Acoustic Models for Speech Recognition Using Deep Neural Networks Based on Approximate Math

by

Leo Liu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2015

Certified by . . . .
James R. Glass
Senior Research Scientist
Thesis Supervisor

Accepted by . . . .
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Acoustic Models for Speech Recognition Using Deep Neural Networks Based on Approximate Math

by

Leo Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Deep Neural Networks (DNNs) are effective models for machine learning. Unfortunately, training a DNN is extremely time-consuming, even with the aid of a graphics processing unit (GPU). DNN training is especially slow for tasks with large datasets. Existing approaches for speeding up the process involve parallelizing the Stochastic Gradient Descent (SGD) algorithm used to train DNNs. Those approaches do not guarantee the same results as normal SGD since they introduce non-trivial changes into the algorithm. A new approach for faster training that avoids significant changes to SGD is to use low-precision hardware. The low-precision hardware is faster than a GPU, but it performs arithmetic with 1% error. In this arithmetic, $98 + 2 = 99.776$ and $10 * 10 = 100.863$.

This thesis determines whether DNNs would still be able to produce state-of-the-art results using this low-precision arithmetic. To answer this question, we implement an approximate DNN that uses the low-precision arithmetic and evaluate it on the TIMIT phoneme recognition task and the WSJ speech recognition task. For both tasks, we find that acoustic models based on approximate DNNs perform as well as ones based on conventional DNNs; both produce similar recognition error rates. The approximate DNN is able to match the conventional DNN only if it uses Kahan summations to preserve precision. These results show that DNNs can run on low-precision hardware without the arithmetic causing any loss in recognition ability. The low-precision hardware is therefore a suitable approach for speeding up DNN training.

Thesis Supervisor: James R. Glass
Title: Senior Research Scientist

Acknowledgments

First and foremost, I would like to thank my advisor, Jim Glass, for his guidance and generosity. It was Jim's encouragement that originally persuaded me to pursue my M.Eng., and I am grateful for the opportunity that he gave me. I also thank Ekapol Chuangsuwanich, who worked with me on this project and guided many of the research questions. I have worked with Ekapol for three years now, and he taught me most of what I know about deep learning and speech recognition.

I would also like to thank Najim Dehak, who worked with me on another project, and Yu Zhang, who helped me with DNN-related software. Thanks to my officemates and everyone else in SLS. You guys have been great friends and mentors to me. I am so blessed to have been a part of this group of people.

This work is a collaboration with Singular Computing LLC. I am grateful to Joe Bates for the opportunity to work on this project. Singular is open to further collaboration and may be reached at contact@singularcomputing.com.

To Sushi, Emily, Madhuri, Chelsea, Quanquan, Jiunn Wen, and Lora: I would not have made it here without you; thank you.

Lastly, I would like to thank my mom and dad for their love and support. You two are the main reasons for my success.

This thesis was supported in part by Qatar Computing Research Institute (QCRI).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning is being used to solve problems that are increasingly important in day-to-day life. Machine learning techniques have been applied in fields such as bioinformatics, computer vision, economics, fraud detection, robotics, and speech. Recently, one of the most successful machine learning techniques has been Deep Neural Networks (DNNs). For instance, DNNs have currently been used extensively in the speech recognition community. For Automatic Speech Recognition (ASR), DNN-based models result in 10-30% relative improvement in word error rates over traditional methods [23]. Unfortunately, training a DNN is a time-consuming process, even when a graphics processing unit (GPU) is used to speed it up. In a speech task that uses 81 hours of speech as training data, the DNN takes more than half a day to train. It is general knowledge that speech recognizers benefit greatly from larger amounts of data; some systems use training sets with thousands of hours of speech. At the present rate, those DNNs would take weeks to train. Worse yet, selecting the best DNN architecture and tuning a DNN's hyper-parameters typically require training many DNNs. The slow training is a significant impediment to DNN researchers.

Efforts to speed up the algorithm have focused on parallelizing the training algorithm, Stochastic Gradient Descent (SGD), to take advantage of more hardware. Chen et al. introduced pipelined SGD, which achieves 3.3x speedup with 4 GPUs. Unfortunately, speedup is limited by the number of layers in the network and the batch size, which must be reduced to maintain the same error rates [16]. Google's DistBelief system replicates a DNN across thousands of CPUs, training each with a subset of the data and periodically syncing the weights. Speedup is achieved for large DNNs with billions of weights, but not for moderately-sized DNNs used for speech [41]. Seide et al. achieved a speedup of 6.3 on a moderately-sized DNN by replicating the DNN on 8 GPUs and synchronizing the gradients. Communication is kept to a minimum with one-bit quantization of the gradients [40]. All these efforts have the drawback that they no longer use the original SGD training, but an approximation of it. They all required careful tuning to produce identical results to the SGD, and it is not clear whether they can always produce identical results. It seems that SGD can be sped up without modification only if the hardware gets faster.

One way to implement faster hardware is to use low-precision arithmetic. Certain algorithms can still produce good results even when using low-precision arithmetic. For those algorithms, the low-precision arithmetic is an acceptable trade-off for the faster computation afforded by the hardware. This thesis seeks to determine whether DNN training is one of those algorithms.

Although many types of low-precision arithmetic might exist, we focus our experiments on the arithmetic found in one particular hardware solution. Singular Computing, led by Joseph Bates, developed hardware that is 50 times faster per watt than a GPU. The Singular chip, as it is called, gains its speed by performing low-precision arithmetic, in which arithmetic operations produce values that are up to 1% away from the correct values. In this arithmetic, $98 + 2 = 99.776$ and $10 * 10 = 100.863$. The precision is much lower than that of floating point arithmetic, which is traditionally used when training DNNs [11].

This thesis answers the following question: can a DNN based on low-precision arithmetic perform as well as a DNN based on floating point arithmetic? If the answer is yes, then low-precision hardware, such as the Singular chip, can be used to train DNNs faster without harming their ability to produce state-of-the-art results. We refer to DNNs that use low-precision arithmetic as "approximate DNNs" and DNNs that use traditional float-based arithmetic as "float DNNs." This thesis makes the following contributions:

1. We add the low-precision arithmetic to a general-purpose math library that can be used to implement a variety of machine learning algorithms.

2. We implement an approximate DNN using the library.

3. We evaluate the approximate DNN on the TIMIT phoneme recognition task and the WSJ speech recognition task. For those tasks, we compare acoustic models based on approximate DNNs to ones based on float DNNs.

The evaluations show that an approximate DNN performs just as well as a float DNN. If Kahan summations are used, both types produce similar frame classification errors. When used as acoustic models in speech recognition, they generate near-identical recognition error rates. The low-precision arithmetic works, which means that the Singular chip can be used to speed up DNN training.

This thesis is organized as follows: Chapter 2 provides background on DNNs, speech recognition, the DNN speed problem, and the low-precision arithmetic solution to that problem. Chapter 3 describes our implementation of the approximate DNN. Chapter 4 compares the approximate DNN to the float DNN on a small computer vision task (MNIST) and two larger speech recognition tasks (TIMIT and WSJ). Chapter 5 concludes.

Chapter 2

Background

This chapter provides background on Deep Neural Networks (DNNs) and the Stochastic Gradient Descent (SGD) training algorithm. We describe the use of DNNs in speech recognition and explain why training speed is an impediment to speech research. Existing approaches for speeding up DNN training are summarized. The low-precision hardware, which is another approach for speeding up training, is introduced. We describe the arithmetic used by the hardware and the software implementation of the arithmetic. The approximate DNNs that we evaluate are based on the low-precision arithmetic.

2.1 Deep Neural Networks

DNNs are used in classification tasks in which the goal is to assign one of C classes, or labels, to each input. The input is a feature vector that describes some observation. For example, in the MNIST classification task from machine vision, each input is a vector of pixel intensities taken from images of isolated handwritten digits. The classes are the digits 0-9, and the task is to predict the digit present in each image [29]. To solve classification tasks, we build a model, such as the DNN, that can estimate a posterior probability, $P(class|input)$, for each class. For simple classification tasks, such as the digit identification problem described above, the predicted class for an input is simply the class with the highest posterior probability.

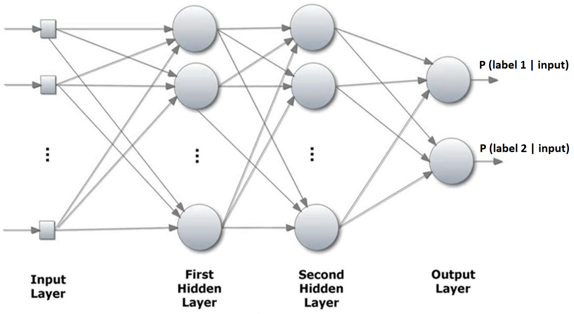


Figure 2-1: A DNN with 2 hidden layers and an output layer [18]. Each hidden layer typically has thousands of neurons, and the output layer has one neuron per class.

For a more complicated task, such as speech recognition, the posterior probabilities are used with other systems to solve the task.

Consider a DNN that has input x, which is the feature vector, and output y, which is the vector of class probabilities. The DNN consists of several hidden layers and an output layer. Each hidden layer typically has thousands of neurons. The output layer has C neurons, one for each class. An example DNN is shown in Figure 2-1.

Denote the output of each layer as y_n, where n is the layer number. The first hidden layer starts at $n = 1$. The output layer has $n = N$, where N is the number of layers in the network. The input can be treated as a pseudo-layer: $y_0 = x$. Each neuron has a weighted connection from every output in the previous layer; there are no intra-layer connections. An example neuron is shown in Figure 2-2.

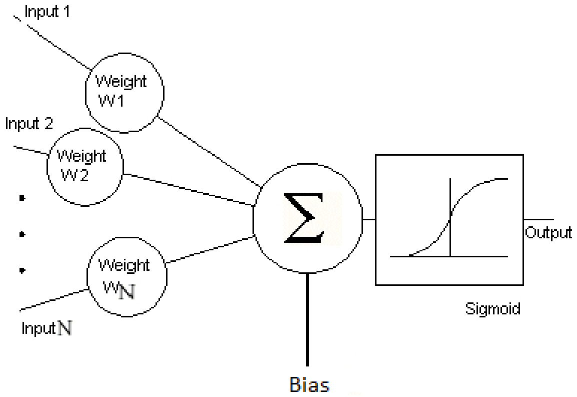


Figure 2-2: An individual neuron [34]. The output of the neuron is the weighted sum of its inputs plus a bias, passed through an activation function, such as the sigmoid.

The output of each neuron is a weighted sum of its inputs and a bias term, which is passed through an activation function F. The output of each neuron in the nth layer, $y_{n,j}$, can be computed as:

$$y_{n,j} = F(\sum_i (y_{n-1,i} * w_{n,i,j}) + b_{n,j}) \tag{2.1}$$

where neurons in layer n are indexed by j and neurons in layer $n-1$ are indexed by i. The weight from i to j is denoted by $w_{n,i,j}$, and the bias of neuron j is denoted by $b_{n,j}$. For the hidden layers, common activation functions include the sigmoid, tangent, and rectifier functions. We use the sigmoid activation:

$$F(z) = \frac{1}{1+e^{-z}} \tag{2.2}$$

The output layer, also called the softmax layer, uses the softmax activation function:

$$F(z_j) = \frac{e^{z_j}}{\sum_j (e^{z_j})} \tag{2.3}$$

Since the softmax normalizes y_N to the range [0, 1] and ensures that it sums to 1, the outputs can be treated as posterior probabilities.

It is believed that the power of a DNN comes from its depth. Layers closer to the input are able to learn low-level representations of the input, while layers further up are able to learn higher-level representations [24].

2.2 Training DNNs

The DNN is a highly non-convex model with multiple local optima; the most popular method of training one is Stochastic Gradient Descent (SGD). Using SGD, we train the DNN to minimize an objective function. Possible objective functions include cross-entropy and mean squared error. Variations of those functions also exist; some objectives have L1 and L2 regularization terms for example [33]. We use cross-entropy, which is computed as:

$$CE_k = -\sum_c ln(y_{N,c,k}) * t_{c,k} \tag{2.4}$$

where the training example is indexed by k. The final layer's output is denoted by the vector $y_{N,k}$, the classes are indexed by c, and t_k is a vector of the one-hot encoded labels for that example. SGD is a supervised training algorithm in which we train the network to predict labels of previously-labeled training data in the hope that the network will be able to predict labels for unlabeled data. The weights and biases of the DNN are first initialized; SGD then proceeds as shown in Algorithm 1 [12].

Algorithm 1 Stochastic Gradient Descent algorithm (one epoch)

```
Shuffle the training dataset, and divide it into evenly-sized batches.
for each batch of examples
    Compute the DNN outputs, $y_{N,k}$, with respect to the current weights and biases.
    Compute the cross-entropy, $CE_k$, for each example $k$.
    for each layer $n$ from $N$ to 1
        Compute the gradients $\frac{dCE_k}{dw_{n,i,j}}$ and $\frac{dCE_k}{db_{n,j}}$ for each example $k$.
        Sum the gradients across the examples to get one gradient per weight and bias:
            $$\frac{dCE}{dw_{n,i,j}} = \sum_k \frac{dCE_k}{dw_{n,i,j}}$$
            $$\frac{dCE}{db_{n,j}} = \sum_k \frac{dCE_k}{db_{n,j}}$$
        Update each weight and bias according to the following formulas:
            $$w'_{n,i,j} = w_{n,i,j} - \alpha * \frac{dCE}{dw_{n,i,j}}$$
            $$b'_{n,j} = b_{n,j} - \alpha * \frac{dCE}{db_{n,j}}$$
    end for
end for
```

Computing the outputs of the DNN is referred to as forward propagation since it involves using each layer's output to compute the next layer's output. Computing the gradients is referred to as backpropagation since it involves propagating the error from the last layer back to the first layer. The interested reader can find the formulas for backpropagation in [33]. The process described above is repeated until the network converges. Each pass through the training set is an epoch. The parameter α, shown in Algorithm 1, is a learning rate that controls how much the network learns from each example; it is decreased as convergence slows. The method for setting and decreasing α is called a "learning rate schedule." The convergence of the network is usually monitored on a held-out validation set.

The weights of a DNN can be initialized with random initialization or with pre-training. One of the possible ways to pre-train a DNN is by first training a stack of Restricted Boltzmann Machines (RBMs). The RBM is a probabilistic graphical model. RBMs are stacked together and trained one layer at a time using greedy unsupervised training. Then, the weights of each RBM in the stack are used as the initial weights of the corresponding layer in the DNN. The interested reader can find details of the RBM training algorithm in [22].

During forward propagation, the training examples pass through the network layer-by-layer, with each node computing a dot product for each example. The dot products can be performed in parallel as a one matrix multiplication per layer. Backpropagation is similar computationally: there is one matrix multiplication per layer. The matrix multiplications dominate the runtime. They accounted for 85% of the runtime in one of our own GPU-based DNNs.

DNN training takes a long time for two reasons. First, SGD is a serial algorithm that processes examples batch-by-batch. The runtime is proportional to the amount of training data, and it typically takes dozens, and occasionally hundreds, of passes through the training data for the DNN to converge. Second, each batch of data causes $2N$ matrix multiplications, where N is the number of layers in the network. The matrix multiplications are slow since the matrices have dimensions equal to the number of units in each layer. The computations can be performed about 50 times faster on a GPU than on a CPU, but training is still slow even with the speedup.

2.3 Speech Recognition

Speech recognition is the task of converting speech audio to text. A speech recognizer consists of three parts: the acoustic model, the lexicon, and the language model. The acoustic model is a probabilistic model that describes which phonemes are likely to be present in a section of speech. The lexicon is a probabilistic model of which phonemes are used to pronounce a word. The language model describes the probabilities of different sequences of words. A related task is phoneme recognition, which aims to convert speech into a sequence of phonemes instead of a sequence of words. DNNs make effective acoustic models. Acoustic models based on a Hidden Markov Model (HMM) and DNN hybrid result in 10-30% relative improvement in word error rates over traditional methods, such as the HMM and Gaussian Mixture Model (GMM) [23]. For DNN-based acoustic models, the inputs are acoustic frames and the labels are phoneme states.

The slow training of DNNs is an impediment to speech research. For example, a DNN with 6 hidden layers of 2048 units, and an output layer of 3405 units, takes 50 minutes per epoch, and 16 epochs to converge, or approximately 13 hours total, on a GeForce GTX TITAN Black GPU. That DNN uses 81 hours of recorded speech from the Wall Street Journal corpus as training data [15], but much larger datasets exist. For example, [20] describes a dataset with 3,000 hours of recorded speech, and [26] describes a dataset with 5,780 hours of speech. Using more training data is desirable since it produces lower error rates, but standard SGD would take many weeks on a GPU for the larger datasets.

2.4 Existing Approaches for Faster Training

There are two main approaches to speed up DNN training. The first approach is to replicate the DNN across available hardware. In a naive implementation, the replicas start with identical weights. Each replica processes a subset of the batch and computes its own gradients. The replicas synchronize by sending their gradients to a master replica. The master replica adds the gradients together to compute the weight updates and sends the new weights back to every replica. Once every replica has updated, they all proceed to the next batch. Although the naive implementation is equivalent to true SGD, it does not work for two reasons. First, the communication cost of sending the gradients and weights is too high. Second, computing on small sub-batches is inefficient.

Google's DistBelief system replicates the DNN across multiple machines. Instead of proceeding in lock-step, the replicas push their gradients and receive weight updates asynchronously. With 81 machines, a locally-connected DNN with 1.7 billion parameters achieves significant speedup. However, a moderately-sized, fully-connected DNN with 42 million parameters achieves almost no speedup over the standard non-parallelized GPU-based SGD [17]. The moderately-sized DNN is the kind typically used in speech recognition.

Seide et al. replicate the DNN across multiple GPUs on a single machine. To lower the communication cost, their system compresses each gradient to a single bit using one-bit quantization. The compression is extremely lossy, so they use error feedback to ensure that all gradients are eventually accounted for. They achieve a speedup of 6.3-times on a moderately-sized DNN using 8 GPUs [40].

The second main approach is pipelining. Chen et al. pipeline the DNN by assigning layers to different GPUs. The GPUs are all kept busy by feeding in the next batch of examples before the current batch has finished. Pipelining requires smaller batch sizes to produce comparable error rates. With 4 GPUs, the pipelined system achieves a speedup of 3.3-times. Further scalability is limited by the number of layers and the reduction in batch size [16].

All these techniques have disadvantages. Some do not achieve speedup on speech DNNs. To maintain the same error rates, most require careful tuning and tricks, such as error feedback and smaller batch sizes. It is not always clear if these techniques can produce the same results as the true SGD algorithm. Faster hardware is likely the only way to speed up true SGD.

2.5 Low-Precision Hardware for Faster Training

Faster hardware can be implemented by using low-precision arithmetic. This type of hardware can speed up algorithms, but there is a trade-off. The low-precision arithmetic may worsen the algorithm's results. The faster hardware is useful only if the algorithm is able to produce near state-of-the-art results when using low-precision arithmetic. In this thesis, we evaluate whether low-precision arithmetic can be used to train DNNs. Although different types of low-precision arithmetic might exist, we focus our evaluations on the low-precision arithmetic from one particular hardware solution: the Singular chip.

The Singular chip, developed by Joseph Bates of Singular Computing, is an alternative to the GPU that is 50 times faster per watt. The Singular chip can be used to speed up true SGD without resorting to modifying the algorithm. The parallelized variants of SGD might also benefit from being run on multiple Singular chips, but this thesis is concerned only with true SGD.

The Singular chip gains its speed by performing low-precision Logarithmic Number System (LNS) arithmetic instead of the traditional float-based arithmetic. Traditional float-based arithmetic, such as the one defined by the IEEE-754 standard, is high precision and high dynamic range. IEEE floats have seven significant decimal digits of precision, which means that the arithmetic results in values that are within a factor of 10^{-7} of the correct values [27]. High dynamic range means that a wide range of values can be represented, such as the range of one billionth to one billion. In contrast, the LNS arithmetic used by the Singular chip is low precision and high dynamic range. A wide range of values is supported, but the arithmetic results in values that are 1% off from the correct values [11].

Conventional floats are represented as $significand * base^{exponent}$. For example, 1.2345 is represented as $12345 * 10^{-4}$. The three parts of the float are stored in memory as a single 32-bit value [25]. In contrast, LNS represents a number as its base 2 logarithm. 1.2345 would be represented as $log_2(1.2345)$; the value of the logarithm is stored in memory using a fixed-point representation. A fixed-point representation represents the value in binary, devoting a fixed number of bits to the integer portion of the number and a fixed number of bits to the fractional portion [11].

The Singular chip uses 6 bits to represent the fractional part of a number's logarithm. With 6 bits, increasing or decreasing the fractional part of the logarithm by 1 corresponds to multiplying or dividing the number by $\sqrt[64]{2}$., which is approximately 1.011. Valid numbers in this system are therefore spaced apart by a factor 1.011. The gaps in the number line mean that numbers will be represented with a possible error of 1% from their true values. Singular uses 5 bits plus a sign bit to represent the integer part of the number's logarithm. 5 bits is enough to represent logarithms from 0-31, meaning that a high dynamic range, with numbers from 0 to a billion, is representable [11].

Low-precision LNS can be implemented in hardware with far fewer physical resources. Conventional floating point arithmetic, particularly multiplication and division, require on the order of hundreds of thousands of transistors to implement. LNS arithmetic, on the other hand, is easier to implement in hardware. For instance, multiplication and division in LNS are implemented simply as addition and subtraction of the logarithms, which require fewer transistors. The number of arithmetic processing units, or cores, that fit on a chip is much higher with LNS than with float [11].

The Singular chip is implemented as a SIMD machine that performs parallel LNS operations. When optimized and manufactured using the same process, a Singular core is 1% the size of a GPU core. This is the source of Singular's 50x increase in compute speed, per watt or per area. Compute speed is also affected by memory bandwidth; Singular's memory bandwidth can be 500 times that of a GPU [9]. The Singular design is shown in Figure 2-3.

A prototype Singular board from mid-2015, shown in Figure 2-4, has 34,000 cores, performs 6 trillion operations per second (Tops), has 13 TB/s of memory bandwidth, and draws 50 watts [10, 9]. For comparison, a high-end commercial GPU has 3,000 cores, performs 5.1 Tops, has 0.3 TB/s of memory bandwidth, and draws 250 watts [38, 28]. In general, using the same silicon technology and with significant effort to optimize implementations, the Singular architecture shows about 50x better speed/power than GPUs [10]. A scaled-up version of the Singular chip (not the prototype) would speed up DNN training with true

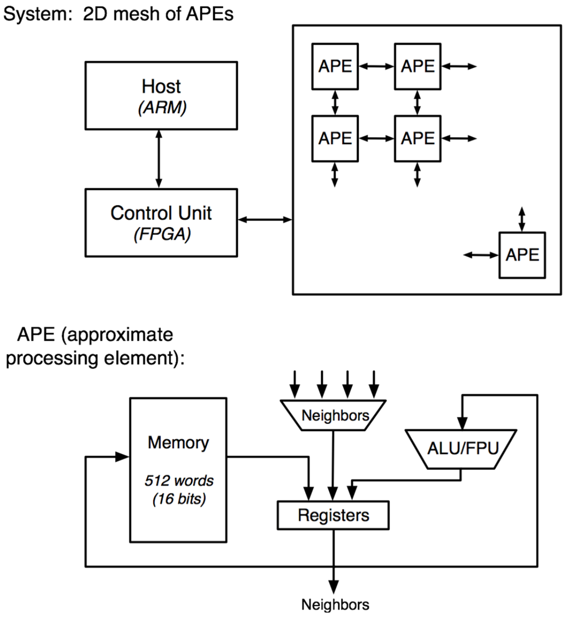


Figure 2-3: Bottom: A single LNS core, also known as an Approximate Processing Element (APE). Top: A Singular chip consists of a grid of thousands of cores [11].

SGD. The question remains whether such a DNN would produce worse error rates due to the low-precision LNS.

Surprisingly, despite the low precision of the arithmetic, a variety of algorithms still produce near state-of-the-art results when using LNS instead of floats. Example algorithms include visual background subtraction via GMMs, stereo depth camera algorithms, visual feature extraction and tracking, image manipulation using FFTs, and deblurring with Richardson-Lucy deconvolution [10].

Note that low-precision LNS is well-studied idea; other open implementations exist. One such implementation can be found in FloPoCo, an open-source VHDL library that can be used to implement LNS in hardware [1]. In general, the details of the LNS can vary. It can be implemented with analog or digital circuits, use more or less precision, and be implemented in any architecture, including SIMD, GPU, CPU, FPGA, and mobile [11]. The Singular chip is one instance of low-precision LNS hardware. This thesis determines whether DNNs based on low-precision arithmetic are also effective as acoustic models in speech recognition. We evaluate the arithmetic, not the hardware.

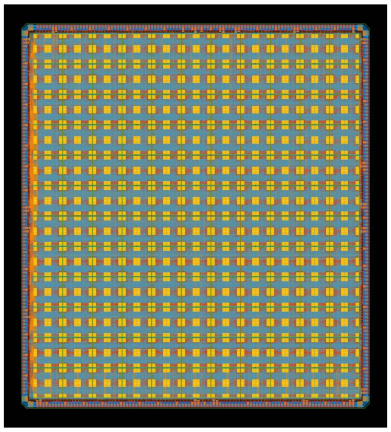


Figure 2-4: One 25mm^2 chip in the Singular prototype board from mid-2015. The board has 16 chips, each with 2,112 cores [10].

2.6 Low-Precision Arithmetic

Although there are many types of low-precision (LP) arithmetic available, we used the one found in the Singular chip. Bates provided us with a software simulation of the Singular chip's low-precision LNS arithmetic. The simulation is written in C, and we used it to implement the approximate DNN. The simulation includes: functions for converting between float and LNS; functions for multiplication, division, addition, subtraction, negation, and comparisons; and exponential and logarithm functions. LNS numbers are stored in the format shown in Listing 2.1..

```
                 |----------------- logarithm of number -----------------|
[number sign bit][sign bit][integer part (5 bits)][fraction part (6 bits)]

// As code:
struct ga_usertype01 {
  int sign; //number sign bit
  int val;  //logarithm of number, stored as fixed-position value
};
```

Listing 2.1: The LP type definition in C. The type stores the sign and logarithm of the number. The code refers to the LNS numbers as ‘ga_usertype01.’

There is also a NaN bit, which is set on overflows and errors; we omit the overflow checks in all the example code for clarity.

Converting from float to LNS involves taking the logarithm of the float and storing it as a fixed-position value and also storing the sign of the float. Multiplication and division are implemented by adding and subtracting the logarithms and adjusting the signs as needed. Multiplication is shown below:

```
ga_usertype01 usertype01_multiply(ga_usertype01 a, ga_usertype01 b) {
  ga_usertype01 result;
  result.sign = a.sign ^ b.sign;
  result.val = a.val + b.val;
  return result;
}
```

Addition and subtraction in LNS is more complicated. Consider two numbers B and C, represented as their logarithms, b and c. The result of adding B and C is represented by $log(B+C)$, which can be computed with the following formula:

$$log(B+C) = log(B*(1+C/B)) = log(B) + log(1+C/B) = b + G(c-b) \tag{2.5}$$

where $G(x)$ is the function, $log(1+2^x)$. To compute $B+C$, the addition function computes $c-b$, feeds that through G, and adds the result to b [11]. There are many ways to compute G efficiently in hardware; Singular uses a proprietary method that is simulated in the C code. Subtraction works similarly.

The LNS exponential and natural logarithm functions are computed respectively with 4th and 5th order Taylor series approximations:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} \tag{2.6}$$

$$ln(x) = ln(\frac{1+y}{1-y}) = 2(y + \frac{y^3}{3} + \frac{y^5}{5}) \tag{2.7}$$

where $y = \frac{x-1}{x+1}$. The interfaces of the C simulation functions are shown in Listing 2.2.[1] The LP operations of the simulation are about 10 times slower than their float equivalents since the simulation does not run on LP hardware.

[1] At this time, the full LP simulation is still proprietary. Those interested in using the code for research may contact: contact@singularcomputing.com.

```
// Conversion functions: float->LNS and LNS->float
ga_usertype01 float_to_usertype01(float f);
float usertype01_to_float(ga_usertype01 x);

// LNS add, subtract, multiply, and divide
ga_usertype01 usertype01_add(ga_usertype01 x, ga_usertype01 y);
ga_usertype01 usertype01_subtract(ga_usertype01 x, ga_usertype01 y);
ga_usertype01 usertype01_multiply(ga_usertype01 x, ga_usertype01 y);
ga_usertype01 usertype01_divide(ga_usertype01 x, ga_usertype01 y);

// LNS comparisons
int usertype01_eq(ga_usertype01 x, ga_usertype01 y);
int usertype01_ne(ga_usertype01 x, ga_usertype01 y);
int usertype01_lt(ga_usertype01 x, ga_usertype01 y);
int usertype01_gt(ga_usertype01 x, ga_usertype01 y);
int usertype01_le(ga_usertype01 x, ga_usertype01 y);
int usertype01_ge(ga_usertype01 x, ga_usertype01 y);

// LNS exp and natural log (using Taylor approximations)
ga_usertype01 usertype01_exp(ga_usertype01 y);
ga_usertype01 usertype01_log(ga_usertype01 y);
```

Listing 2.2: Function interfaces of the C simulation provided by Bates. There are functions for converting between LNS and float and for performing LNS arithmetic.

2.6.1 Kahan Sums

One of the negative effects of low-precision LNS arithmetic is that in-place additions, such as A += B, produce no change in A if B is much smaller than A. Since valid numbers in this system are spaced apart by a factor 1.011, the only way for the addition to change A is if B is large enough to push A to the next valid value. B must be more than about 1% of A. This effect causes problems when summing many numbers. Once the total grows large enough, the remaining numbers in the sum do not get added to the total. For example, adding 1000 ones in LNS produces a wildly inaccurate value of 179.069.

```
total = LNS(0)
for i in xrange(1000):
        total += LNS(1)
print total # 179.069397
```

Once the sum gets above 179.069, the remaining ones in the sum are too small to change the total. Note that conventional float arithmetic also has the same problem, but the problem occurs only in more extreme cases. With seven significant decimal digits, float sums fail only when $B < 10^{-7} * A$ instead of when $B < 10^{-2} * A$. Adding 1000 ones is one of the non-extreme cases; the float arithmetic produces the expected result of 1000:

```
total = float(0)
for i in xrange(1000):
        total += float(1)
print total # 1000
```

Bates found that the Kahan summation algorithm can solve the inaccuracy in linear sums of LNS numbers. The algorithm works by keeping a running compensation of the accumulated error in the total. The error is added back to the total on each iteration of the sum. Kahan is implemented by adding an extra variable to hold the compensation and by replacing the addition operation with 2 additions and 2 subtractions [21]. As shown below, Kahan produces a decent result of 991.263, which is within 1% of the correct result.

```
total = LNS(0)
compensation = LNS(0)
for i in xrange(1000):
        tmp = compensation + LNS(1)
        newtotal = total + tmp
        compensation = tmp - (newtotal - total)
        total = newtotal
print total # 991.263611
```

The Kahan sum bounds the error to $O(1)$. An alternative solution is to sum the LNS numbers pairwise. Pairwise addition uses divide-and-conquer to add pairs of numbers at a time, and it bounds the error to $O(logN)$ [21]. As shown below, pairwise summation of 1000 ones in LNS produces a better result of 1002.058.

```
def pairwise(n):
        if n == 0:
                return LNS(0)
        if n == 1:
                return LNS(1)
        mid = n/2
        return pairwise(mid) + pairwise(n-mid)

total = pairwise(1000)
print total
# 1002.057800
```

It turns out that an LNS-based DNN cannot perform as well as a float-based DNN unless Kahan or pairwise sums are used to maintain precision in certain sums.

2.7 Summary

The SGD algorithm used to train DNNs is slow, especially when the training dataset is large. Existing approaches to speed up SGD all involve significant modification to the algorithm. Since the variations of the algorithm work differently, we cannot expect them to produce the same results as SGD every time. The only way to speed up the true SGD algorithm is with faster hardware. Low-precision hardware, such as the Singular chip, is faster than existing GPUs but performs only low-precision arithmetic. This thesis determines whether a DNN can be trained using low-precision arithmetic to produce state-of-the-art results. The rest of this thesis refers to low-precision arithmetic as LP arithmetic and any type of hardware that uses it as LP hardware.

Chapter 3

Implementation of Approximate DNN

Most DNN toolkits, including the one we use as a baseline, are based on IEEE-754 float. To implement the approximate DNN, we added the low-precision (LP) type to an existing DNN toolkit. We then trained approximate DNNs as acoustic models for speech recognition and compared them to DNNs trained using IEEE-754 float. The approximate DNNs take much longer to train because they are based on the slow LP arithmetic simulation. Training time is not an issue since we only care about whether the approximate DNN will converge. Training will be faster once it implemented on LP hardware, but we only evaluate the LP arithmetic for now.

The DNN toolkit we selected for our experiments was Pdnn [32]. Pdnn is written in Python, and it uses the Theano library for all its computations. Theano is a Python library for general-purpose math [8, 13]. It allows users to define and evaluate mathematical expressions involving arrays and tensors with very few lines of high-level code. Behind the scenes, Theano compiles the mathematical expressions into C++ code, which can execute either on the CPU or the GPU. Adding the LP type to Pdnn requires adding it to the Theano library. The main advantage of using Theano is that we can quickly change and test our approximate DNN. In addition, by adding LP to Theano, we also get a LP library that can be used to implement other non-DNN algorithms.

We use the Kaldi toolkit to build our speech recognizers [39]. Kaldi also includes its own

IEEE-754 float DNN, which it uses to build acoustic models. (Kaldi actually has two DNNs; this thesis only considers the "Karel DNN" version). For our experiments, we treat the Kaldi DNN as the state-of-the-art baseline. We modify Pdnn so that it can be switched in as a replacement for the Kaldi DNN. We train acoustic models using the Kaldi DNN, the Pdnn float DNN, and the Pdnn approximate DNN and compare the word error rates achieved by the three. The rest of this chapter describes how we add LP to Pdnn and Theano.

3.1 Pdnn

Our first task was to make sure that Pdnn could achieve state-of-the-art results comparable to Kaldi DNN. The Pdnn website reports that an acoustic model built with Pdnn DNN achieves a dev set phoneme error rate (PER) of 18.8% and a test set PER of 20.2% on the TIMIT phoneme recognition task [31]. These results, which we reproduced, are worse than the 17.5% dev set PER and 18.5% test set PER reported for Kaldi DNN [39]. It would not make sense to use Pdnn for our experiments if it could not match the results of the Kaldi DNN. Therefore, we changed Pdnn to match the Kaldi DNN results.

Pdnn differs from Kaldi DNN in several ways. Pdnn performs its own feature processing and label alignment using Kaldi's scripts. It converts the features and labels into non-Kaldi storage format. Pdnn also has different weight initialization. The random initialization of Pdnn draws from slightly different distributions, and RBM-based initialization uses Pdnn's own implementation of RBM. Pdnn also trains the network using an exponentially decaying learning rate, which differs from the learning rate schedule of Kaldi DNN. Kaldi's schedule can be found in the experiment of Section 4.3.2. Finally, it performs its own posterior extraction and decoding using Kaldi's scripts.

We eliminated those differences to make Pdnn perform as well as Kaldi DNN. Pdnn now uses the same features and label alignments as the Kaldi DNN; it loads the features and labels using a Python port of the C++ Kaldi data loader. To port the Kaldi data loader, we wrote a C wrapper for the necessary C++ classes and exposed the wrapper to Python using the Ctypes (C-to-Python) library. We also wrote a model loader that could initialize Pdnn using either Kaldi-initialized DNNs or Kaldi-trained RBMs. We replaced the exponentially decaying learning rate in Pdnn with the Kaldi DNN learning rate schedule. Finally, we made sure that Pdnn extracted posteriors and decoded in the same way as Kaldi DNN. With these changes, Pdnn produces error rates that are very close to those of Kaldi DNN. Pdnn is suitable for our experiments since Pdnn and Kaldi DNN can be now be compared in an apples-to-apples comparison.

3.2 Theano

In this section, we present background on Theano and describe its architecture. Then we describe how we add LP to each component of Theano's software stack. This section assumes basic knowledge C, C++, and Python. We first introduce additional programming concepts needed for understanding Theano.

3.2.1 Background

Theano is made up of several libraries. The libraries are a mix of Python, C, C++, and Cuda code. Cuda is a variant of C++ that runs on Nvidia GPUs [37]. In Cuda, functions are declared as either "host" functions that run on the CPU, or as "device" functions that run on the GPU. Host and device functions are the same, except that host functions must access host memory and device functions must access device memory. Cuda code also has kernels, which are another type of device function. Cuda is compiled with its own compiler (nvcc). There are associated Cuda libraries, such as the Cublas library that performs linear algebra and matrix math [37].

Many of the libraries are for Python. Python modules and classes are usually written in Python, but they can also be written in C/C++. The modules and classes written in C/C++ are referred to as Python Extensions [6]. Python Extensions are compiled into shared objects that can be dynamically loaded by Python. Both Cuda and Python Extensions are used frequently in Theano.

3.2.2 Architecture

Theano allows users to build mathematical expressions involving symbolic variables in Python. For example, a Theano expression for a DNN hidden layer can simply be written as:

```
layer_output = T.sigmoid(T.dot(layer_input, weights) + biases)
```

where `layer_output`, `layer_input`, `weights`, and `biases` are Theano variables that represent arrays and vectors. The user compiles the expression, specifying which variables are inputs and which variables are outputs. The result of compilation is a callable Python function that performs the computation. When compiling an expression, Theano internally creates a computation graph, where nodes are operations and variables. Theano's optimizer then applies optimizations to the graph. The optimizations speed up the computation by simplifying the expression, reducing memory usage, reducing data transfers, and, if a GPU is available, promoting operations from the CPU to the GPU.

Finally, Theano generates C++ code for each operation. The C++ code for each operation is saved onto the filesystem and compiled into a Python Extension using the system's C++ compiler. Instead of C++, Cuda code is generated for operations that execute on the GPU; it is compiled with the Cuda compiler. Theano then returns a function that, when called, executes the entire computation graph by loading each Python Extension and executing each operation. Some operations do not have C++ or GPU equivalents yet; those instead execute directly in Python. All the compiled C++ and Cuda code is cached to avoid unnecessary re-compiling. The entire process is summarized in Figure 3-1.

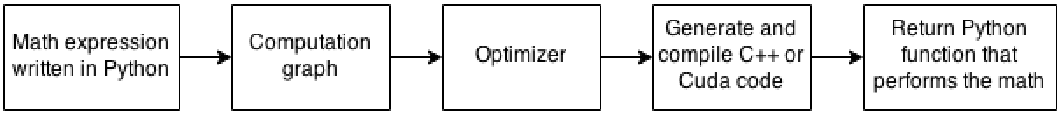

Math expression written in Python
Computation graph
Optimizer
Generate and compile C++ or Cuda code
Return Python function that performs the math

Figure 3-1: How Theano works: Theano converts the math expression into a Python function that performs the computation.

Pdnn depends on Theano. Theano generates the C++ code by itself and depends on a GPU backend to generate the Cuda code. There are two possible GPU backends. The first GPU backend, "sandbox.cuda," is included with Theano. The second GPU backend, Libgpuarray [3], is an external component. We chose to use the second GPU backend, Libgpuarray, for our approximate DNN. The GPU backend depends on the Cublas library to perform some matrix operations, such as matrix multiply. All the components, except for Cublas, depend on Numpy to manipulate arrays in Python [5]. The Theano stack is shown in Figure 3-2.

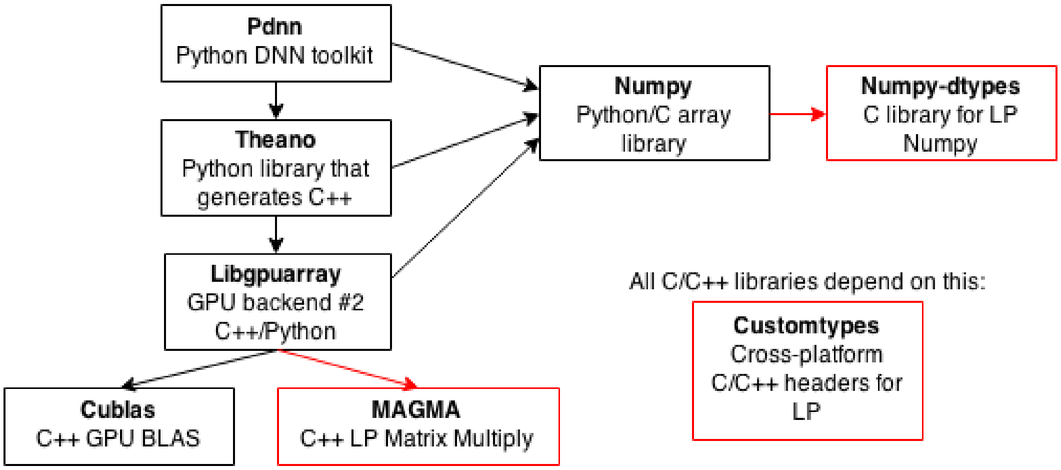


Figure 3-2: Architecture of the approximate DNN. Pdnn depends on Theano. Theano itself has many dependencies. The red components are ones that we added. LP was added to every component.

To add LP to Pdnn-Theano, we add three components to this stack. Our additions are pictured in red in Figure 3-2. The first addition is Customtypes, a library that contains the LP simulation code in one place for all the other components to use. The second addition is Numpy-dtypes [14], a library that allows us to add user-defined types to Numpy. The third addition is Magma, an open-source version of Cublas that we use to write LP matrix multiplication [4].

We added LP to the stack one component at a time, making sure to observe the dependencies so that we could test each component before moving on to the next. LP was added to components in this order: Customtypes, Numpy-dtypes, Magma, Libgpuarray, and Theano.

3.2.3 Customtypes

The C simulation code provided by Bates is needed by nearly every component. It is used by the C code in Numpy-dtypes and Libgpuarray and the Cuda code in Magma. Theano generates C++ and Cuda code that also uses it. Since all the components need it, we implemented the Customtypes library to provide a version of the simulation that works in C, C++, and Cuda.

Customtypes has a single header file with all the code for LP. We chose not to compile the LP code into a static or shared library since linking against it caused runtime errors in Cuda. Instead, the other components include the header wherever they need it. The header depends on compiler macros for cross-device and cross-language compatibility. If the C compiler is detected, the header defines the LP struct and C functions that operate on it. If the C++ compiler is detected, the header defines a LP struct with C++ methods, the C functions, and overloaded versions of built-in C++ arithmetic functions and operators. If the Cuda compiler is detected, the header behaves as if the C++ compiler was detected, but it also declares all the methods, functions, and operators to be executable on both host and device. Finally, we use Cuda macros to generate different versions of the functions when necessary: host functions access host global variables and call host-only functions, while device functions access device variables. A snippet of the header file is shown in Listing A.1 of Appendix A.

Theano normally generates C++ and Cuda code only for floats and other built-in types. That code uses built-in C++ operators (e.g. `+`, `-`, `/`, `*`) and functions (e.g. `max` and `abs`). Since we overloaded all those operators and functions for LP, most of the generated code that works for floats also works for LP. We did not need to change code generation for it to work with LP.

The LP struct has a constructor that allows for float-to-LP casts. We made sure not to define a LP-to-float cast using a user-defined conversion [7]. Omitting LP-to-float ensures that C++ uses the LP arithmetic. In C++, operations on variables of two different types will promote one of the types as necessary. For example, when adding a float and integer, the integer is promoted to a float, and float addition is performed. By overloading the built-in operators, we enabled operations between LP and float. By omitting LP-to-float casts, we ensure that those operations will promote float to LP and perform LP arithmetic instead of promoting LP to float and performing float arithmetic. If Theano generates any code that violates these conditions, the compiler will fail, and we can easily identify and fix the offending code. In addition, by leaving out LP-to-float casts, there is no way to store an LP result in a float without using a specific function that we wrote. The function is easy to search for in the source code, so it is easy to verify that the generated code performs only LP arithmetic on LP variables. These techniques guarantee that once a number has been converted to LP, it stays in LP unless explicitly converted to float. The operators and functions were overloaded in such a way that operations involving at least one LP operand use LP arithmetic and produce an LP result.

3.2.4 Numpy

The next component was Numpy, the standard library used for performing numerical operations on Python arrays. The Numpy-dtypes library provides an example of adding a user-defined type to Python and Numpy. Following that example, we added LP to Numpy-dtypes.

The LP version of Numpy-dtypes is a Python Extension written in C. To define a LP type that can be manipulated in Python, we registered the LP struct and C implementations of a class constructor, getters, setters, casts, string representation, and serialization functions using Python's C API. We also registered the LP arithmetic functions. We then added the LP type to Numpy by registering the arithmetic functions using Numpy's C API. The two APIs set a few variables in the Python extension. Theano requires access to them, so we used the Python Capsules API to export the variables. The LP struct and arithmetic functions came from the C section of the LP header in Customtypes.

Finally, we made two changes to the Numpy library itself. The first change fixed a bug related to user types. Numpy allows user-defined types to be stored in memory either as a struct or as a pointer to a struct, but part of it incorrectly assumes only the latter. The second change deals specifically with LP. The value of zero in LP is not represented in memory by a zero. However, Numpy wrongly assumes that the value of zero for all data types is represented by zero in memory, and it uses C's `calloc()` function to clear the memory when creating arrays of LP zeros. We changed Numpy so that it uses Python to assign LP zeros instead of using C to clear the memory.

With Numpy-dtypes, we can instantiate and manipulate LP values in Python or Numpy just like any built-in data type.

3.2.5 Magma

The Theano GPU backend depends on Cublas for many array operations, including matrix multiply. To implement LP matrix multiply on the GPU, we used Magma, an open-source alternative to Cublas, as a starting point. We only implemented the LP matrix multiply in Magma since the other operations are not used by Pdnn.

Magma contains matrix multiplies for float, double, single precision complex, and double precision complex numbers. Each type's matrix multiply includes: (1) data type struct declaration, (2) definition of add, subtract, multiply, and convert-from-float functions, (3) a function for fetching the data type from device memory, and (4) several kernels and header files that define the matrix multiply for that type. We copied the float matrix multiply code and replaced the appropriate definitions to change it into a LP matrix multiply. We then changed the LP matrix multiply to a Kahan matrix multiply: the dot products inside the matrix multiply use Kahan summations.

Magma is the one component of our Theano stack that does not use Customtypes directly. Because the performance of matrix multiply is critical to the speed of the DNN, we needed to optimize the LP data type specifically for matrix multiply. We copied the LP add, subtract, and multiply code into Magma and experimented with it to increase the speed of LP matrix multiply. We tried rewriting the LP functions to be as simple as possible and varying the thread block sizes.

With Kahan matrix multiply of LP, the best performance we ever achieved when multiplying two 2048x2048 matrices was 40 Gops, or billion operations per second. In contrast, Cublas float matrix multiply on the same-sized matrices achieves 3000 Gops. Despite the difference, the 100x slowdown is not unexpected, according to our benchmarks. Magma's matrix multiply, on which we based the LP version, achieves 1500 Gops, or half the speed of Cublas. If you change the float into a struct containing a float, which is similar to the LP definition, Magma's performance halves to 750 Gops. Adding Kahan further reduces performance by a factor of 2.5 to 300 Gops since the two operations in the dot products (add and multiply) turn into five operations (two adds and two subtracts from Kahan and one multiply). If you account for the fact that each of the LP simulation operations takes around 10 times as many steps, reducing the float performance to 30 Gops is reasonable, and this estimate is close to what our LP Kahan matrix multiply achieves. In practice, our approximate DNNs were 10-30 times slower than the float DNNs.

3.2.6 Libgpuarray

Theano includes two GPU backends. Theano uses the backend to generate Cuda code and to execute GPU operations. The first backend, "cuda.sandbox," is the default included with Theano. It only supported operations on 32-bit floats; selecting another type causes Theano to fall back to CPU execution. The second backend, Libgpuarray, supported GPU execution with 32-bit floats and 64-bit floats. Although other types would cause Theano to fall back to CPU execution, Libgpuarray already had partial support for other types (characters, integers, complex numbers, unsigned, and signed) that Theano was not taking advantage of. For that reason, we chose to add LP to Libgpuarray instead of "cuda.sandbox."

Libgpuarray is made of a C library and a Python library. The C library is a utility library for GPU programming; it defines functions for GPU initialization, memory management, compilation and execution of kernels passed in as strings, array manipulation, and Cublas execution. The Python library depends on the utilities in the C library; it defines a Python class for a GPU array and functions for compiling and executing kernels. Theano uses the

Python class and functions.

To add LP to Libgpuarray, we searched the code for all instances of '`complex`' or '`cfloat`' (the name of the struct for complex numbers) and added corresponding code for LP. Both the C library and the kernels generated by the Python library used the LP code from Customtypes. We changed the C library so that the matrix multiply of non-LP continues to rely on Cublas, but the LP matrix multiply uses our LP Kahan matrix multiply from Magma.

One interesting part of the implementation was in the array copy. Libgpuarray has a function that copies from one GPU array to another. The source and destination of the copy each have one of over a dozen types. If you wrote a Cuda kernel to perform each copy, there would be N^2, or over a hundred, kernels in the source code. To avoid this problem, Libgpuarray dynamically generates a kernel for each source and destination type as they are needed during runtime and uses Cuda to compile, load, and execute the kernel. To speed up the kernel compilation, Libgpuarray generates the kernel in Cuda Assembly instead of Cuda C++.

Unfortunately, assembly code for the LP type is hard to write, so we changed Libgpuarray to generate the kernel for LP array copy in Cuda C++. Compiling this kernel each time it is used turned out to be much slower since Cuda C++ takes much longer to compile than Cuda Assembly. Instead, we decided to avoid kernel generation for LP array copies. We wrote float-to-LP, LP-to-LP, and LP-to-float versions of the array copy kernel in Cuda C++ and compiled those with the rest of the source code instead of using the slower dynamic kernel generation and compilation. We did not support array copies from non-float types to LP or from LP to non-float types, otherwise we would have needed to write dozens of such kernels. However, if such copies are needed, the user can use a float array as an intermediary.

3.2.7 Theano and Pdnn

At this point, all of Theano's dependencies had LP. The final step was to add LP to Theano and to make Pdnn use it. To make Pdnn use LP, we had to make sure that all its Theano variables were created with type LP. Fortunately, Pdnn already created most its Theano variables with type ‘`theano.config.floatX`.’ ‘`theano.config.floatX`’ is a user-configurable setting in Theano that is typically set to 32-bit float and occasionally 64-bit float. By making sure all Theano variables in Pdnn were created with type ‘`theano.config.floatX`’ and then adding LP to the list of permissible `floatX` types, we made it so that Pdnn could run with either 32-bit float or LP depending on this setting. When Pdnn is configured to use LP, float is only used for feature I/O and for saving the DNN; all computations are LP.

With Pdnn using LP, the last step was to add LP to Theano. To add LP to Theano, we took the approach of running Pdnn and looking for failures caused by operations that did not support the LP type. There are too many operations in Theano, and Pdnn only uses a small subset. This approach allowed us to add LP where it was needed more quickly. Often, the failure was caused by an operation asserting that its operands were 32-bit or 64-bit float. For these operations, we simply added LP to the list of permissible types. Theano included a few tables for defining data types; we added LP to those, again finding the tables by looking for table lookup errors when running Pdnn. We also changed the C++ code generation to include our Customtypes LP header file.

Theano operations can run as Python on the CPU, C++ on the CPU, or Cuda on the GPU. Most operations are added to the computation graph as Python operations; the Theano optimizer then examines the computation graph and promotes Python operations to C++ and C++ operations to GPU operations. The optimizer had type assertions that prevented the LP operations from being promoted to either C++ or GPU. We added LP to the permissible types in those assertions so that all LP operations would be promoted to C++ or, when possible, the GPU.

Finally, we ran the LP Pdnn on the MNIST task and fixed any errors that prevented it from converging. Two of the errors dealt with overflows in LP. We changed the LP exponential function to return 0 for inputs less than -20. We also changed the LP logarithm function to return a large negative value (-10^8) for an input of 0. As a final optimization, we rewrote the Theano GPU softmax operation to use more parallelism. For our TIMIT experiments, this halved the training time when using LP. This completes the implementation of the basic approximate DNN.

3.3 Kahan Summation

With the approximate DNN completed, the next step was to add Kahan sums. Kahan sums turn out to be critical for the approximate DNN to match the float DNN. In Chapter 4, we analyze the importance of the Kahan sums. In this section, we describe how we add it to Theano and Pdnn.

In Theano and Pdnn, four of the operations that we use have linear sums: (1) Theano (Libgpuarray) matrix multiply, (2) Theano tensor sum, (3) Theano softmax, and (4) Pdnn weight updates. The tensor sum is used when summing the gradients for all the examples in a batch to get a single gradient for each weight and bias. We modified the four operations so that they use Kahan when operating on LP. Adding Kahan to the matrix multiply, tensor sum, and softmax is fairly straightforward since that mostly involves changing C++ code. Adding Kahan to the Pdnn weight updates is trickier.

Weight updates are a target for Kahan summation because they are a linear sum. During training, the gradient multiplied by the learning rate is added to each weight on each batch. There are thousands of batches per epoch and dozens of epochs. The gradients are usually much smaller than the weights that they are added to. Thus, weight updates are extremely long linear sums where very small numbers are being added to comparatively very large numbers. Weight updates are therefore an excellent candidate for Kahan sums. In Theano, the non-Kahan weight update is implemented as follows:

```
new_weights = T.add(weights, updates)
```

Kahan of the weight updates requires a compensation variable for every weight. For every weight matrix and bias vector in Pdnn, we create a separate matrix or vector of compensations. An obvious way to implement Kahan would then be to use the Theano add and subtract operations on each pair of weight and compensation arrays exactly as the Kahan algorithm is specified:

```
tmp = T.add(compensations, updates)
new_weights = T.add(weights, tmp)
new_compensations = T.sub(tmp, T.sub(new_weights, weights))
```

This method does not work; it turns out to have the same effect as the non-Kahan update. The problem is caused by an aggressive optimizer. An aggressive optimizer will optimize away the Kahan sum if it applies the associative rule and simplifies the resulting expression [19]. This is true of optimizers in certain programming language compilers and of the Theano optimizer. Fortunately, the C, C++, and Cuda compilers do not apply the optimization, so our Kahan sums in the C++ code work as expected. Unfortunately, the Theano optimizer that analyzes the computation graph does apply the rules of associativity to associative operations, such as addition. To prevent the Theano optimizer from optimizing away our Kahan sum, we copied the Theano tensor ‘add’ operation and declared that this new operation was not associative. Our new operation is named ‘`add_no_assoc`,’ and using it to implement Kahan prevented Theano from optimizing away the Kahan. The final implementation of Kahan weight updates is shown below:

```
tmp = T.add_no_assoc(compensations, updates)
new_weights = T.add_no_assoc(weights, tmp)
new_compensations = T.sub(tmp, T.sub(new_weights, weights))
```

3.4 Verification

It is critical that we verify that the approximate DNN performs all computations with LP and not float. The slow execution of the approximate DNN is one indicator that we are performing LP computations since the LP simulation is slower than floats. However, to be absolutely sure, we verified our approximate DNN in two ways.

First, we printed the Theano computation graph for the training function of the approximate DNN. The resulting image displays all the Theano operations and their input and output types. If the computations in the DNN are performed with LP, then there should be no values with floats. We found 5 values with floats, but they were acceptable since they are not involved in any arithmetic used by the training.

Our second verification method was to examine the C++ and Cuda code generated by Theano. For our approximate DNN, Theano generated 46 C++ and Cuda files; each file implements one operation. Libgpuarray also generates 5 Cuda files. We examined every source file to check that all arithmetic operations were performed with at least one LP operand and without typecasting that LP operand to float. After doing so, we know that all C++ and Cuda Theano operations used by Pdnn perform LP-only arithmetic. The only remaining place where float arithmetic might hide is in Python operations. It is harder to reason about the arithmetic performed by Python since Python is a dynamically typed language that will cast variables as needed. However, we ran Theano in ‘profile mode’ to print every operation and its method of execution. We confirmed that all operations had been promoted to either C++ or Cuda, which means no operations were running in Python. Thus, we concluded that no float arithmetic is performed in the approximate DNN.

3.5 Summary

Our final approximate DNN system is implemented in Pdnn. With one configuration change, we can run Pdnn as an approximate DNN or as a float DNN. We made sure that Pdnn could reproduce the state-of-the-art results achieved by the Kaldi DNN before converting it to an approximate DNN. To implement the approximate DNN, we added LP to the Theano math library used by Pdnn, which required adding LP to every component in the Theano software stack. Finally, we added Kahan summation to the LP versions of matrix multiply, tensor sum, softmax, and Pdnn weight updates. We verified that LP arithmetic was used in the approximate DNN instead of float arithmetic. In the next chapter, we describe experiments that compare the performance of a float DNN to that of an approximate DNN.

Chapter 4

Approximate DNN Experiments

In the this chapter, we evaluate the approximate DNN on three tasks - MNIST, TIMIT, and WSJ - to determine if it can do as well as a conventional float DNN. MNIST is a computer vision task, and TIMIT and WSJ are speech tasks. The tasks get progressively larger. MNIST, the smallest, has 50,000 training examples. TIMIT has 1,000,000 training examples. WSJ, the largest task, has 26,000,000 training examples. For all three tasks, we train approximate Pdnn and float Pdnn, using the same setup for both. We compare the classification error rates achieved by the two DNNs. For the two speech tasks, we also include the state-of-the-art Kaldi DNN results for comparison.

Although our changes to Pdnn should give float Pdnn similar performance to Kaldi DNN, we include both to give a complete comparison. Kaldi DNN still performs marginally better than float Pdnn, and therefore LP Pdnn. However, the difference is small enough that they can all be compared fairly. For speech tasks, we also compare an additional metric. With Kaldi, we build speech recognizers using the approximate Pdnn, float Pdnn, and Kaldi DNN. We compare the error rate achieved by the entire recognizer system, in addition to the DNN classification error rates. The approximate DNNs take longer to train since they use the slower LP simulation instead of faster LP hardware. We do not care about the training times because our experiments determine only whether the approximate DNN converges. All DNNs were trained on a GeForce GTX TITAN Black GPU using Cuda 6.0.

4.1 Selection of Comparison Metrics

DNNs are usually compared with two metrics. The first is the classification error, which describes the percentage of examples that the DNN is able to correctly classify. The second is the objective function that the backpropagation algorithm seeks to optimize, which in our case is cross-entropy. In our comparisons, we avoid comparing the cross-entropy of the approximate and float DNNs. The cross-entropy of the approximate DNN is computed using LP arithmetic and might be slightly higher or slightly lower than the float DNN's cross-entropy simply by virtue of the approximate computation. The cross-entropies are generally similar, but we avoid drawing any insights from them. The cross-entropies for TIMIT and WSJ are included in Appendix B.

The other metric, classification error, simply counts the number of incorrectly labelled examples. Approximate arithmetic influences how the examples are labelled, but counting the incorrectly labelled examples has nothing to do with approximate or float arithmetic. Therefore, classification error is a valid comparison metric.

Finally, classification error provides limited information about the quality of the posteriors. The two speech tasks, which use the DNN posteriors as part of a larger recognizer system, allow us to evaluate whether an approximate DNN can generate useful posteriors. The error rates of the entire recognizer system is our second comparison metric. Unless otherwise stated, the float DNNs are trained using floats and evaluated using floats, and the approximate DNNs are trained with LP and evaluated with LP. For the two speech tasks, the use of LP is limited to the DNN; once the posteriors are extracted using LP, the rest of the speech recognizer system uses floats.

It is critical that the approximate DNNs converge in around the same number of epochs as the float DNNs. If this were not true, then any speedup gained by using LP hardware might be offset by requiring more epochs of computation. Fortunately, approximate DNNs do not require additional epochs. For brevity, we exclude the epoch-by-epoch numbers, but they are included for TIMIT and WSJ in Appendix B.

4.2 MNIST

We performed four experiments on MNIST. The first experiment evaluates approximate DNNs trained with limited use of the Kahan sums. The second experiment analyzes the potential impact of leaving out the Kahan sum in the weight updates. The third experiment evaluates approximate DNNs with Kahan sums used in every possible place. The fourth experiment tests whether an approximately trained DNN can be evaluated using floats. In this section, we first explain the MNIST task followed by our results in the four experiments.

4.2.1 Data, Task, and Model

The MNIST dataset consists of isolated handwritten digits. Each 28x28 grayscale image contains one handwritten digit and is size-normalized and centered. Example images are shown in Figure 4-1. The feature vectors are 784-dimensional, corresponding to the intensities of the image's pixels, which are in the range [0, 1.0] [29]. The training set contains 50,000 examples, and the validation set has 10,000 examples. The task is to classify each image as one of ten digits, 0-9.

Figure 4-1: The MNIST task is to identify the digit in each image.

For MNIST, the weights of the DNN's hidden layer are randomly initialized with a uniform distribution between $\pm 4 * \sqrt{\frac{6}{fan\_in + fan\_out}}$, where fan_in is the dimension of the previous layer, in this case the input, and fan_out is the dimension of the hidden layer. This uniform distribution is suggested by [12] and included in the original Pdnn distribution. The biases of the hidden layer are initialized to 0. The weights and biases of the softmax layer are both initialized to 0. The training and validation set are normalized to zero mean and unit variance. During training, we shuffle the training set after every epoch and use a batch size of 20.

MNIST is a tiny task that is easy for the DNN to perform well on. Almost any DNN setup will converge to a classification error rate of a few percent. Our DNNs all have one sigmoid hidden layer of 512 units and one softmax layer of 10 units. When using floats, this setup achieves an error near 2%. If we tried other setups, we could push it lower. For the MNIST task, we choose not to pursue the state-of-the-art since our default setup already performs well. We compare the approximate DNN to the float DNN under this setup. In this respect, our MNIST experiments are more a proof-of-concept for the approximate DNN rather than state-of-the-art comparisons. We perform true state-of-the-art comparisons in our experiments with the two larger speech tasks.

4.2.2 Initial Attempt with Limited Use of Kahan Sums

Our first experiment tests whether an approximate DNN can match a float DNN with limited use of Kahan sums. We trained four DNNs: one float and three approximate. The approximate DNNs use Kahan only in the matrix multiply; Kahan was kept out of the tensor sum, softmax, and weight updates. We included the Kahan in the matrix multiply because our unit test for Magma revealed that the result of LP matrix multiply differs vastly without it.

The first approximate DNN (DNN #2) only uses LP arithmetic in the matrix multiply. Everything is computed and stored as floats, but values are converted to LP before the matrix multiply and converted back to floats after the matrix multiply. The other approximate DNNs (#3 and #4) use LP arithmetic everywhere. The DNNs were trained as follows:

Initial learning rate: 0.1 per batch, except for the last DNN which uses 0.5

Momentum: 0.5 after epoch 1

Fixed learning rate for 8 epochs

After 8 epochs, start halving the rate on every epoch once the cross-entropy relative improvement falls below 0.001

Load the network with the lowest cross-entropy at each epoch

Train for 15 epochs

DNN #		Classification error on Validation
1	Float, rate=0.1	2.14%
2	LP in matrix multiply, rate=0.1	2.12%
3	LP everywhere, rate=0.1	3.62%
4	LP everywhere, rate=0.5	2.15%

Table 4.1: MNIST comparison of float DNN and approximate DNN. Kahan sums are used only in the matrix multiply of the approximate DNNs.

Figure 4.1 shows the classification errors for the four DNNs on MNIST. The baseline, float DNN, achieves a error rate close to 2%. The LP arithmetic in the matrix multiply does not cause a difference in the error, as DNN #2 achieves the same error rate. However, the LP arithmetic in other parts of the computation causes the error to worsen from 2% to 3%, as shown with DNN #3.

Interestingly, if the approximate DNN is trained with a larger learning rate of 0.5, that difference disappears, and DNN #4 is able to match the baseline float DNN. We suspect that DNN #4 converged while DNN #3 did not because the larger learning rate in DNN #4 preserved some precision in the weight update. Recall that both DNN #3 and #4 do not use Kahan in the LP weight updates. Without Kahan, it is impossible to add small LP numbers to much larger LP numbers. This means that the smaller weight updates will be ignored since they cannot be added to the much larger weights. Ignoring the smaller weight updates worsens the performance of DNN #3. DNN #4 performs better possibly due to its larger learning rate. Since weight updates are proportional to the learning rate, it is possible that the larger learning rate makes the tiny updates large enough to be added to the weights under LP arithmetic. The larger learning rate acts like a substitute for Kahan in that it prevents weight updates from being ignored. However, using a much larger learning rate is not an attractive option for training and probably does not work for most tasks. In the next experiment, we determine how important the small updates are to the convergence of the approximate DNN.

4.2.3 Small Weight Updates

In earlier work, we trained a DNN, generated a histogram of the weight updates, and determined that weight updates are usually much smaller than the weights. Unfortunately, LP is incapable of adding a small number to a large number if the small number is less than one percent of the larger number. This experiment determines whether the small weight updates matter, if they do in fact matter.

We changed our DNN to ignore small weight updates. For each weight update, we compute the relative change that it would cause to the weight. If the relative change is below some threshold, we ignore it. We implement this feature in Pdnn using Theano by multiplying the updates by a mask of 1s and 0s, where a 1 indicates that the relative change of the update is larger than our threshold. This is implemented as shown:

```
mask = T.abs_(update / weights) > weight_update_threshold
update = update * mask
```

The threshold is adjustable. A threshold of 0.01, for example, would ignore all updates that are less than one percent of the weight being updated and would simulate the effect of LP addition ignoring small weight updates. We trained seven float DNNs, varying the threshold for ignoring weight updates. We used float DNNs instead of approximate DNNs to isolate just the effect of ignoring small weight updates. The DNNs were trained as follows:

Initial learning rate: 0.1 per batch

Momentum: 0.5 after epoch 1

Fixed learning rate for 15 epochs

Load the network with the lowest cross-entropy at each epoch

Train for 15 epochs

The results on MNIST are shown in Figure 4-2. The DNN's accuracy degrades as the threshold for ignoring weights is increased. Even at a threshold of 0.001, the accuracy has already degraded significantly. LP addition ignores updates that cause less than one percent change, which falls around the threshold of 0.01. This experiment demonstrates that small

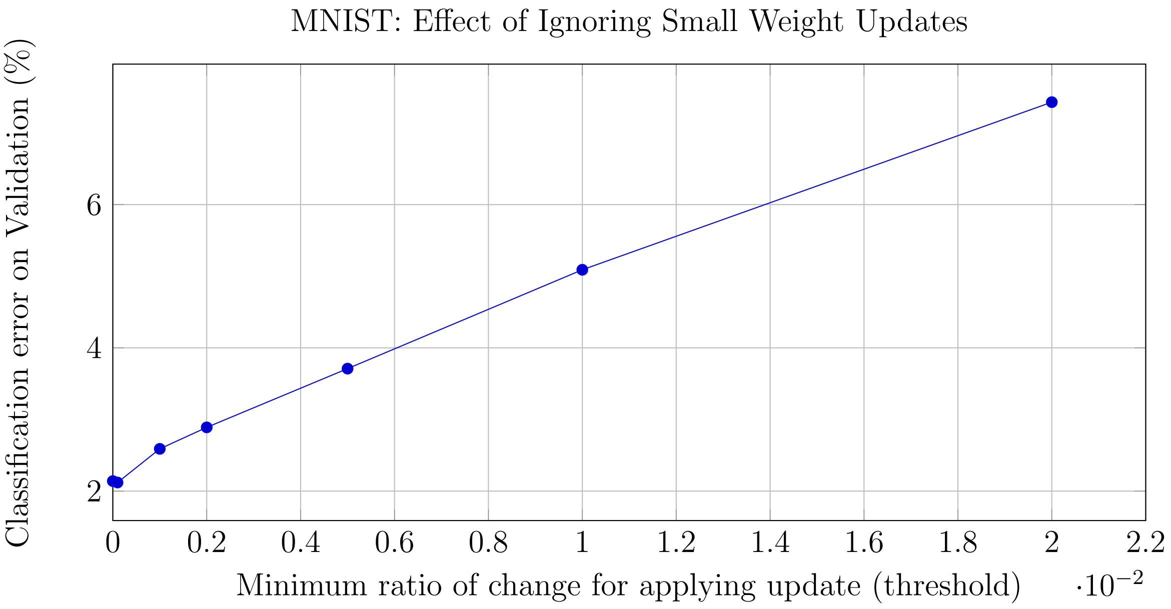


Figure 4-2: MNIST experiment to determine whether small updates to the weights can be ignored in a float DNN. Weights updates are ignored if the absolute value of the ratio of change is below the threshold.

weight updates must not be ignored, unless the ratio of change is around 0.0001, or one-hundredth of one percent. At the very minimum, an approximate DNN must use some method, such as Kahan, to maintain precision when updating weights using LP addition. Since we performed this experiment with float DNNs, we know this to be true even if LP arithmetic is limited only to the weight update.

4.2.4 Kahan Sums

With the insights of the previous experiment, we added the Kahan version of the LP weight updates to Pdnn. We also added Kahan to the LP versions of tensor sum and softmax. Matrix multiply already had Kahan. Kahan is necessary in the LP weight update, but its importance is questionable in the other three areas. In Section 4.3.4, we analyze which of these three areas requires Kahan. For this experiment, we used Kahan everywhere that we could. We retrained our approximate DNN using the same setup from Section 4.2.2. As shown in Table 4.2, the approximate DNN nearly matches the results of the float DNN, without resorting to using a large learning rate as before. The frame errors are similar.

	Classification error on Validation
Float, rate=0.1 (from table 4.1)	2.14%
LP everywhere + Kahan sum, rate=0.1	2.29%

Table 4.2: MNIST comparison of float DNN and approximate DNN. Kahan sums are used everywhere in the approximate DNN.

4.2.5 Mixing Training and Evaluation Types

The last experiment we performed on MNIST determined whether an approximate DNN trained using LP could be evaluated using floats and whether an exact DNN trained using floats could be evaluated with LP. We took the two DNNs trained from the previous section and evaluated them using both float and LP arithmetic. The results are shown in Table 4.3.

	Float evaluation - classification error on Validation	**LP evaluation - classification error on Validation**
Float-trained DNN	2.14%	2.11%
LP-trained DNN	2.34%	2.29%

Table 4.3: MNIST experiment that shows that a DNN can successfully be trained with one datatype and evaluated with another.

According to the results, it is fine to train with one type of arithmetic and evaluate with another type. The classification error is nearly the same regardless of arithmetic. Thus, a DNN trained on the LP hardware will work on traditional float-based hardware, and vice-versa. There is no need to constrain oneself to a particular hardware platform depending on whether the DNN is approximate or float. This result means that approximate DNNs can be used in heterogeneous computing environments or in mobile hardware that might not support LP.

4.3 TIMIT

We performed four experiments on TIMIT. The first experiment compares the approximate DNN to the float DNN using Kahan sums everywhere on a smaller-sized DNN. The second experiment repeats the small weight updates experiment from MNIST. The third experiment determines where Kahan sums are actually needed. The fourth experiment compares the approximate DNN to the float DNN on the full-sized DNN. In this section, we first explain the TIMIT task followed by our results in the four experiments.

4.3.1 Data, Task, and Model

The TIMIT corpus consists of 6,300 sentences (5.4 hours) spoken by 630 speakers of 8 major dialects of American English [30]. The training set contains 3,696 sentences from 462 speakers. The dev set contains 400 sentences from 50 speakers. We use the Core Test Set, which contains 192 sentences from 24 speakers, as our test set [39]. The TIMIT phoneme recognition task converts speech audio into a sequence of phonemes. We use the Kaldi TIMIT 's5' recipe (r4762) to build the phoneme recognizer; we do not modify the recipe in any way. The recognizer includes a bigram phoneme language model created from the training set. The 61 phonemes are mapped into 48 phonemes for training and 39 phonemes for testing. The acoustic model is a HMM-DNN hybrid. The features for our DNN are "MFCC-LDA-MLLT-fMLLR with CMN" features [2]. Each feature vector consists of eleven 40-dimensional frames stacked together to provide context; thus the inputs to the DNN are 440-dimensional. We use the 'tri3' HMM-GMM model of the s5 recipe to force align the data to generate a label for each feature vector. There are 1,943 class labels [39]. Our experiments compare the Kaldi float DNN included in the s5 recipe to the Pdnn approximate DNN and Pdnn float DNN. When training the DNNs, we use 90% of the training set's sentences (1,009,937 frames) as training data and the remaining 10% (114,886 frames) as a validation set. We treat DNN initialization as a black box and simply load initialized Kaldi DNNs into Pdnn. The Kaldi DNNs themselves are either initialized randomly or using stacked RBMs. The training and validation set are normalized to zero mean and unit variance. During training, we shuffle the training set after every epoch and use a batch size of 256. We compare two types of metrics: the frame classification error on the validation set and the phoneme error rates (PERs) on the dev and test sets.

4.3.2 Smaller DNN

The Kaldi s5 recipe, which we use as the state-of-the-art baseline, uses a pre-trained 6x1024 DNN (6 sigmoid hidden layers of 1024 units and a softmax layer). Only our final TIMIT experiment uses that configuration; the following experiments use smaller DNNs. For this experiment, we determine whether an approximate DNN with Kahan sums in all four places can match a float DNN. We trained a Kaldi DNN, Pdnn float DNN, and Pdnn approximate DNN. We also train a fourth DNN: a Pdnn approximate DNN that uses Kahan everywhere except in the weight updates. All are 6x512 DNNs with pre-training. We initialized using six stacked RBMs, trained using Kaldi. The DNNs are trained using the following learning rate schedule. We use this schedule for all TIMIT experiments:

Initial learning rate: 2.0 per batch (0.008 per frame)
Momentum: 0
Start halving the rate on every epoch once the cross-entropy relative improvement falls below 0.005
Load the network with the lowest cross-entropy at each epoch
Train for 20 epochs; stop early if the cross-entropy is not improving

The float DNNs took 1 minute per epoch to train, and the approximate DNNs took 10 minutes per epoch. The approximate DNN is supposed to take longer to train since it runs on the LP simulation instead of the faster LP hardware. We are evaluating only the convergence of the approximate DNN, not its speed.

The results of the DNNs are shown in Table 4.4. The results of the approximate DNN with Kahan everywhere are very close to those of both float DNNs. That approximate DNN

	Validation frame error	**Dev PER**	**Test PER**
Kaldi: float	42.08%	18.0	19.1
Pdnn: float	42.19%	17.9	19.3
Pdnn: LP	42.33%	18.0	19.3
Pdnn: LP without Kahan weight updates	45.75%	19.3	21.0

Table 4.4: TIMIT comparison of smaller 6x512 float DNN and approximate DNN. The first approximate DNN "Pdnn: LP" uses Kahan sums everywhere. The second approximate DNN "Pdnn: LP without Kahan weight updates" leaves out the Kahan sum in the weight updates.

worked nearly as well as the float DNN. The fourth DNN, the approximate DNN without Kahan weight updates, shows once again that the Kahan sum is needed when performing LP weight updates.

4.3.3 Small Weight Updates

We repeated the small weight updates experiment from MNIST (Section 4.2.3) to confirm that small weight updates still matter in TIMIT.

We trained several 6X512 float DNNs. They are randomly initialized using Kaldi and trained using the same TIMIT scheduler as before. Each DNN ignores weight updates that fall below a threshold. We train several DNNs with different thresholds to find a threshold that will not harm the results. We use float DNNs instead of approximate DNNs to isolate just the effect of ignoring small weight updates. Results are shown in Figure 4-3.

The results demonstrate that small weight updates are extremely important. The frame error is even more sensitive to the small weight updates in TIMIT than in MNIST. Even when the threshold is 0.0001, the TIMIT frame error rate increases by almost 1%, whereas the MNIST frame error did not worsen much at that threshold.

LP addition in the weight updates will ignore updates below a threshold of 0.01, but even the smallest threshold would be detrimental, according to this experiment. Therefore, naive summation absolutely will not work when updating weights in LP. Some method must be used to preserve the weight update precision; we choose to use Kahan sums. Using

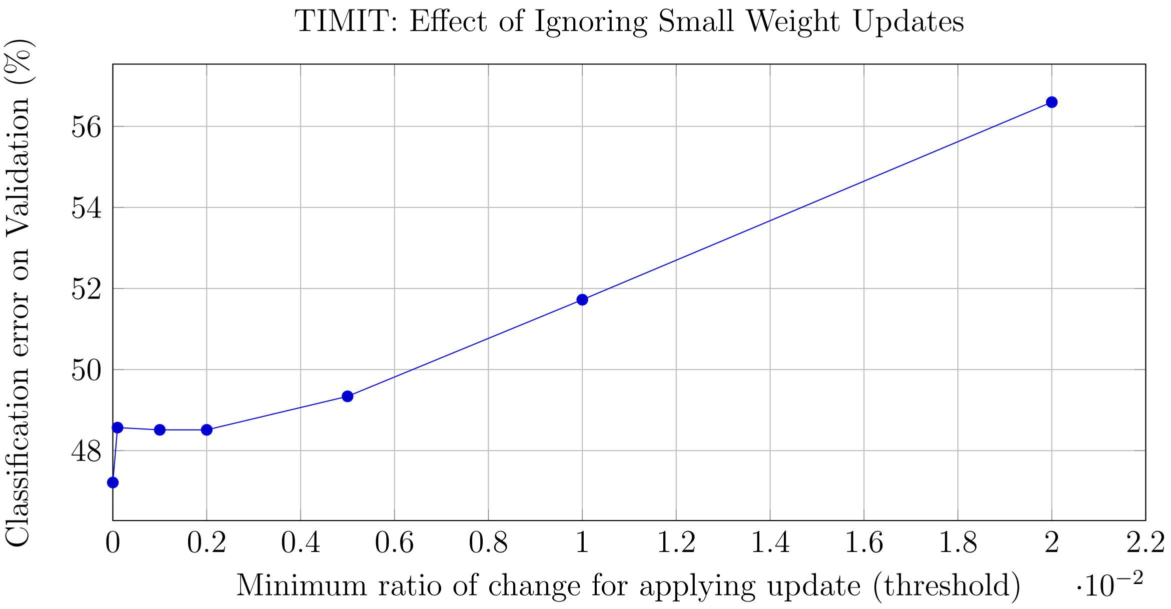


Figure 4-3: TIMIT experiment to determine whether small updates to the weights can be ignored in a 6x512 float DNN. Weights updates are ignored if the absolute value of the ratio of change is below the threshold.

Kahan sums is similar to the technique found in [40], which uses error feedback to maintain precision when quantizing gradients.

4.3.4 Where Kahan Sums are Needed

Kahan sums are used in four parts of the DNN: (1) weight updates, (2) matrix multiply, (3) softmax, and (4) tensor sum (which sums the gradients). The small weight updates experiment (Section 4.3.3) and the approximate DNN without Kahan weight updates (Section 4.3.2) show that, at the very least, the Kahan sum or some other method for maintaing precision is required in the weight update. In this experiment, we determine whether Kahan sum is needed in the other three parts where it is used.

We trained eight approximate DNNs using the same setup as before: 6x512 and initializing with Kaldi pretraining. We varied the types of addition used in the matrix multiply, softmax, and tensor sum. The three types of addition we used were Naive, Kahan, and

Pairwise. Naive addition is a simple linear sum. Pairwise addition uses divide-and-conquer addition to add pairs of numbers at a time. All the DNNs use Kahan in their weight updates.

We tested Naive and Kahan with matrix multiply. To implement this, we added an environment variable to Magma that allowed us to toggle between the two types addition for the dot products. We tested Naive, Kahan, and Pairwise with the softmax and tensor sum. We added configuration options to Theano that would allow us to choose between the types of addition used during runtime and changed the softmax and tensor sum operations to generate the appropriate code for the type of addition selected. The results are shown in Table 4.5; we include our Pdnn float DNN results as a baseline.

Experiment	**Matrix multiply**	**Softmax**	**Tensor sum**	**Validation frame error**	**Dev PER**	**Test PER**
Pdnn: float baseline	—	—	—	42.19%	17.9	19.3
LP 1	Naive	Naive	Naive	61.17%	21.1	23.4
LP 2	Naive	Kahan	Kahan	43.90%	17.9	19.8
LP 3	Kahan	Naive	Naive	60.02%	23.4	24.5
LP 4	Kahan	Kahan	Naive	42.25%	17.9	19.4
LP 5	Kahan	Naive	Kahan	57.93%	21.7	22.9
LP 6	Kahan	Pairwise	Pairwise	42.06%	17.9	19.2
LP 7	Kahan	Kahan	Kahan	42.33%	18.0	19.3

Table 4.5: TIMIT experiment to determine where Kahan or pairwise sums are needed in an approximate 6x512 DNN. DNNs with good results are highlighted in green.

The approximate DNNs that came close to the float DNN's results all used a non-Naive LP addition in matrix multiply and softmax. These DNNs are highlighted in green in Table 4.5. The worst numbers resulted from approximate DNNs that used Naive LP addition in the softmax. Using Naive LP matrix multiply and non-Naive versions for the other two operations resulted in an in-between result. We conclude that approximate DNNs are most sensitive to approximation in the softmax. However, an approximate DNN needs non-Naive additions in all three areas in order to match state-of-the-art results.

Pairwise sums guarantee less precision than Kahan sums. However, using pairwise sums in "LP 6" provided the lowest error rates. Using pairwise sums in the softmax and tensor sum is particularly attractive because existing GPU code usually implements those operations using pairwise sums.

Kahan sums incur extra computation and sometimes extra memory usage. Our results indicate that Kahan sums, or pairwise sums if they are available, are needed to replicate state-of-the-art results and are therefore not a waste of computation power. For the rest of the TIMIT and WSJ experiments, we train approximate DNNs using the “LP 6” setup: Kahan LP weight updates, Kahan matrix multiply, pairwise softmax, and pairwise tensor sum. Pairwise sums might not suffice for other tasks, but we found that they work for TIMIT and WSJ.

4.3.5 Full DNN

Our final TIMIT experiment compares the full-sized approximate DNN to the full-sized float DNNs. We trained a Kaldi DNN, Pdnn float DNN, and Pdnn approximate DNN. All three DNNs are 6x1024 and are initialized with six stacked RBMs pre-trained with Kaldi. Everything was done according to the original Kaldi TIMIT s5 recipe's setup for the Kaldi DNN. Results are shown in Table 4.6.

	Validation frame error	**Dev PER**	**Test PER**
Kaldi: float	40.66%	17.5	18.6
Pdnn: float	40.66%	17.6	18.6
Pdnn: LP	40.52%	17.8	18.7

Table 4.6: TIMIT comparison of full-sized 6x1024 float DNN and approximate DNN.

As shown in Table 4.6, all three DNNs produce similar frame errors and PERs. We ran the Matched Pairs Sentence-Segment Word Error (MAPSSWE) Test [35], which is a PER/WER significance test from the NIST Speech Recognition Scoring Toolkit [36]. We ran the test between “Kaldi: float” and “Pdnn: LP”. The p-value of 0.674 on the test set is above the significance threshold of 0.05, so there is no statistically significant difference between the two DNNs. We conclude that the approximate DNNs and float DNNs produce similar PERs on TIMIT.

4.4 WSJ

We performed two experiments on WSJ. The first experiment compares the approximate DNN to the float DNN on a smaller-sized DNN. The second experiment compares the two with full-sized DNNs. In this section, we first explain the WSJ task followed by our results in the two experiments.

4.4.1 Data, Task, and Model

The WSJ corpus consists of consists of read speech and text from the Wall Street Journal newspaper. Speech was recorded from many speakers reading subsets of the text. We use the si284 training set with 81 hours of speech. We also use Dev93 as the dev set and Eval92 as the test set [39]. The WSJ speech recognition task converts speech audio into a sequence of words.

The WSJ experiments are similar to the TIMIT experiments. We use the Kaldi WSJ 's5' recipe (r4762) to build the speech recognizer; we do not modify the recipe in any way. The acoustic model is a HMM-DNN hybrid. The features for our DNN are "MFCC-LDA-MLLT-fMLLR with CMN" features [2]. Each feature vector consists of eleven 40-dimensional frames stacked together to provide context; thus the inputs to the DNN are 440-dimensional. We use the 'tri4b' HMM-GMM model of the s5 recipe to force align the data to generate a label for each feature vector. There are 3,405 class labels [39].

Our experiments compare the Kaldi float DNN included in the s5 recipe to the Pdnn approximate DNN and Pdnn float DNN. When training the DNNs, we use 90% of the training set's utterances (26,469,621 frames) as training data and the remaining 10% (2,677,486 frames) as a validation set. All the DNNs are trained using the same initialization methods, learning rate schedule, and scripts used for TIMIT (Section 4.3). We compare two types of metrics: the frame classification error on the validation set and the word error rates (WERs) on the dev and test sets.

4.4.2 Smaller DNN

Compared to TIMIT, WSJ is a huge task. There are 25 times more frames of training data, and the default WSJ Kaldi DNN specified by the s5 recipe has 6 hidden layers of 2048 units. Each layer is twice as large as that of TIMIT. The default Kaldi DNN for WSJ takes 50 minutes per epoch to run. The full-sized approximate DNN would take 10-30 times longer since the LP simulation is slow; this equates to 1.5 days per epoch. To determine if the full-sized network would work at all, we first experimented with smaller networks and less data. In this section, we present those results. In the next section, we present the results of training full-sized networks.

First, we determined how small a network and how much less data we could use to get decent results on WSJ. The full network and full training set is not needed to achieve good results. We trained several Kaldi float DNNs of different sizes and with different amounts of training data. Results are shown in Table 4.7.

Experiment	**# parameters**	**Time/epoch**	**Dev93 WER**	**Eval92 WER**
6x2048: baseline	28,861,773	54 minutes	6.80	3.92
6x1024	9,189,709	25 minutes	6.84	4.04
6x512	3,285,837	15 minutes	7.29	4.52
7x1024	10,239,309	30 minutes	6.74	3.97
7x1024: 50% data	10,239,309	15 minutes	7.24	4.25
7x512	3,548,493	18 minutes	7.09	4.38
7x512: no pretrain	3,548,493	18 minutes	7.27	4.47
7x512: 50% data	3,548,493	9 minutes	7.54	4.64
7x512: 25% data	3,548,493	5 minutes	7.82	5.10
7x256	1,382,733	16 minutes	7.77	5.19
7x256: 50% data	1,382,733	8 minutes	8.42	5.21

Table 4.7: Experiment to determine how small a DNN and how much less data we could use to get decent results on WSJ. The DNNs are all float-based. Bad results are highlighted in gray. We used the "7x512: 50% data" (yellow) for our small-scale approximate DNN comparison.

When selecting our small-scale DNN setup, we ignored any setups that differed from the baseline by more than 1% in WER. Those ignored setups are highlighted in gray in Table 4.7. Of the remaining setups, we chose the fastest one for our approximate DNN comparisons. We selected the 7x512 DNN with 50% training data, initialized using stacked RBMs (highlighted in yellow).

It remains an open question whether RBMs can be trained using approximate arithmetic. Fortunately, pre-training is only important in tasks with less training data, such as TIMIT. In tasks with more data, such as WSJ, pre-training improves the results by very little. To demonstrate this fact, we include the results for a 7x512 DNN trained on 100% data without pretraining ("7x512: no pretrain") in Table 4.7. Its WERs differ by 0.18% and 0.09% compared to the DNN with pretraining ("7X512"). The approximate hardware will mostly be used to speed up tasks with large training sizes, such as WSJ, which do not need pre-training. Tasks with small training sizes, which do require pre-training, are fast enough as is. Therefore, it it is fine if RBMs cannot be trained using approximate arithmetic.

We still perform our approximate DNN comparisons using float pretraining since the pretraining is used only for DNN initialization, and we are only evaluating approximate DNN training. Using the 7x512 DNN and 50% training data setup, we trained a Kaldi DNN, Pdnn float DNN, and Pdnn approximate DNN. We use the same learning rate schedule as in TIMIT. The results are shown in Table 4.8. The approximate DNN performs as well as the float DNNs. All the frame errors and WERs are close.

	Validation frame error	**Dev93 WER**	**Eval92 WER**
Kaldi: float	37.54%	7.54	4.64
Pdnn: float	37.55%	7.30	4.73
Pdnn: LP	37.96%	7.38	4.78

Table 4.8: WSJ comparison of smaller 7x512 float DNN and approximate DNN using 50% training data.

4.4.3 Full DNN

With the small-scale approximate DNN converged, we repeated the small-scale experiments with the full-sized 6x2048 DNN using 100% data and Kaldi pre-training. The approximate DNN took 21 days to train.

	Validation frame error	**Dev93 WER**	**Eval92 WER**
Kaldi: float	33.12%	6.80	3.92
Pdnn: float	33.21%	6.84	4.08
Pdnn: LP	33.56%	6.95 (LP extract) 6.98 (Float extract)	3.95 (LP extract) 3.99 (Float extract)

Table 4.9: WSJ comparison of full-sized 6x2048 float DNN and approximate DNN using 100% training data.

As shown in Table 4.9, the frame errors and WERs are all close. We ran the MAPSSWE significance test between "Kaldi: float" and "Pdnn: LP"; the p-values are 0.246 on Dev93 and 0.749 on Eval92. The p-values are above the significance threshold of 0.05, which means there is no statistically significant difference between the two DNNs. Therefore, the approximate DNN does indeed match the results of a float DNN on the WSJ task.

Finally, we tested the mixing of training and evaluation types, similar to what we did for MNIST in Section 4.2.5. The purpose of this experiment was to determine whether a DNN trained using LP arithmetic can be evaluated using float arithmetic. The ability to mix types is important in heterogeneous computing environments. For instance, one might train a DNN on LP hardware and then use the DNN on float hardware. The types can also be mixed the other way; one might want to train on float hardware and deploy to mobile LP hardware. We tested the ability to mix types for both frame classification and posterior extraction.

To test frame classification, we took the DNN trained using LP arithmetic ("Pdnn: LP"), converted its weights from LP to float, and classified the validation set using float arithmetic. The float arithmetic resulted in a frame error rate of 33.48%, which is close to the 33.56% produced by the approximate evaluation. To test posterior extraction, we extracted the posteriors of the approximate DNN using both LP and float arithmetic. We used both posteriors for speech recognition; the WERs are shown in Table 4.9 as either "LP extract" or "Float extract". Posteriors extracted using LP or float both produced similar results. We ran the MAPSSWE test between the "LP extract" and "Float extract" results; the p-values are 0.674 on Dev93 and 0.317 on Eval92, which are above the significance threshold of 0.05.

There is no statistically significant difference between using LP or float arithmetic to extract the posteriors. From these experiments, we conclude that a DNN can be trained on LP hardware and used by float hardware afterwards.

4.5 Even Less Precision

The previous experiments proved that DNNs work perfectly fine even when using low-precision arithmetic that results in 1% error. The following experiment determines whether the DNN can tolerate even less precision in the arithmetic.

Recall from Section 2.5 that the low-precision LNS that we used has 6 bits in the fractional portion. Adjacent values in the LNS are spaced apart by a multiplicative factor of $\sqrt[64]{2} = 1.011$. Numbers are therefore represented with an error of 1% from their true values. Generalizing, if a LNS number has n bits in the fractional portion, the adjacent numbers are spaced apart by a factor of $2^{(1/2^n)}$, and numbers can be represented with a relative error of $2^{(1/2^n)} - 1$. The precision of LNS can be decreased by using smaller values of n. Table 4.10 shows the lower-precision versions of the arithmetic. The error increases as fewer bits are used.

n	**Arithmetic error**	**Value of** $98 + 2$	**Value of** $10 * 10$
6 (current)	$2^{(1/64)} - 1 \approx 1\%$	99.78	100.86
5	$2^{(1/32)} - 1 \approx 2\%$	98.70	98.70
4	$2^{(1/16)} - 1 \approx 4\%$	94.52	98.70
3	$2^{(1/8)} - 1 \approx 9\%$	90.51	90.51
2	$2^{(1/4)} - 1 \approx 19\%$	90.51	90.51
1	$2^{(1/2)} - 1 \approx 41\%$	90.51	64.00
0	$2^{(1/1)} - 1 = 100\%$	64.00	64.00

Table 4.10: Even lower-precision versions of LNS arithmetic. n is the number of bits in the fractional portion. Example computations are shown for $98 + 2$ and $10 * 10$.

The LP simulation code only supported $n = 6$. To implement arithmetic for other values of n, we changed the simulation to clear the bottom $6 - n$ bits after each arithmetic operation. We then trained approximate DNNs for the MNIST task using different values of n to determine if they can use less precision than 1% error. To maintain precision, the approximate DNNs use Kahan sums everywhere possible. We trained the DNNs with the same setup from Section 4.2.2. Results are shown in Table 4.11

n	**Arithmetic error**	**Classification error on Validation**
6	1%	2.29%
5	2%	3.76%
4	4%	9.73%

Table 4.11: MNIST results for approximate DNNs that use even less precision.

In Table 4.11, $n = 6$ is the precision that we used in all the other experiments. We know that $n = 6$ works nearly as well as conventional floats. At $n = 5$, there is already significant degradation in the results. At $n = 4$, the DNN no longer converges. We did not try lower values of n since they would do worse. We also tried training $n = 4$ and $n = 5$ using exact calculation of the LP logarithm and exponential functions, instead of the Taylor series approximations, because the inaccuracy in those functions might have caused the degradation. Doing so did not improve the results.

From this experiment, we conclude that approximate DNNs can only tolerate about 1% error in the arithmetic. Even at 2% error, there is significant degradation. At 4% error, the DNN simply does not work. Perhaps with other techniques, the lower precision versions of the arithmetic can work. However, with the current techniques, only 1% error is tolerable.

4.6 Discussion

We evaluated the approximate DNN on the MNIST, TIMIT, and WSJ tasks. An approximate DNN can match a float DNN in frame classification error. An approximate DNN can also function as well as a float DNN as part of a larger system; using it as the acoustic model in a speech recognizer generates similar recognition error rates. A DNN trained with LP arithmetic can be evaluated with float-based arithmetic, and vice-versa. A DNN using LP arithmetic needs a way of maintaining precision in the weight updates; we use Kahan sums. It also needs a way to maintain precision in the matrix multiply, tensor sum, and softmax;

the softmax is the most sensitive to a lack of precision. Kahan or pairwise sums can be used to address the precision issue.

We have proven that a DNN using LP arithmetic can achieve near state-of-the-art results. The LP simulation of the arithmetic is slow, taking 21 days to train one of our DNNs. However, a hardware platform that performs native LP arithmetic, such as the one proposed by Bates, can lead to faster training.

The Singular chip is 50 times faster per watt, but how much faster will it make DNN training? That depends heavily on the computational cost of Kahan. Each Kahan sum requires 4 times more operations than a normal sum. In the worst case, Kahan would slow down SGD by a factor of 4. However, the worst case is unlikely. SGD has other operations, not just linear sums. Those other operations will amortize the cost of using Kahan in the linear sums. For example, matrix multiply has one multiplication operation for each addition operation. If the additions are converted to Kahan, then the matrix multiply cost goes up only by 2.5 times, instead of 4 times. Amortization helps some parts of SGD. Other parts of SGD might avoid the Kahan cost using pairwise summation or some other alternative. In all likelihood, the computational cost of Kahan is some epsilon less than 4. Unfortunately, we cannot determine the exact speedup until we implement SGD on a Singular chip. However, we have shown that the low-precision arithmetic will work with SGD.

Chapter 5

Conclusion

In this thesis, we evaluated the use of low-precision arithmetic in DNNs for automatic speech recognition. We implemented an approximate DNN that uses the low-precision arithmetic, and evaluated it on the TIMIT and WSJ tasks. We trained acoustic models based on approximate DNNs and found that they work just as well as acoustic models based on conventional float DNNs. Both type of DNNs generate similar error rates. In addition, it is possible to train a DNN using one type of arithmetic and to evaluate it with another type of arithmetic. This means that approximate DNNs can be converted to float DNNs, and vice-versa. Finally, we found that that the approximate DNNs require Kahan or pairwise summations in order to perform as well as the float DNNs. The approximate DNNs are particularly sensitive to the lower precision in the weight update and softmax operations. All-in-all, the low-precision arithmetic works, and the approximate DNNs were able to converge.

DNN research has always been tied to advances in hardware. For example, DNNs have been around since the 1990s, but they were not popular then. They only became popular recently due to the widespread use of the GPU. The GPU, which is 10-100 times faster than a CPU, sped up the training process enough to make it practical. The GPU enabled DNN researchers to perform more experiments and to use larger datasets than before. Now, DNNs are one of the most popular techniques in machine learning.

Low-precision hardware, such as the Singular chip, is even faster than the GPU. This new type of hardware could potentially unleash another wave of DNN research, just as the GPU did. This thesis is a step toward running DNNs on the new hardware and achieving another huge boost in training speed.

There are several areas for future research. Since approximate DNNs have been proven to work, the next step is to implement one directly on faster low-precision hardware, such as the Singular chip. Future work could also focus on finding alternatives to the Kahan summations that would cause less computational overhead. Once the DNN is running on low-precision hardware, one could also try the parallelized variants of SGD on the low-precision hardware for further speedup. Finally, since the Theano library now has the low-precision datatype, we could repeat our analysis for other non-DNN algorithms to determine if they too could benefit from lower-precision arithmetic.

Appendix A

Customtypes Header File

```
// If Cuda compiler: declare all functions as both host and device.
#ifdef __CUDACC__
#define CUDA_CALLABLE __host__ __device__
#else
#define CUDA_CALLABLE
#endif

// If C compiler: define simple LP struct.
#ifdef __cplusplus
struct ga_usertype01;   //forward declaration
#else
typedef struct _usertype01 {
    int sign;
    int val;
} ga_usertype01;
#endif

// For all compilers: define C functions.
static CUDA_CALLABLE ga_usertype01 usertype01_add(ga_usertype01 x, ga_usertype01 y);
/* More functions... */

#ifdef __cplusplus

// If C++ or Cuda compiler: define LP struct with methods.
struct ga_usertype01 {
    int sign;
    int val;

    inline CUDA_CALLABLE ga_usertype01();
    inline CUDA_CALLABLE ga_usertype01(const float& y);

    inline CUDA_CALLABLE ga_usertype01 operator +() const;
    inline CUDA_CALLABLE ga_usertype01 & operator +=(const ga_usertype01 &y);
    /* More methods... */
};

// If C++ or Cuda compiler: Define overloaded operators.
inline CUDA_CALLABLE ga_usertype01 operator +(const ga_usertype01 &x, const float &y);
inline CUDA_CALLABLE ga_usertype01 operator +(const float &y, const ga_usertype01 &x);
inline CUDA_CALLABLE ga_usertype01 operator +(const ga_usertype01 &x, const ga_usertype01 &y
    );
```

```
/* More operators... */

// If C++ or Cuda compiler: Define overloaded arithmetic functions.
static CUDA_CALLABLE ga_usertype01 max(const ga_usertype01 &x, const ga_usertype01 &y);
static CUDA_CALLABLE ga_usertype01 abs(const ga_usertype01 &y);
static CUDA_CALLABLE ga_usertype01 exp(const ga_usertype01 &y);
static CUDA_CALLABLE ga_usertype01 log(const ga_usertype01 &y);
static CUDA_CALLABLE ga_usertype01 pow(const ga_usertype01 &y, const float &x);
static CUDA_CALLABLE ga_usertype01 pow(const ga_usertype01 &y, const ga_usertype01 &x);
static CUDA_CALLABLE ga_usertype01 sqrt(const ga_usertype01 &y);

#endif
```

Listing A.1: The Customtypes header file that contains the LP simulation code. We use compiler macros to make it compatible with C, C++, and Cuda.

Appendix B

Additional Data

Chapter 4 includes data for only the best epoch from each DNN. This appendix includes data from all epochs. We include the data for the TIMIT and WSJ experiments that compare conventional float DNNs to approximate DNNs.

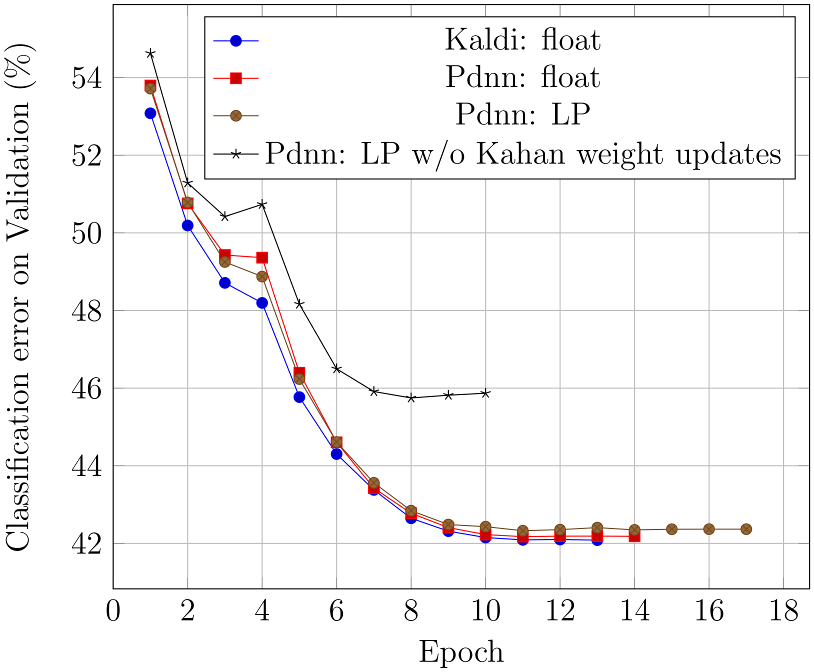
Two metrics are included: cross-entropy and classification error. The cross-entropy of the approximate DNN is computed using low-precision arithmetic; it might be slightly higher or slightly lower than the float DNN's cross-entropy simply by virtue of the approximate computation. Therefore, the cross-entropies of the approximate and float DNNs cannot be fairly compared. In general, the cross-entropies are similar for both types of DNNs, but no conclusions should be drawn from comparing cross-entropies. Unlike cross-entropy, classification error is a valid metric for comparison. Approximate arithmetic influences how the examples are labelled, but counting the incorrectly labelled examples has nothing to do with the type of arithmetic used. Therefore, classification errors can be fairly compared across DNNs.

Two observations can be made from the plots below. First, the approximate DNNs converge in roughly the same number of epochs as the float DNNs. Second, for the DNNs that do converge, the shapes of the cross-validation and cross-entropy curves are very similar. The approximate DNN behaves similarly to the float DNN, and both DNNs progress at similar rates during training.

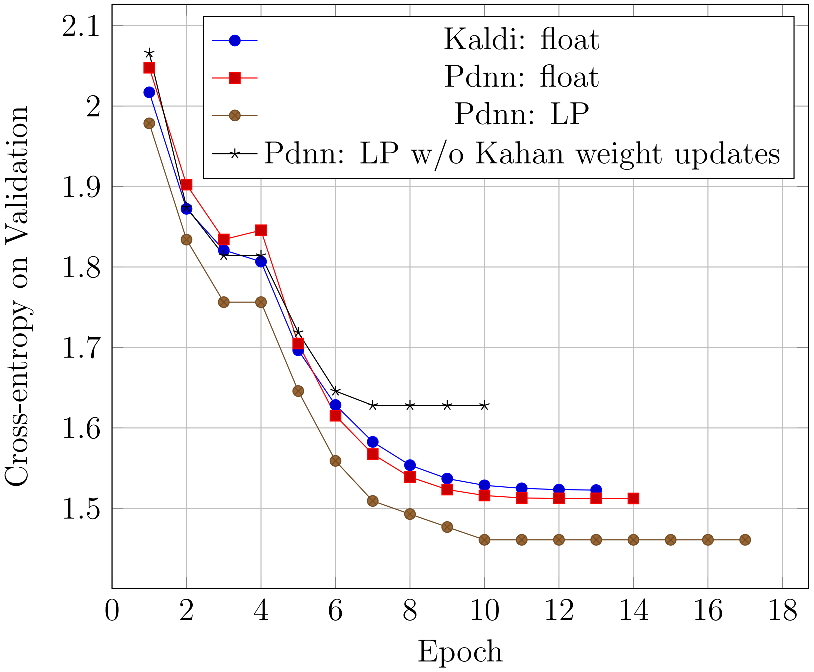
B.1 TIMIT Smaller DNN

Additional data for experiment in Section 4.3.2

Classification errors by epoch for Table 4.4



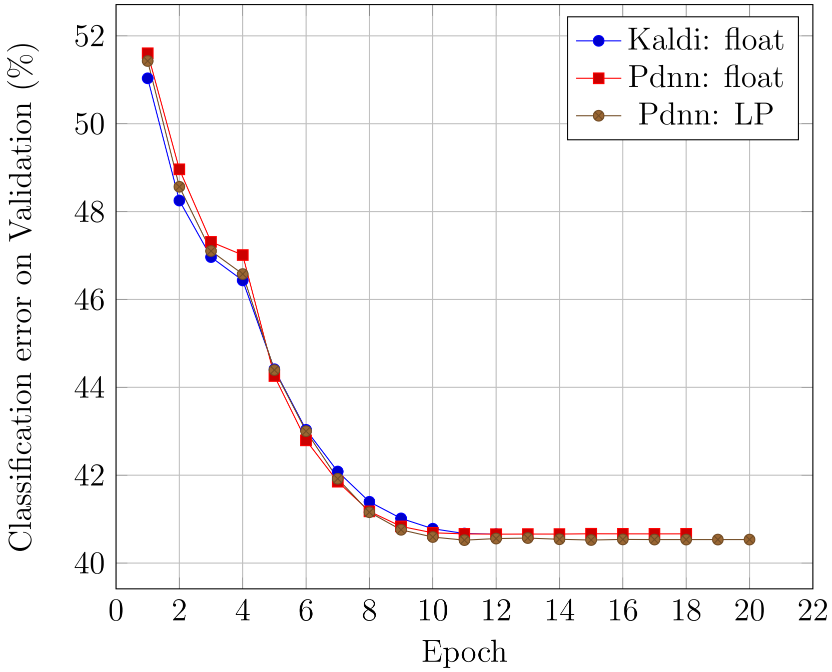
Cross-entropies by epoch for Table 4.4



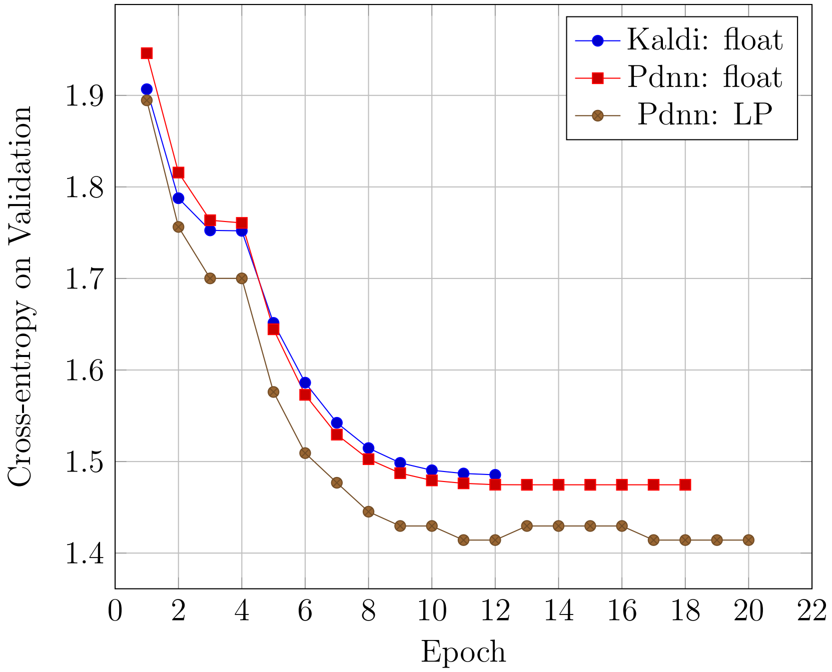
B.2 TIMIT Full DNN

Additional data for experiment in Section 4.3.5

Classification errors by epoch for Table 4.6



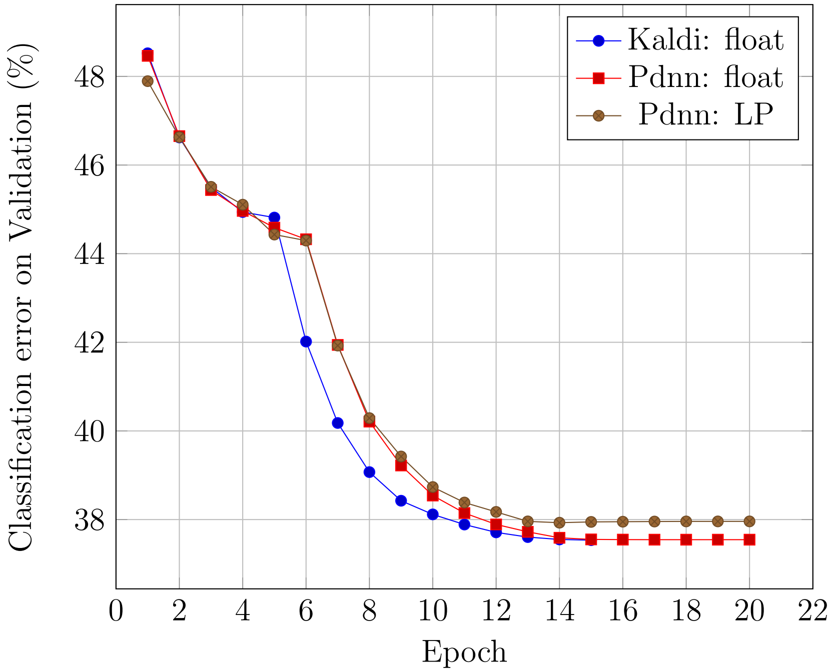
Cross-entropies by epoch for Table 4.6



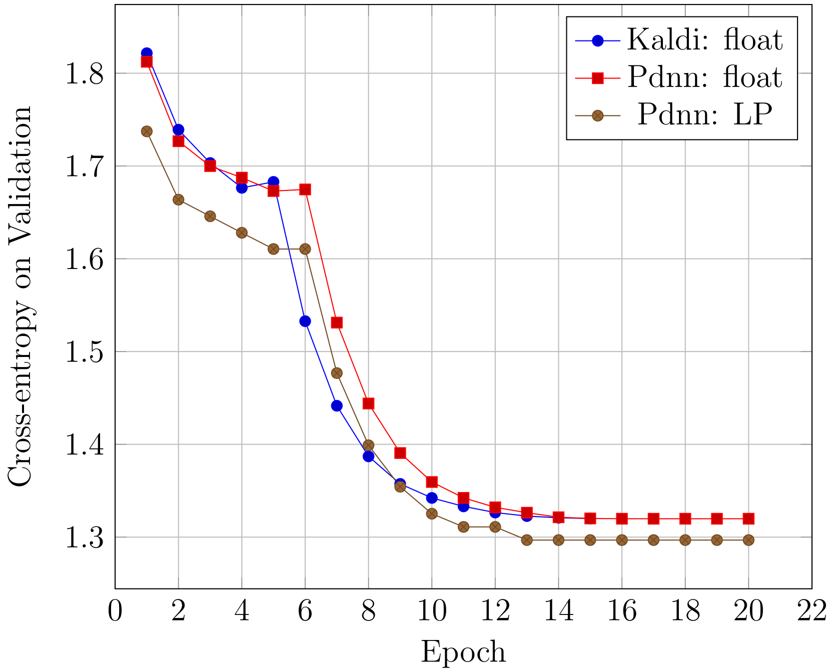
B.3 WSJ Smaller DNN

Additional data for experiment in Section 4.4.2

Classification errors by epoch for Table 4.8



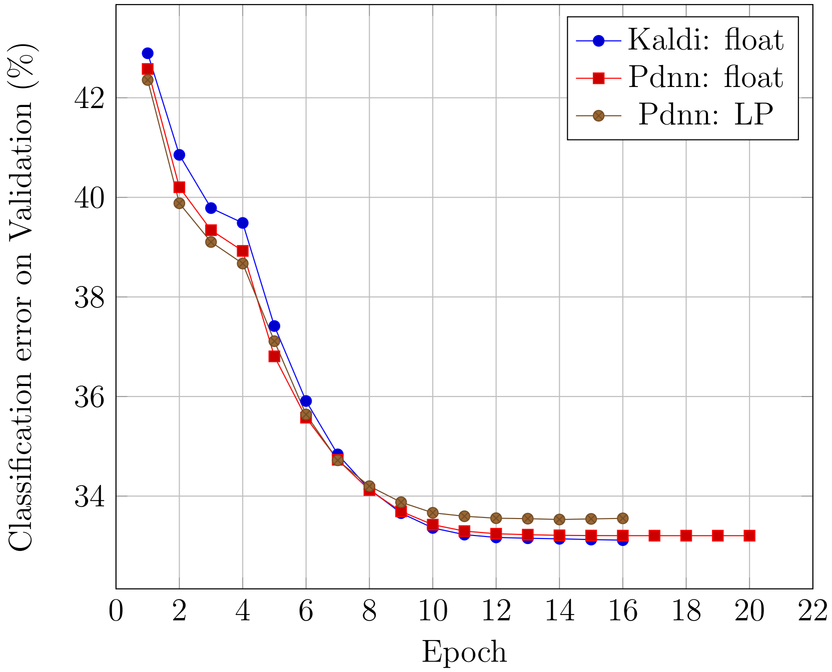
Cross-entropies by epoch for Table 4.8



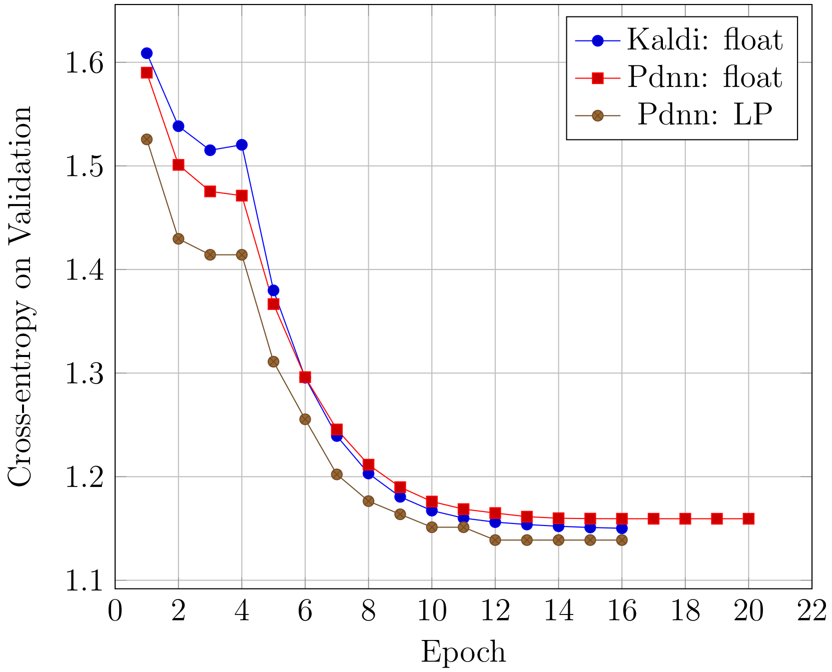
B.4 WSJ Full DNN

Additional data for experiment in Section 4.4.3

Classification errors by epoch for Table 4.9



Cross-entropies by epoch for Table 4.9



Bibliography

[1] FloPoCo. http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/.

[2] Karel's DNN implementation. http://kaldi.sourceforge.net/dnn1.html.

[3] Libgpuarray. http://deeplearning.net/software/libgpuarray/.

[4] Magma. http://icl.cs.utk.edu/magma/software/.

[5] NumPy. http://www.numpy.org/.

[6] Python/C API reference manual. https://docs.python.org/2.7//c-api/index.html.

[7] User-defined conversion. http://en.cppreference.com/w/cpp/language/cast_operator.

[8] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[9] Joe Bates. Personal communication.

[10] Joe Bates. Approximate computing. Personal communication.

[11] Joseph Bates. Processing with compact arithmetic processing element, April 3 2012. US Patent 8,150,902.

[12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[13] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[14] Jay Bourque. Numpy-dtypes. https://github.com/jayvius/numpy-dtypes/.

[15] William Chan and Ian Lane. Deep recurrent neural networks for acoustic modelling. *arXiv preprint arXiv:1504.01482*, 2015.

[16] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. Pipelined backpropagation for context-dependent deep neural networks. In *INTERSPEECH*, 2012.

[17] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[18] Jalal Foroozesh, Abbas Khosravani, Adel Mohsenzadeh, and Ali Haghighat Mesbahi. Application of Artificial Intelligence (AI) modeling in kinetics of methane hydrate growth. *American Journal of Analytical Chemistry*, 2013, 2013.

[19] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[20] Georg Heigold, Vincent Vanhoucke, Andrew Senior, Patrick Nguyen, M Ranzato, Matthieu Devin, and Jeffrey Dean. Multilingual acoustic models using distributed deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8619–8623. IEEE, 2013.

[21] Nicholas J Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993.

[22] Geoffrey Hinton. A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1):926, 2010.

[23] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[24] Geoffrey E Hinton. To recognize shapes, first learn to generate images. *Progress in brain research*, 165:535–547, 2007.

[25] Steve Hollasch. IEEE standard 754 floating point numbers. `http://steve.hollasch.net/cgindex/coding/ieeefloat.html`.

[26] Navdeep Jaitly, Patrick Nguyen, Andrew W Senior, and Vincent Vanhoucke. Application of pretrained deep neural networks to large vocabulary speech recognition. In *INTERSPEECH*, 2012.

[27] William Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[28] Matthew Lambert. Nvidia GeForce GTX TITAN Black review - power and thermals. `http://www.bit-tech.net/hardware/graphics/2014/02/26/nvidia-geforce-gtx-titan-black-review/8`.

[29] Yann LeCun and Corinna Cortes. The MNIST database of handwritten digits, 1998.

[30] Carla Lopes and Fernando Perdigão. Phone recognition on the TIMIT database. *Speech Technologies*, 1:285–302, 2011.

[31] Yajie Miao. Kaldi+PDNN - building DNN-based ASR systems with Kaldi and PDNN. `http://www.cs.cmu.edu/~ymiao/kaldipdnn.html`.

[32] Yajie Miao. Kaldi+PDNN: Building DNN-based ASR systems with Kaldi and PDNN. *Computing Research Repository*, abs/1401.6984, 2014.

[33] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[34] Nikolay Nikolaev. Single-layer perceptrons. `http://homepages.gold.ac.uk/nikolaev/311perc.htm`.

[35] NIST. Matched pairs sentence-segment word error (MAPSSWE) test. `http://www.icsi.berkeley.edu/ftp/pub/speech/papers/wikipapers/nist_mapsswe.html`.

[36] NIST. Speech Recognition Scoring Toolkit (2.4.9). `http://www.nist.gov/itl/iad/mig/tools.cfm`, 2014.

[37] Nvidia. CUDA / Cublas. `http://www.nvidia.com/object/cuda_home_new.html`.

[38] Nvidia. GeForce GTX TITAN Black - specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/specifications`.

[39] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The Kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, 2011.

[40] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[41] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. On parallelizability of stochastic gradient descent for speech DNNs. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 235–239, 2014.