12-2018

Low-Precision Linear Algebra for Neural Networks

Frost Bennion Mitchell
Utah State University

LOW-PRECISION LINEAR ALGEBRA FOR NEURAL NETWORKS

by

Frost Bennion Mitchell

Capstone submitted in partial fulfillment of
the requirements for graduation with

UNIVERSITY HONORS

with a major in

Computer Engineering
in the Department of Electrical and Computer Engineering

Approved:

Capstone Mentor	**Departmental Honors Advisor**
Dr. Don Cripps	Dr. Jacob Gunther

University Honors Program Director
Dr. Kristine Miller

UTAH STATE UNIVERSITY
Logan, UT

Fall 2018

Abstract

Neural networks have been widely responsible for recent advances in machine learning, powering technologies such as digital assistants and AR photography. LPLANN (Low-Precision Linear Algebra for Neural Networks) is a cross-platform library written in C++ used for implementing neural networks. The software allows users to set specific levels of precision for calculations. Low-precision calculations use advanced parallelization techniques (SIMD, SWAR) to run neural networks at faster rates than full-precision calculations. This library is lightweight enough to run on embedded systems, only relies on OpenMP as a dependency, and is portable to any operating system. LPLANN also includes optimizations to provide drastic speedups on a workstation, allowing it so serve as a testbed for novel low-precision neural network architectures. The purpose of this project was to implement optimizations and to test how the execution time of binary networks compares to that of floating-point networks. Performing 2-dimensional convolution using 3×3 filters is implemented on binary weights with 43% overhead, and 1.5% overhead for 7×7 filters. Depending on the architecture of the neural network, speedup with 3×3 filters varied from 2.5× to 8×.

Acknowledgements

I would like to thank Dr. Jacob Gunther, Dr. Koushik Chakraborty, and the USU Department of Electrical and Computer Engineering for providing computing resources to train neural networks, as well as Dr. Don Cripps for his counsel in system design and how an ideal software system should behave. I would also like to thank Jolynne Berrett for her help in producing such fine documentation for the project. Most of all, I want to thank my wife Sarah for her support as I've pursued my passions.

Contents

1 Introduction **1**

1.1 Background 1

1.2 What LPLANN is (and what it isn't) 1

2 Methods **3**

2.1 System Design 3

2.2 Software Design 4

3 Testing and Results **6**

3.1 Mathematical Tests 6

3.2 Interface Tests 7

3.3 Functional Tests 7

4 Conclusion **9**

List of Figures

1 A simple convolutional neural network which classifies input. 1

2 A high-level system diagram of the LPLANN system. 3

3 Binary values are stuffed into a single variable, then XOR is used to simulate multiplication. The sum of the bits is calculated, and the sign bit of the sum is the final result. 4

4 Software diagram for LPLANN. Input and output layers are fed to an operation wrapper which performs calculations. 4

List of Tables

1 Verification of interface requirements. 6

2 Verification of mathematical requirements. 7

3 Verification of functional requirements. 7

1 Introduction

Convolutional neural networks (CNNs) have been used in a variety of applications such as voice transcription and translation, image classification, object recognition, and more. The widespread use of "deep learning" in these applications is largely due to advancing graphics processing unit (GPU) technology. The massive parallelization afforded by GPUs allows CNNs to be trained on large datasets in days or hours rather than weeks or months.

GPUs are power hungry, expensive devices which are not typically used in size and power-constrained environments. Low-precision Linear Algebra for Neural Networks (LPLANN) is a library which implements neural networks in low-precision arithmetic. By using low-precision operations, a CNN can be run several times faster than is possible using full-precision arithmetic.

As research on the feasibility of using low-precision weights to compress CNNs is ongoing, LPLANN is a testbed where low-precision CNNs can be evaluated.

All of the LPLANN source code, along with documentation and instructions for use, is available at `https://github.com/kbentum/LPLANN`.

1.1 Background

Since the introduction of AlexNet in 2012 by Krizhevsky et al. [1], the use of deep neural networks (DNNs) has dominated the field of machine learning to solve tasks that are difficult to explicitly program. DNNs can have over 100 layers and millions of 32-bit parameters. Each layer of a DNN applies convolutional filters to an input, and a feature map is output by each filter. Initial layers in the network extract features which are fed into subsequent layers. These features are used to classify an input, as shown in Figure 1.

Recent efforts to improve the computational and memory requirements of deep neural networks use low-precision representations for network parameters. Jain et al. [2] have developed a technique using 8-bit parameters for a classifier network with reduction in accuracy of less than 0.5%. Other researchers have compressed networks even further, producing entirely binary networks [3, 4].

These binary networks can be stored using 1/32 of the memory required by an equivalent full-precision model and also require a fraction of the operations. LPLANN provides a framework to execute binary and low-precision CNNs. With these energy and memory savings, CNNs can be implemented on the CPU rather than the GPU, even on embedded systems.

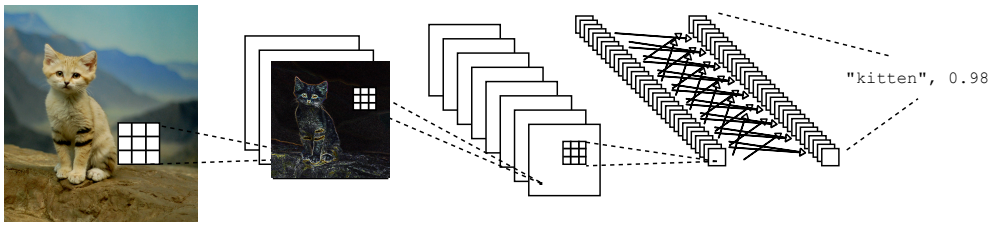

Figure 1: A simple convolutional neural network which classifies input.

1.2 What LPLANN is (and what it isn't)

LPLANN is a library written in C++ to perform the operations necessary to execute binary and low-precision CNNs. Some of the implemented operations are:

- Convolution with zero and repeat padding
- Activation functions such as ReLU and leaky ReLU
- Matrix operations, including addition and multiplication

LPLANN does not perform the backpropagation operation used to train neural networks. The library executes a network using pre-trained weights and an input. The library is intended to operate on multi-channel input images, such as ordinary color images. The system is currently designed only to perform the 2-dimensional convolution operation.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[2] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, "Compensated-dnn: energy efficient low-precision deep neural networks by compensating quantization errors," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[3] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[4] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

2 Methods

This project consists entirely of a system-independent software library. System design and optimizations are discussed, then software structure and modules are outlined. The main objectives of the LPLANN system are the following:

- Implement functions required to execute basic convolutional neural networks (CNNs).
- Execute CNNs in variable precision. Operations can be performed in precision ranging from 1 to 32 bits.
- Write portable code that can operate with minimal configuration on any operating system.
- Optimize code to run entirely on the CPU.

These goals and the associated requirements were outlined in the LPLANN Specifications Document, and are discussed in depth in Section 3.

2.1 System Design

The LPLANN system acts as a high-level API for implementing neural networks. To implement a neural network, a user would follow five main steps:

1. Pass network dimensions as function arguments.
2. Read corresponding parameters from a file.
3. Initialize network layers to allocate memory and prepare for execution.
4. Pass input to the first layer.
5. Call operation wrappers to execute each layer.

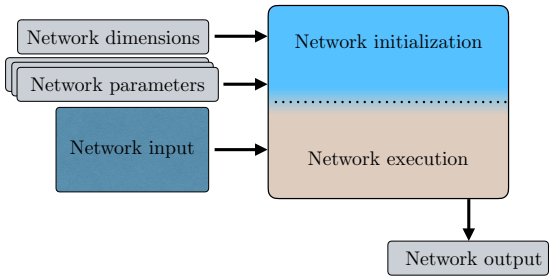

Figure 2: A high-level system diagram of the LPLANN system.

Each layer is a separate object with a specified precision type of either 1, 2, 4, 8, or 32-bits. Binary (1-bit) values are stored as multiple values within a single 16 or 64-bit register. 2, 4, and 8-bit values are implemented as an 8-bit fixed precision number, which means that 2 and 4-bit values have no performance gain over 8 bit values, but are valuable to test accuracy for low-precision CNNs.

The main optimization implemented for binary calculations was applying Single Instruction, Multiple Data (SIMD) principles to multiple values stored in a single register. The values 0 and 1 were used to represent 1 and -1, respectively. A 3×3 filter could be stored in a single 16-bit variable and applied to input using a single operation rather than 9 separate multiply/accumulate operations, as shown in Figure 3. This technique is referred to in this paper as "binary stuffing". By stuffing binary values into a single register, the number of operations can be reduced by 89% or more. However, some overhead is incurred by moving data into single registers. A custom hardware implementation could achieve faster results than a general purpose processor for binary stuffing.

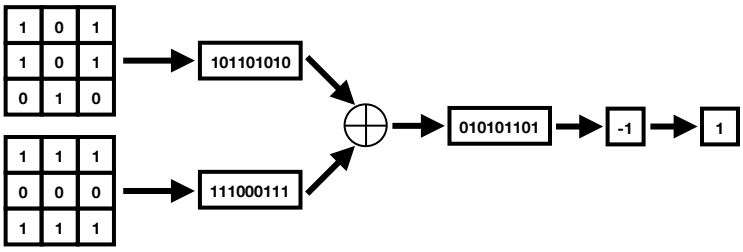

Figure 3: Binary values are stuffed into a single variable, then XOR is used to simulate multiplication. The sum of the bits is calculated, and the sign bit of the sum is the final result.

When a 3×3 filter is stuffed into a 16-bit register, 7 bits remain unused, representing a 43% overhead. This overhead can be reduced by using larger filters and registers. For example, a 7×7 filter stuffed into a 64-bit register would only have one bit of overhead, approximately 1.5%. When performing matrix multiplication, the overhead when using binary stuffing is typically quite small.

2.2 Software Design

LPLANN was written entirely in C++. It relies on one open source library to read in network weights. Operations are templatized for floating point and fixed point calculations, with custom convolution and multiplication functions written for binary calculations.

Using the LPLANN system begins with a user initializing layer objects with the desired shape and precision. Layers contains weights and output values, and these values are used by an operation wrapper, as shown in Figure 4. An operation wrapper function does the following:

1. Receives two layer objects
2. Performs any necessary precision conversion, such as conversion from 8-bit fixed precision to 32-bit floating point precision,
3. Performs the required operation
4. Stores values in the output layer.

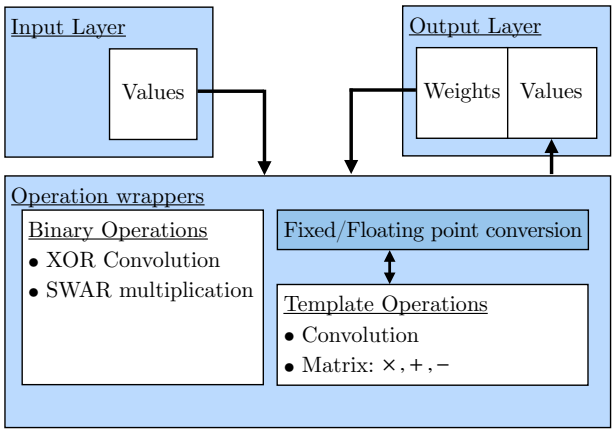

Figure 4: Software diagram for LPLANN. Input and output layers are fed to an operation wrapper which performs calculations.

2.2.1 Layers

Each layer contains a pointer to a three Matrix objects: bias, weight, and output. The layer also contains functions to read values from a `.npy` file into the bias and weight objects. A special input layer inherits from the layer class and has additional functions to read in an input from a `.npy` file and can also accept a pointer to an array as input.

To create a layer, the type of the layer must be specified along with the necessary dimensions. Not every type of layer accepts the same number of dimensions. For example, a convolution layer accepts 4

dimensions since the weight object is 4-dimensional. However, a pooling layer will only perform a 2-dimensional operation, so only 2 dimensions must be specified.

During the initialization of each layer, the user must also specify if the layer has a bias vector to add to output after an operation is completed.

2.2.2 Operation Wrappers

Although LPLANN does not have as many operations available as some neural network libraries, the functions necessary to execute basic DNNs are implemented. Each operation wrapper accepts an input and an output layer, as shown in Figure 4.

LPLANN has the following wrappers available:

- **Convolution** - Perform convolution with repeat or zero padding.
- **Pooling** - Downsample a layer using either the maximum value or average value in an area. For binary networks, a special zero-pooling function is used and is equivalent to max-pooling.
- **ReLU** and **Leaky ReLU** - Set negative values to either zero or a small value. ReLU works on fixed or floating point inputs.
- **Flatten** - Prepare an input for a fully connected layer by flattening the layer.
- **Fully connected layer** - Perform matrix multiplication, known as a fully connected layer.
- **Softmax** - Normalize a layer using the softmax function. This is typically used to convert a layer output to a confidence percentage.
- **Binary to float** - Convert binary values to floating point precision.

2.2.3 Execution

Each operation wrapper is called individually to execute the operation. The values are stored in the output object of the output layer. Since LPLANN is designed to be a high level API, calling each wrapper is an single line of code where the only parameters are the layer objects and one additional parameter, depending on the particular wrapper. For examples of how this code is called, see `https://github.com/kbentum/LPLANN/wiki`.

3 Testing and Results

The tests from the LPLANN Specifications Document are listed here, followed by tables of test results. Descriptions of test results are provided where requirements were not met.

3.1 Mathematical Tests

3.1.1 Input network parameters which will prepare layers in each of the available data types. Input data in each of the types of low-precision data, and verify accuracy.

3.1.2 Add 1, 2, and 3-dimensional arrays, and verify accuracy.

3.1.3 Test element-wise multiplication and ordinary matrix multiplication on 1, 2, and 3-dimensional arrays, and verify accuracy.

3.1.4 Test and verify the ReLU function on low-precision data. Functions will be run element-wise on 1, 2, and 3-dimensional arrays.

3.1.5 Test and verify the parameterized ReLU function on low-precision data. Functions will be run element-wise on 1, 2, and 3-dimensional arrays.

3.1.6 Test the Boolean operations, AND, OR, NOT, and XOR on 1 bit data, and verify accuracy.

3.1.7 Verify that the output of an operation is changed to higher precision if upgrading precision, and to a lower precision of downgrading. Verify that downgrading precision results in data being rounded to the closest available number in the lower precision set of numbers.

3.1.8 Verify that 1, 2, and 3-dimensional convolution give expected output. Verify convolution with and without zero padding.

Table 1: Verification of interface requirements.

Test Number	Verification	Requirement met
3.1.1	Testing	Yes
3.1.2	Testing	Yes
3.1.3	Testing	Partial
3.1.4	Testing	Yes
3.1.5	Testing	Yes
3.1.6	Testing	Yes
3.1.7	Testing	Partial
3.1.8	Testing	Partial

Verification 3.1.3 requires that element-wise multiplication and ordinary matrix multiplication are accurate on 3 dimensional arrays. Although there are some specific instances where element-wise multiplication would be useful, it was not needed for the basic networks tested during development, so element-wise multiplication was not implemented due to time constraints. Matrix multiplication is also not implemented for a 3-dimensional matrix.

Verification 3.1.7 requires that operations are converted to higher and lower precision according to input parameters. Currently only 8-bit operations are implemented in LPLANN, due to time constraints. Conversion between binary, floating point, and 8-bit fixed point numbers is working correctly.

Verification 3.1.8 requires that 1, 2, and 3-dimensional convolution functions are implemented. Due to time constraints, only 1 and 2-dimensional convolution is implemented. For 8-bit fixed point values, tests revealed an unexpected error in output from the convolution function, and accuracy was not verified successfully.

3.2 Interface Tests

3.2.1 Verify that the function calls of the API follow the ISO C++ Core Guidelines.

3.2.2 Pass 1, 2, and 3-dimensional data into the system and verify output is correct according to the requirements of 3.1. Verify that the data is either floating point, fixed point, or integer data.

Table 2: Verification of mathematical requirements.

Test Number	**Verification**	**Requirement met**
3.2.1	Inspection	Yes
3.2.2	Testing	Partial

Verification 3.2.2 requires that the mathematical requirements were met for 1, 2, and 3-dimensional data. All these requirements were not met. See Section 3.1 for test results.

3.3 Functional Tests

3.3.1 Verify the system operates on Linux, MacOS, and Windows operating systems.

3.3.2 Verify that the system can access and utilize a GPU for computation when available. Verify this on Linux, MacOS, and Windows operating systems.

3.3.3 Verify that the system can access and utilize multiple processors. Verify this on Linux, MacOS, and Windows operating systems.

3.3.4 Time each operation on each value in the low-precision data, and verify that timing for lower-precision operations is less than higher-precision operations.

Table 3: Verification of functional requirements.

Test Number	**Test**	**Requirement met**
3.3.1	Demonstration	Yes
3.3.2	Testing	No
3.3.3	Testing	Yes
3.3.4	Testing	Partial

Verification 3.3.2 requires that a GPU be utilized for computation when availabe. Due to time constraints, a GPU kernel was not implemented.

Verification 3.3.4 requires each operation runs faster at lower precision than at higher precision. Currently binary operations run much faster than floating point operations, but 8-bit fixed point operations run more slowly than both. In a test classifying input from a USB webcam, the binary network ran $2.5\times$ faster than the floating point version. Tests with a different network architecture had a binary network that ran $6\times$ faster.

4 Conclusion

LPLANN accurately executes DNNs. Although some of the features outlined in the specifications document were not implemented in the final design, the system is able to execute binary and full precision neural networks. As expected, the binary network is much faster than the floating-point network, with even higher speedups possible based on the architecture of the network. It is possible that even greater speedups could be achieved by implementing low-level optimizations such as removing branches and stuffing multiple filters into a single register.

The software ran easily on embedded systems as well as multicore laptops, showing how portable and versatile the software is. With no dependencies on large libraries, LPLANN is an ideal system to implement on IoT devices or on other constrained systems.

The system does not accurately execute 2, 4, or 8-bit code as it was originally intended to do. This was due to the time constraints of the project. Although this was unfortunate, the ability of LPLANN to execute binary neural networks is a significant accomplishment. It is an unprecedented project, since no open source library is focused specifically on executing binary neural networks.

LPLANN has successfully completed the objectives outlined in Section 2. It can successfully execute basic CNNs in variable precision, and is portable to various operating systems, relying only on one common open-source library (OpenMP). It is also optimized for CPU processing. As there are still optimizations that can be made to improve the speed of binary and full-precision networks, the software could be improved. However, LPLANN has accomplished the main objectives of the project.

All of the LPLANN source code, along with documentation and instructions for use, is available at `https://github.com/kbentum/LPLANN`.

Capstone Reflection

Word count - 1,009 words

Working on LPLANN has been one of the highlights of my undergraduate education, mostly because of how it has been an extremely different experience compared to coursework as well as my undergraduate research. I've had extensive experience working with three different professors on research projects where fundamental research was being conducted. This project was quite different in a few key ways. Instead of using existing software in order to conduct research as I have done in my undergraduate research, my capstone project has been focused on engineering a system from the ground up, with research focused on the results of my software design. It has provided me with incredibly valuable software engineering experience, which has definitely not been a focus of my undergraduate coursework. Ultimately, it has given me the opportunity to tie together concepts such as processor pipelines, basic signal processing operations, and mathematical computation all in a single software system.

Although I intend to pursue a Ph.D. in computer science, my undergraduate degree in computer engineering has not focused on the principles of computer science and good development techniques. As I began to write code for LPLANN, I was overwhelmed by how little I knew and how much there was to learn about software engineering. As it stands now, I feel that I have developed a wonderful piece of software which uses modern C++ techniques in responsible ways, something that I could have only dreamed to have written just 6 months ago. Next month I am starting a full time position in industry which I secured entirely because of the software engineering skills that I developed while writing LPLANN.

It was surprising to me how simple a technique can seem until you sit down to program it. LPLANN began with an idea that it should be possible to run a binary network in a fraction of the time, since you could stuff a register full of individual bits and perform a single operation to compute multiple results. It's a fairly simple idea that I felt certain I could implement. But when it came to writing the code, I was frankly astonished by how long I had to sit and think through where individual bits were being stored and how the system would need to be changed based on variable such as register size and kernel size. It was definitely an exercise that required a lot of focus.

Testing the performance of LPLANN was a very rewarding experience. Although I had done a few homework assignments that involved optimizing code, it's quite a different experience when you have written the entire library yourself. When you have the complete freedom to change the behavior of an entire function, it can be a little overwhelming to find exactly which optimization to make. Finding these optimizations and evaluating their effects was the main research aspect of my capstone. I had to apply the techniques learned in computer architecture classes to my code, then verify that I was getting the expected behavior. In several cases, I found that my code was introducing precision errors so certain optimizations couldn't be implemented, such fully fixed point arithmetic. In these cases I had to determine exactly which functions I had time to implement and what I could reasonably expect from myself.

One of the highlights of this entire capstone project was running tests on a Raspberry Pi. A $35 single-board computer was running a computationally intensive network with 1.29 billion operations at just under 1 frame per second. It was kind of a marvellous experience, and quite humbling to consider how technology has advanced to this point.

LPLANN required extensive research into very recent papers published on binary neural networks. Although the mathematical concept is not complicated, it was only recently shown that a binary network could be used for a task as complex as image classification, so I had to become familiar with what binary networks could accomplish, as well as how they could be applied usefully. One research paper theorized that binary networks could have 58× speedups compared to full-precision networks, but my implementations had only fraction of that speed, which goes to show how theoretical computations and actual implementations can be drastically different.

Although I don't believe the software I wrote is perfect or even better than some existing libraries available today, I'm proud of the code that I have written. To my knowledge no open source project has attempted to implement binary neural networks as a portable project, so in this area LPLANN is an original, unprecedented work. In particular, I'm proud of the modular design of my code. Writing template functions that can operate on both floating point and fixed point numbers, as well as performing completely different operations based on the input type was a significant part of my software.

If I had to repeat my project again, I would have designed a simpler system so that I could spend more time on the research aspects of my project. There were particular aspects of my research plan that I was quite excited about, but wasn't able to dive into because I was trying to get other parts of the system working. I would have make the 2, 4, and 8-bit code simpler by simulating the low-precision values using floating-point numbers so that I could focus on optimizing binary code to reduce overhead for convolution. I also would have not attempted such an ambitious undertaking for someone with relatively little software engineering experience. However, I can't say it was a bad idea, because I have learned a tremendous amount.

I'm extremely grateful for this capstone project. I've learned that I am capable of writing such a complex library. I've learned that applying the research of other scientists and engineers can be both rewarding and frustrating, but mostly educational. I've learned software engineering skills that I wouldn't have in my undergraduate courses, and I've become familiar with a fascinating topic in machine learning. I hope that LPLANN continues to be useful to me in graduate studies as I research machine learning.

About the Author

Frost Bennion Mitchell is graduating with a degree in Computer Engineering from Utah State University, with minors in Computer Science and Mathematics and a GPA of 3.98. He has been recognized as an Undergradate Research Fellow and Undergraduate Teaching Fellow. He was awarded the Outstanding Junior Award from the Department of Electrical and Computer Engineering. He intends to pursue a Ph.D. at the University of Washington, studying Computer Science.