiDianNao[20] tries to address the issue of poor convolution implementation by designing an architecture specifically for the high amount of matrix-matrix multiplies in the convolution layers. This system was built as a peripheral co-processor for camera system and would sit near the sensor and process data immediately after the image is taken. In this architecture an on-chip SRAM is used to store all of the weights and neurons and a 2D mesh of elements (PEs) exploits the regular structure of CNN matrix based computation. Similar to DianNao the on-chip memory, SRAM in this case, stores the input and output neurons and have to store the feature maps and activations as well. Paradoxically this means this architecture can only store 64k of parameters which is three orders of magnitude less parameters than AlexNet. This is without even beginning to consider the memory needed to store the output feature maps which is several times more. The next iteration of DianNao, DaDianNao[21] broadens the scope of this system by taking a tile based approach for they call a neural "supercomputer". The architecture uses a tile based architecture to compensate for the high off chip DRAM accesses in the original DianNao architecture and make two main optimizations to overcome this limitation. The first is to utilize high storage density eDRAM banks to store weights and the second was to adopt a multi-node system to distribute weights across a cluster. One tile can be thought of as a single DianNao accelerator with four high storage density eDRAM banks to store weights with nodes. Each node is networked internally through a fat tree which the broadcasts neurons to each tile. The authors envision this system as a datacenter level chip for which many nodes can be interconnected and report up to a significant speedup over state of the art GPUs. One issue with this system is that the size of the internal and external eDRAM are far too small to accommodate even a moderately sized image recognition network such as AlexNet. Although the claim is that this network works for CNN training as well as inference, a 2GB network like AlexNet equates to a 64-node cluster with 32MB of eDRAM per node. At 64 node the energy cost of the interconnects between nodes
starts to dominate energy with 30% of total energy going to communication costs, most of which in fully connected layers. The problem stems when trying to address this problem by increasing the on-chip eDRAM as it leads to significantly higher design and manufacturing costs. Trying to map a network such as VGG which requires up to 50GB of memory then becomes infeasible.

As noted before, one of the main problems with the DaDianNao is the energy cost of intra-node communication between the fully connected layers. The convolution layers can stay largely localized to a node as each filter can be in a different node but the dense classification layers are the source of the communication overhead. The EIE: Efficient Inference Engine[35] accelerator attacks this problem by focusing exclusively on improving the throughput on the fully connected layers on the network. EIE is a good example of integrating AEA optimizations into a TEA technique. Deep Compression [36] is a technique to reduce the number of parameters through compression and encoding but the weight index need to be looked up during inference. Additionally, the increased sparsity of the network also requires systems for efficient sparse matrix operations. EIE approaches this problem by encoding the sparse weights in a denser column format and operates on this new data structure entirely in SRAM. Compared to the DaDianNao accelerator EIE is able to perform at 2.9x, 19x and 3x better throughput, energy efficiency and area efficiency. More about deep compression is discussed in the AEA section later in the paper.

Table 2: Comparison of various TEA based accelerators. The Eyeriss, ISSAC and EIE accelerators were omitted due to the vast number of configurations which made it difficult to obtain a fair comparison.

<table>
<thead>
<tr>
<th>publication</th>
<th>type</th>
<th>platform</th>
<th>theor. GOp/s</th>
<th>power W</th>
<th>power eff. GOp/s/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cavigelli et al. [37]</td>
<td>CPU</td>
<td>Xeon E5-1620v2</td>
<td>118</td>
<td>230</td>
<td>0.51</td>
</tr>
<tr>
<td>Cavigelli et al. [37]</td>
<td>GPU</td>
<td>GTX780</td>
<td>3977</td>
<td>sd:200</td>
<td>20</td>
</tr>
<tr>
<td>cuDNN R3 [38]</td>
<td>GPU</td>
<td>Titan X</td>
<td>6600</td>
<td>d:250</td>
<td>26</td>
</tr>
<tr>
<td>Cavigelli et al. [37]</td>
<td>SoC</td>
<td>Tegra K1</td>
<td>365</td>
<td>s:11</td>
<td>33</td>
</tr>
<tr>
<td>Microsoft [39]</td>
<td>FPGA</td>
<td>Arria 10 GX1150</td>
<td>1400</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td>ConvEngine [40]</td>
<td>synth.</td>
<td>45nm</td>
<td>410</td>
<td>c:1.0</td>
<td>410</td>
</tr>
<tr>
<td>DianNao [18]</td>
<td>layout</td>
<td>TSMC 65nm</td>
<td>452</td>
<td>c:0.485</td>
<td>0.93</td>
</tr>
<tr>
<td>DaDianNao [21]</td>
<td>layout</td>
<td>ST 29nm LP</td>
<td>2090</td>
<td>n:15.9</td>
<td>130</td>
</tr>
<tr>
<td>ShiDianNao [20]</td>
<td>layout</td>
<td>TSMC 65nm</td>
<td>128</td>
<td>c:0.32</td>
<td>400</td>
</tr>
<tr>
<td>NeuFlow [41]</td>
<td>layout</td>
<td>IBM 45nm SOI</td>
<td>1280</td>
<td>d:5</td>
<td>256</td>
</tr>
<tr>
<td>NeuFlow [41]</td>
<td>layout</td>
<td>IBM 45nm SOI</td>
<td>320</td>
<td>c:0.6</td>
<td>533</td>
</tr>
<tr>
<td>HWCE [42]</td>
<td>layout</td>
<td>ST 28nm FDSOI</td>
<td>37</td>
<td>c:0.18</td>
<td>205</td>
</tr>
<tr>
<td>HWCE [42]</td>
<td>layout</td>
<td>ST 28nm FDSOI</td>
<td>1</td>
<td>c:0.73m</td>
<td></td>
</tr>
<tr>
<td>Origami [43]</td>
<td>silicon</td>
<td>umc 65nm</td>
<td>196</td>
<td>c:0.51</td>
<td>384</td>
</tr>
<tr>
<td>Origami [43]</td>
<td>silicon</td>
<td>umc 65nm</td>
<td>74</td>
<td>c:0.093</td>
<td>795</td>
</tr>
</tbody>
</table>

Figure 11: (a) DianNao accelerator. (b) The right side shows a single tile with an NFU (neural functional unit) which is similar to (a). The left shows a chip with 16 tiles.
System Enabled Acceleration

The training algorithm for neural networks is a computationally intense procedure due to the nature of the backpropagation algorithm discussed in Section 3. There are three main steps when performing training - the forward pass, backward pass, and the gradient update. The large memory and computational footprint of training come from the fact that the images are trained in mini-batches and have to perform several million updates in an iterative manner using stochastic gradient descent optimization. Mini-batch processing quickly blows up the memory footprint of the network and the serial formulation of SGD makes it less conducive to parallelization.

The DistBelief\[13\] system from Google first tackled this problem by throwing away notion that updates in SGD had to be synchronous and exploited model parallelism within a machine and data parallelism with several model instances or replicas. Model replicas refer to a single multi-machine compute node that distribute the network model across machines. During training, each model replica is fed a different mini-batch of data and sends the gradients to a central parameter server. The parameter server updates the model parameters with the gradient values and then distributes the new gradients among the model replicas. Each model replica is stored in a 4 machine node exploiting the idea of model parallelism reducing the per machine load. To perform the actual training the DistBelief system tested two separate approaches to the training problem. The first approach was to have a central parameter server with model replicas asynchronously updating the parameter server with a variant of asynchronous SGD, Downpour. With Downpour SGD the model replicas are run independently of each other and each shard in the parameter server updates independently (talk about shard). Before processing a mini-batch the each machine in the model replica requires a different piece of the model parameters to make up the full network. After processing, the corresponding gradients are sent to the parameter server. The second technique uses a quasi-Newton optimization method known as L-BFGS, with the asynchronous variant dubbed SandBlaster. The main idea in the SandBlaster based architecture is to have distributed parameter manipulation. Instead of the direct communication between the nodes and the parameter server a coordinator machine issues commands to coordinate the optimization process by issuing commands for multiply, dot product, and scaling. Unlike Downpour which has access to multiple data shards per node, SandBlaster has one centralized data block. The reason for this approach is because the BFGS algorithm requires computation of the gradient on the whole dataset to make an update. While this can mean a better optima it also means it does not scale well with the number of examples as the computational cost per gradient iteration is high\[44\]. The authors found that for both of these implementations, the asynchronous techniques were able to outperform their synchronous variants on the distributed system. When looking at how these algorithms scaled both of these methods started to show diminishing returns after 5000 cores. In fact, Downpour SGD began to loss accuracy training beyond 1500 cores. While SandBlaster did not lose accuracy as the system scaled up, there was almost no improvement in training time going from 2000 machines to 5500 machines. One reason for this could be that the cluster was not saturated enough to take advantage of the number of cores, but the network was trained on a very large 1.7 billion parameter neural network with 21 million images from ImageNet.

Microsoft’s work on Project Adam\[26\] recognizes that communication between model replicas and the parameter server was the primary bottleneck when scaling up the system. Adam adapts an asynchronous SGD algorithm similar to DistBelief, but does not use the same data parallelism scheme with multiple model replicas updating the parameter server. Rather, they chosen to split the model on multiple machines across layers, i.e. every machine will hold one vertical slice of the whole model. Using this approach they are able to make two further optimizations given that every machine has a very small piece of the entire network. Now the working set of the network can completely fit into the L3 cache of a machine. This purely model parallel approach works well in a distributed system as the each update is small. To attack the problem of high communication between the model replica and parameter server Adam chooses to off-load some of the computation of the gradient updates into the parameter server. In the DistBelief system, the purpose of the parameter server was just to take the gradient updates that each machine calculated and apply it to the parameter server. This can lead to a bottleneck in the network as the gradient maps take a large pool of memory as they are aggregated over each layer. Instead Adam proposes to just send the error gradients calculated after every layer to the parameter server and let the parameter server calculate the gradient for every machine and apply it. This has the benefit of load balancing the computation between model and parameter machines and overcomes the memory bandwidth problem that would occur when increasing the number of
Figure 12: (a)-DownPour: The parameter server exchanging weights with the workers which pass new weights up to the parameter server. (b)-SandBlaster: A coordinator manages communication and one entire batch is sent to all model replicas

In DownPour, the parameter server exchanges weights with the workers, which pass new weights up to the parameter server. SandBlaster uses a coordinator to manage communication, and an entire batch is sent to all model replicas.

Activates cores per machine. Similar to the DistBelief paper, Adam demonstrates that having asynchronous updates to gradient descent acts as a form of regularization and improves overall accuracy on the test set. Compared to DistBelief, this system was tested on a quarter of the cluster size, so it's difficult to speculate how it scales beyond 1000 cores. In DistBelief, scaling was also linear until about 2000 cores at which point the scaling tapered off and accuracy began to drop.

**Algorithm Enabled Acceleration**

For years, the development of new neural networks and better training algorithms has been largely orthogonal to what kind of hardware might drive the usage of these algorithms. As the application scope and potential of neural networks continue to expand, some research has been placed in developing new techniques at the algorithm level to enable low power inference or scalable parallel computation. In the realm of inference, finding a way to lower the memory and energy footprint of neural networks is something that has traditionally been approached from a purely hardware standpoint as seen by examples such as the DianNao[18] and ORIGAMI[43]. To hardware researchers, the algorithms are a recipe that will stay essentially the same and hardware just has to optimize the tools. In reality, the state of the art in neural network research is changing so quickly that any chip that is built can be rendered obsolete in a matter of months. Neural networks are nowhere close to being a mature computation and any chip built either has to target the most basic neural network primitives such as matrix multiples or convolution[40] which only optimize throughput but not necessarily memory. To place these powerful algorithms in low power devices such as mobile phones, contact lenses, and next generation IoT devices, an approach that both optimizes throughput, energy, and memory must be taken. One way to do this is to bring in some insights and ideas from the theoretical side.

It has long been noted that neural networks are resilient to perturbations and these small variance even have the potential to enhance generalization in neural networks[30]. Several recent papers work toward reducing parameter complexity during both inference and training by compressing or approximating weights. By finding a way to relax the constraints of having high precision operators or reducing the number of multipliers, both energy and storage space can be reduced. While most of the work in neural compression is applied toward inference, as training requires the network to accumulate small changes in the parameters with high precision, there has been some work exploring the space of limited precision training parameters[27]. One way to approach this problem is to reduce the amount of multiplies by having low precision multiplies with high precision accumulates. In this approach, the parameters for updates are kept at high precision while the number of bits for forward and backward propagation are varied. The results showed that by utilizing 12-bit dynamic fixed-point computation they are able to maintain accuracy compared to 32-bit computations[45]. These experiments open the possibility that the weights in a network can be low precision as long as the parameter updates are high precision. This was the approach taken for the BinaryConnect series.
of papers that take this approach to an extreme by binarizing the weights during the forward and
backward pass. Psuedocode for this operation is shown in Figure 13. BinaryConnect was shown to

Algorithm 1 SGD training with BinaryConnect. C is the cost function for minibatch and the functions
binarize(w) and clip(w) specify how to binarize and clip weights. L is the number of layers.

Require: a minibatch of (inputs, targets), previous parameters \( w_{t-1} \) (weights) and \( b_{t-1} \) (biases),
and learning rate \( \eta \).
Ensure: updated parameters \( w_t \) and \( b_t \).
1. Forward propagation:
   \( w_h \leftarrow \text{binarize}(w_{t-1}) \)
   For \( k = 1 \) to \( L \), compute \( a_k \) knowing \( a_{k-1} \), \( w_h \) and \( b_{t-1} \)
2. Backward propagation:
   Initialize output layer’s activations gradient \( \frac{\partial C}{\partial o_{Lt}} \)
   For \( k = L \) to \( 2 \), compute \( \frac{\partial C}{\partial o_{k-1}} \) knowing \( \frac{\partial C}{\partial a_k} \) and \( w_h \)
3. Parameter update:
   Compute \( \frac{\partial C}{\partial w_h} \) and \( \frac{\partial C}{\partial b_{t-1}} \) knowing \( \frac{\partial C}{\partial a_k} \) and \( a_{k-1} \)
   \( w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_h}) \)
   \( b_t \leftarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}} \)

Figure 13: Pseudocode for BinaryConnect. The binarization process happens in the forward propagation but
not in the parameter updates.

have close to state-of-the-art results while reducing the number of multiplies by 2/3. The authors
note that by binarizing the network in this way it acts as a form of regularization which improves the
overall network performance.

Other methods of compressing such as hashing the weights during training has also been explored.
In this method, parameters are hashed in a way that connections within the same hash bucket share a
single parameter value. At this point a virtual weight matrix is created which turns the fully connected
network into a sparsely connected network with shared weight filters, similar to the kernels in a CNN.
The results show that this technique works well when compressing the network to 1/8 its normal
size and going to 1/64 of the size error does increase slightly relative to an uncompressed network
(1.5% to 3%). HashNets were tested on a small benchmark of 3 and 5 layer MLPs but the accuracy
drops as the number of network layers increases. This may become an issue occurs when
trying to apply this to deep unsupervised fully connected networks such as an autoencoder for
which hashed training may severely hurt test accuracy. Another interesting experiment the authors
ran want to expand the hashed virtual network. While this obviously does not save any space they
noted that expanding the virtual weight matrix by 4-8x accuracy improved. In some respects this is to
be expected as the network is has a more rich space to learn features with more weights but it is an
interesting property.

Applying compression to the inference process has a wide range of implications for embedded
systems and has spawned a series of work in compressing the weights of a neural network.
The main idea here is to take an already trained network and prune remove all connections with
weights below a certain threshold from the network. After this is completed, the next step is to
perform a new batch of training on this now sparse network until acceptable accuracy is achieved.
The results of this technique show that benchmarked against very deep networks such as VGG this second
training process with sparse connections increases the error compared to the reference network by a
negligible amount while compressing the network at a rate of 9-13x. It should be noted that this is not
an optimization to lower the memory usage during training and in fact increase total training time as
there are now two training phases. The authors address this concern by stating that the cost of training
the network a second time only happens when the model is ready for deployment rather than during
model prototyping but this may be an issue in the future with as networks become deeper. Modern
state of the art networks can take months to train and the iterative pruning and training processes
incurs an additional overhead of about 2.5x. A later work, Deep Compression further improves on
this idea by adding two more stages to the compression pipeline, quantization and Huffman encoding.
The idea behind quantization is similar to in which the authors cluster weights using k-means in
order to use less bits per weight. In that paper quantization was able to reduce network parameters by
about 16x without loss in top-five accuracy. In Deep Compression, the largest benefit comes from the pruning-retraining phase with about 10x improvement, the quantization reduces the remaining network by 3x and the encoding reduces the remaining weights to about 1.5x. This results in a 35-49x reduction on the original network with essentially no loss in accuracy benchmarked against AlexNet.

Reducing training time and improving parallelization with new optimization techniques is an area where less effort has been focused due to the complexity of the training problem and generally marginal improvements in model accuracy. One way to approach this problem is to attacking the communication bottleneck by reducing parameters and lower the amount of data sent across servers. Where Deep Compression focused exclusively on compression inference, the SqueezeNet architecture introduces a new approach for developing CNNs with 50x fewer parameters. In this architecture, a normal convolution step is replaced with squeeze-expand stages which first reduces the total parameters with 1x1 filters and then expands them to stacked 1x1 and 3x3 filters. While parameter reduction is useful, the memory requirements go up drastically. The AlexNet baseline that they compare against has 240MB of parameters and uses about 2GB of memory for training with batch size of 128. The authors do not report what the batch size of the network during training is, but the default batch-size in the released code is 512. Performing the same back of the envelope calculation on SqueezeNet as performed on VGG, SqueezeNet uses 30GB of memory an increase of 15x over AlexNet. Note that 30GB is the same memory footprint of VGG during training but where VGG had an error rate of 6.8% SqueezeNet has an error rate of 20%. In summary SqueezeNet is able to reduce parameter of an equivalent network by 50x but increases memory usage by 2-10x and results in 3x less accuracy than a network with the same memory footprint.

Moving away from compression, a few techniques incorporated directly into the training process have shown to both improve accuracy and reduce total training steps using techniques like batch normalization. As a pre-processing steps in neural networks data is normalized to speedup training and reduce chances of being stuck in a local optima. As the inputs are processed through a neural network the benefits of normalization loss its effectiveness as the parameters change to become larger or smaller. This phenomena is referred to as internal covariate shift and the authors propose normalising data in each mini-batch in the middle of the network. This reduces the total number of training steps by 14x, as the network can tolerate more noise and learning rates can be raised, and improves accuracy over the 22 layer GoogLeNet baseline without having to increase the size of the network.

8 Conclusion

This survey reviews state of the art algorithms in neural networks and the various approaches taken to optimize speed, accuracy, and energy. The hardware, systems, and theory communities are all working toward one goal to enable faster and more accurate networks and in general each community tackles a different neural problem - hardware community works on inference, systems works on training, and theory works on both. While this has worked well in the past, new research ideas in this area cannot be developed without a good understanding in all three. The most interesting papers
are the ones that can synthesize ideas from two or more of these groups to produce a system well informed theoretically and grounded in real hardware. When AlexNet won the ILSVRC in 2012, it was the result of a new network architecture trained with an efficient software kernel and run on high performance hardware. It was an understanding and optimization of all three areas TEA-SEA-MEA that enabled AlexNet to outperform decades of computer vision research and change the landscape of every research community.
References


