SHM-Learn-Dist: An optimized GPU-Powered C++ Distributed Deep Learning Library

Souham Biswas

Illinois Institute of Technology

sbiswas7@hawk.iit.edu

Abstract

To handle training over large datasets, distributed training approaches are very popular. There are mainly two types of distributed training – data-parallel & model-parallel. While data-parallel approaches are easier to implement and include greater support, model-parallelism is usually used to train large neural nets too big to fit on one GPU. In this paper, the workings of the SHM-Learn-Dist library is described which yields training performance boosts in a distributed setting. Support currently exists for data-parallel training.

1. Introduction

Data parallel training is particularly useful in cases where there is huge diversity in data or the mapping to be learnt is complex. The main idea behind data parallel training involves the following two modes –

* Gradient Averaged Data Parallelism
* Weight Averaged Data Parallelism

We shall discuss about both forms of data parallelism and give some pointers about the use cases of each.

Model parallelism is especially useful when we are training very big neural networks of the order of billions of parameters. Here, the model parameters may be distributed across different nodes. It is illustrated in Fig. 1.0.

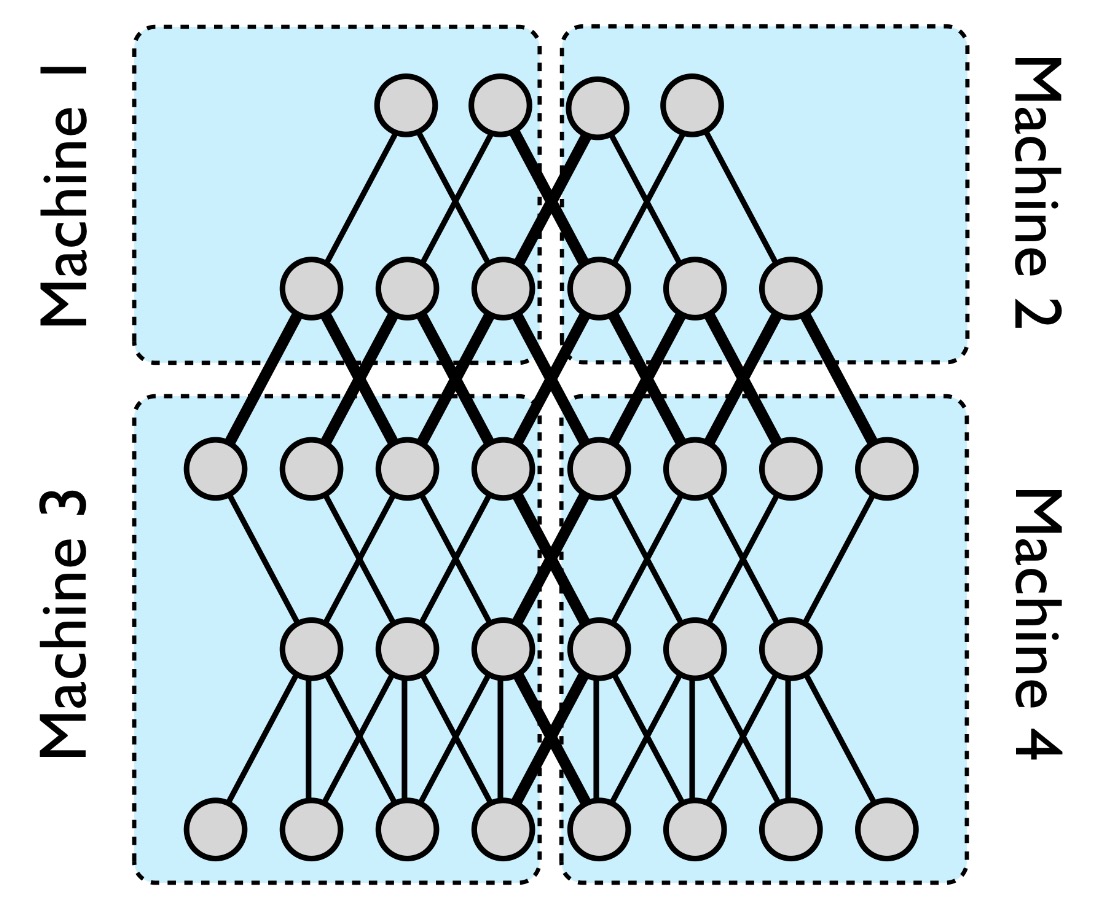


Fig. 1.0 – Model Parallelism illustration[1]

As is clearly evident, there is significant communication involved within every layer here.

The organization of the paper shall be in the following way –

1. Firstly, explore the 2 different types of data parallelisms mentioned previously.
2. Second, we delve in to understanding the fundamental effects of data parallelism on training by observing the parameters of a single neuron which is subject to distributed training.
3. Next, the results of some parameter searches for finding the optimal distributed setting to train on the CIFAR-10 dataset are presented.
4. The fourth section elicits the exact mathematical explanation of the memory-realignment algorithm which contributes greatly to the speedup and presents the GPU kernels and their invocation patterns.
5. Next, the plots involving distributed training on the CIFAR-10 dataset using a custom-augmented Tensorflow are presented to give a general idea of how the CIFAR-10 dataset behaves in a distributed setting.
6. In the sixth section, performance and scaling plots of distributed training experiments with SHM-Learn-Dist are presented and compared with other toolkits.
7. Finally, the future work is mentioned which is planned.

2. Types of Data Parallelism

Data parallel trainings are of 2 types namely -

1. Gradient Averaged
2. Weight Averaged

The main difference between these two methods is that in gradient averaging, every worker gets the same set of weights from the parameter server, computes gradients which are averaged element-wise by the parameter server and applied on the same set of weights which each worker started with. The update equation is presented in Eq. 2.1 –

Eq. 2.1 Weight update equation in gradient averaged data parallelism.

Here, is the number of workers and is the gradient from the worker; is the learning rate.

In weight averaging, every worker starts with its own set of weights which vary between workers. The workers compute *and apply the gradients* to get new weights which are sent to the parameter server. The parameter server averages these weights element-wise and sends them back to the workers who continue the process. The update equation for the weight averaged technique is presented in Eq. 2.2 –

Eq 2.2 Weight update equation in weight averaged data parallelism

Upon simplifying, it becomes Eq. 2.3 –

Eq. 2.2 Simplified weight update equation in weight averaged data parallelism

Here, is the set of network parameters of the model; the remaining symbols have the same meanings as explained previously.

The main difference intuitively observable here is that the weight averaged technique promotes diversity in the weights along with the gradients; whereas in the gradient averaged technique, only the gradients are averaged.

3. Single Neuron Distributed Training

An experiment script was written which simulates distributed training of one single neuron with a sigmoid activation function which is trying to learn the following simple function as stated in Eq. 3.1.

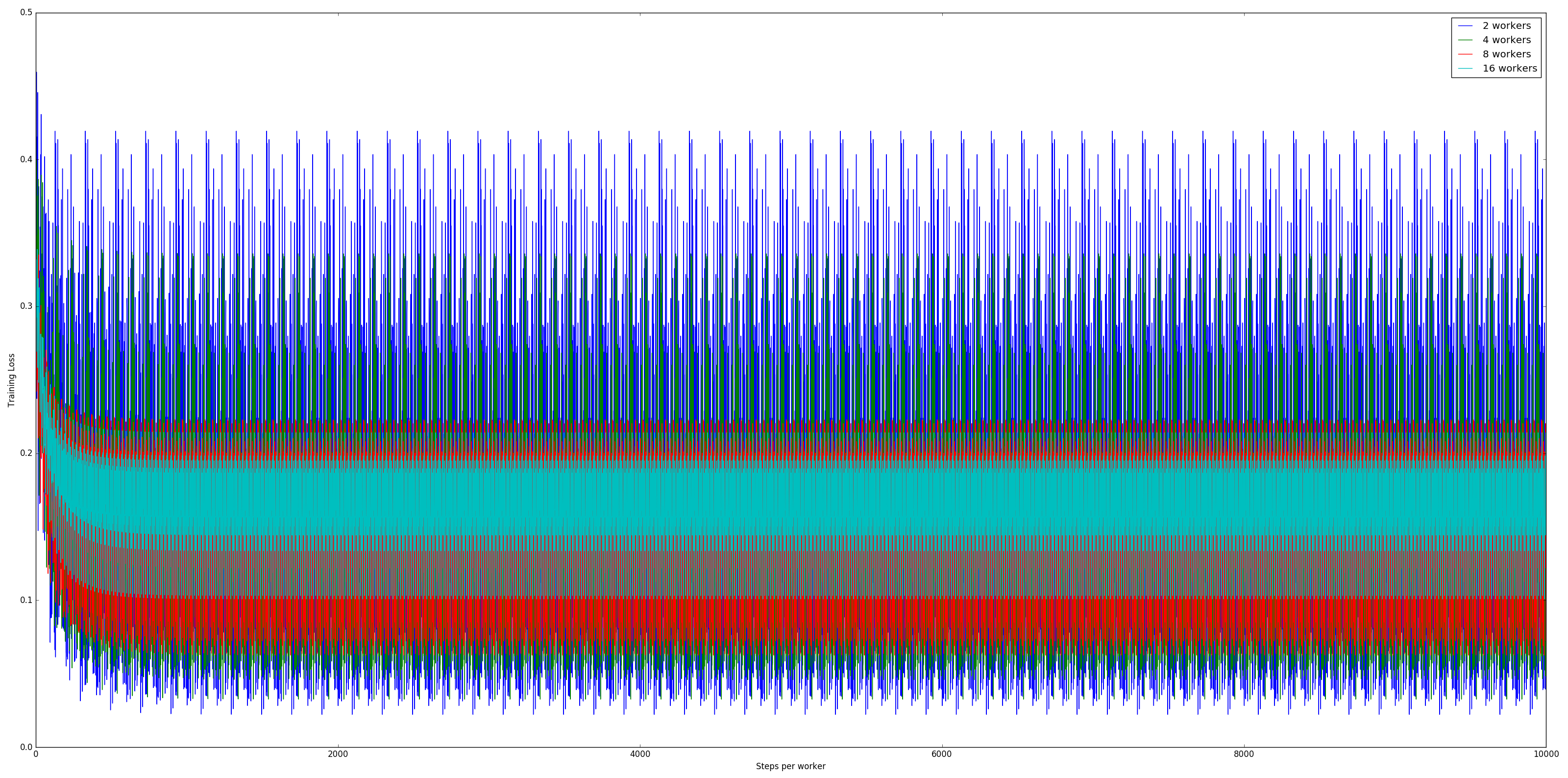


Fig. 2.1 Single Sigmoid Neuron distributed training loss plots over 2, 4, 8 & 16 nodes.

Eq. 3.1 Function subject to distributed learning by a single sigmoid Neuron

The distributed training was done over 10,000 steps per worker for different numbers of workers for each data parallel training strategy. The learning rate was kept constant at 0.5. Stochastic Gradient Descent was used as the optimization algorithm with no regularization for simplicity coupled with a sigmoid activation function.

The loss plots so obtained from subjecting the single sigmoid neuron to gradient averaged distributed training are presented in Fig. 2.1.

As is clearly evident, the convergence is slower with more workers; however, the loss is less noisy with more workers which corresponds with our observations from distributed CIFAR-10 experiments presented in section 6.

The noisy loss plot can be attributed to the fact that the function being learnt is too simple to be scaled across multiple compute nodes. Hence, visually illustrating that data-parallelism is most useful on complex mappings and/or on large datasets.

It is not unusual to notice that some optimization algorithms perform better with certain distribution schemes. For example, asynchronous Stochastic Gradient Descent scales better with lesser nodes whereas synchronous Stochastic Gradient Descent scales better with more nodes[2].

4. Mathematical explanation of Memory Re-alignment algorithm

An algorithm to minimize the number of matrix multiply operations by combining the weights and biases efficiently in to a single matrix was presented in the previous paper[3]. In this section, the exact mathematical explanation of its working and the GPU kernels along with their invocation patterns shall be presented.

Two types of re-alignment procedures are implemented namely ShiftRight and ShiftLeft which are explained in sections 4.1 & 4.2 respectively.

*4.2 ShiftRight Memory Realignment Matrix transform-*

In this scheme, the following transform is done, as shown in Fig. 4.2.1 –

Fig. 4.2.1 ShiftRight matrix transform illustration

Recall[3] that this when represented in the row-major memory layout as in the case of NVIDIA GPUs, actually looks like what is shown in Fig. 4.2.2 below –

Fig. 4.2.2 ShiftRight transformation actual memory layout illustration

The way this is done has been explained at a high level in [3]; here, the actual implementation details are presented. Let us make the following assumptions about the variable names used to explain the approach –

* rows – The original number of rows in the input matrix.
* cols – The original number of columns in the input matrix.
* d\_mat – GPU pointer to the memory location holding the original matrix in a row-major format as depicted on the LHS in Fig. 4.2.2.
* d\_helper – GPU pointer to the memory location to hold the values which may be damaged due to Write-After-Read race conditions[3].

Firstly, we initialize the d\_helper pointer to a memory location with a size of . Next, we populate the d\_helper array with values at the affected indices[3]. Ideally, we’d require threads to do this job as each thread simply performs a copy operation. However, to satisfy the constraint of launching threadblocks with number of threads being a multiple of the GPU warp size[4] (32), the number threadblocks of size to be launched will be as shown in Eq. 4.2.1.

Eq. 4.2.1 Number of threadblocks needed to populate d\_helper.

And we have Eq. 4.2.2 -

Eq. 4.2.2 Size of each threadblock to populate d\_helper

This operation in done by the following GPU kernel–

\_\_global\_\_ void ShiftRight\_PopulateHelper\_GPUKernel(float \*d\_mat, float \*d\_helper, int damaged\_elems, int rows, int cols) {

int idx = (blockDim.x \* blockIdx.x + threadIdx.x) % damaged\_elems;

int i = floor(0.5f \* (sqrt((float)1 + 8 \* idx) - 1.0f)) + 1;

int j = idx - i \* (i - 1) / 2;

int read\_idx = j + i \* cols;

d\_helper[idx] = d\_mat[read\_idx];

}

Mapping each thread to its respective element in the original matrix which its copying to the d\_helper array, if idx is the global threadIdx of a particular thread and assuming the 2D index (in row, column form) from where it shall read in the original matrix d\_mat as , we have the following relations as shown in Eq. 4.2.3 and Eq. 4.2.4 –

**4. Library Functionalities**

The library currently contains implementations for Convolutional and Fully connected Layers. They have been implemented as classes which can be instantiated to build a neural network architecture.

Care has been taken to ensure that all the memory allocation and initialization happens only once before training/inference is performed to extract maximum GPU performance.

Convolutional Layer-

The class implementing the convolutional layer has a constructor of the following form-

ConvLayer(

const cudnnHandle\_t &cudnn\_handle\_arg,

const cublasHandle\_t &cublas\_handle\_arg,

int num\_images\_arg,

int input\_channels\_arg,

int input\_h\_arg,

int input\_w\_arg,

int pad\_h\_arg,

int pad\_w\_arg,

int vert\_stride\_arg,

int hor\_stride\_arg,

int kernel\_h\_arg,

int kernel\_w\_arg,

int feature\_maps\_arg,

float learning\_rate\_arg = 1e-2f,

float momentum\_arg = 1e-3f,

float regularization\_coeff\_arg = 1e-2f,

regularizer\_type\_Conv regularizer\_arg = L2\_Conv,

float weight\_init\_mean\_arg = 0.0,

float weight\_init\_stddev\_arg = 0.5f);

The arguments accepted in the constructor are described below-

* cudnn\_handle\_arg – Handle to the cuDNN context used to execute functions from the cuDNN library.
* cublas\_handle\_arg – Handle to the cuBLAS context used to execute matrix operations from the cuBLAS library.
* num\_images\_arg – Number of images in the batch.
* input\_channels\_arg – Number of channels in each image if this is an input layer. If previous layer is a convolutional layer, this is the number of feature maps in the previous Convolutional layer. If the previous layer is a fully connected layer, this is the number of neurons in the previous layer.
* input\_h\_arg – Height or the number of rows in each image in the input batch.
* input\_w\_arg – Width or the number of columns in each image in the input batch.
* pad\_h\_arg – Vertical padding to be applied to the input images. This is the number of extraneous ‘0’ pixels appended on the top and the bottom edges of the input images.
* pad\_w\_arg – Horizontal padding to be applied to the input images. This is the number of extraneous ‘0’ pixels appended on the left and the right edges of the input images.
* vert\_stride\_arg – Vertical stride length of convolutional filter which slides across the images.
* hor\_stride\_arg – Horizontal stride length of convolutional filter which slides across the images.
* kernel\_h\_arg – Height or the number of rows of each convolution filter.
* kernel\_w\_arg – Width or the number of columns of each convolution filter.
* feature\_maps\_arg – Number of filters to learn or to use for inference.
* learning\_rate\_arg – Coefficient to be multiplied with the gradients before performing weight update during training.
* momentum\_arg – This is the fraction of the previous gradient which is subtracted from the current gradient before applying them to perform weight update during training.
* regularization\_coeff\_arg – Regularization coefficient or the fraction of the regularized weights to be applied during weight update during training.
* regularizer\_type\_Conv – This is an enum which dictates whether to apply L1 or L2 regularization during training.
* weight\_init\_mean\_arg – Mean of the Gaussian distribution from which the initial weights before training are determined.
* weight\_init\_stddev\_arg – Standard deviation of the Gaussian distribution from which the initial weights before training are determined.

Fully Connected Layer

The class implementing the convolutional layer has a constructor of the following form-

FCLayer(

const cudnnHandle\_t &cudnn\_handle\_arg,

const cublasHandle\_t &cublas\_handle\_arg,

int input\_batch\_size\_arg,

int input\_n\_arg,

int output\_n\_arg,

bool is\_softmax\_layer\_arg = false,

float learning\_rate\_arg = 1e-2f,

float momentum\_arg = 1e-3f,

float regularization\_coeff\_arg = 1e-3f,

regularizer\_type\_FC regularizer\_arg = L2,

float weight\_init\_mean\_arg = 0.0f,

float weight\_init\_stddev\_arg = 0.05f);

The arguments accepted in the constructor are described below-

* cudnn\_handle\_arg – Handle to the cuDNN context used to execute functions from the cuDNN library.
* cublas\_handle\_arg – Handle to the cuBLAS context used to execute matrix operations from the cuBLAS library.
* input\_batch\_size\_arg – Number of examples in the batch being input.
* input\_n\_arg – Number of neurons in the previous layer.
* output\_n\_arg – Number of neurons in the given layer.
* is\_softmax\_layer\_arg – Boolean value which is true if the layer is a Softmax layer.
* learning\_rate\_arg – Coefficient to be multiplied with the gradients before performing weight update during training.
* momentum\_arg – This is the fraction of the previous gradient which is subtracted from the current gradient before applying them to perform weight update during training.
* regularization\_coeff\_arg – Regularization coefficient or the fraction of the regularized weights to be applied during weight update during training.
* regularizer\_arg – This is an enum which dictates whether to apply L1 or L2 regularization during training.
* weight\_init\_mean\_arg – Mean of the Gaussian distribution from which the initial weights before training are determined.
* weight\_init\_stddev\_arg – Standard deviation of the Gaussian distribution from which the initial weights before training are determined.

The neural network can be simply defined by just a few lines of C++ code. An example is given below –

ConvLayer cl0(cudnnHandle, cublasHandle, BATCH\_SIZE, CHANNELS, DATA\_SIDE, DATA\_SIDE, 2, 2, 1, 1, 5, 5, 32);

cl0.SetPoolingParams(CUDNN\_POOLING\_AVERAGE\_COUNT\_INCLUDE\_PADDING, 3, 3, 2, 2, 0, 0);

cl0.SetActivationFunc(CUDNN\_ACTIVATION\_RELU);

cl0.is\_input\_layer = true;

ConvLayer cl1(cudnnHandle, cublasHandle, cl0.output\_n, cl0.output\_c, cl0.output\_h, cl0.output\_w, 2, 2, 1, 1, 5, 5, 32);

cl1.SetPoolingParams(CUDNN\_POOLING\_AVERAGE\_COUNT\_INCLUDE\_PADDING, 3, 3, 2, 2, 0, 0);

cl1.SetActivationFunc(CUDNN\_ACTIVATION\_RELU);

ConvLayer cl2(cudnnHandle, cublasHandle, cl1.output\_n, cl1.output\_c, cl1.output\_h, cl1.output\_w, 2, 2, 1, 1, 5, 5, 64);

cl2.SetPoolingParams(CUDNN\_POOLING\_AVERAGE\_COUNT\_INCLUDE\_PADDING, 3, 3, 2, 2, 0, 0);

cl2.SetActivationFunc(CUDNN\_ACTIVATION\_RELU);

FCLayer fcl0(cudnnHandle, cublasHandle, cl2.output\_n, cl2.output\_c \* cl2.output\_h \* cl2.output\_w, 64);

fcl0.SetActivationFunc(CUDNN\_ACTIVATION\_RELU);

FCLayer fcl1(cudnnHandle, cublasHandle, fcl0.input\_batch\_size, fcl0.output\_neurons, 32);

fcl1.SetActivationFunc(CUDNN\_ACTIVATION\_RELU);

FCLayer fcl2(cudnnHandle, cublasHandle, fcl1.input\_batch\_size, fcl1.output\_neurons, 10, true);

The code described above initializes a version of the famous AlexNet [1] to train on CIFAR-10 images [2]. The architecture is as follows –

Layer 0-

Convolutional layer which accepts BATCH\_SIZE number of images having resolution of 32x32 over RGB channels.

This layer pads the input image with by a factor of 2 vertically and horizontally.

The convolution filter windows slides across the image with a stride of 1 and the size of each filter is 5x5 over all 3 channels.

The number of filters or feature maps learnt by this layer is 32. This is succeeded by an average pooling layer with a ReLU activation function.

Layer 1-

Convolutional layer which accepts BATCH\_SIZE number of input examples. Each example has a dimension of cl0.output\_h X cl0.output\_w which are computed by the previous layer depending on the kernel size, convolution stride, padding, and the size and stride of the pooling layer if any. The number of input channels is equal to the number of feature maps learnt by the previous layer.

This layer pads the input image with by a factor of 2 vertically and horizontally.

The convolution filter windows slides across the image with a stride of 1 and the size of each filter is 5x5 over all 3 channels.

The number of filters or feature maps learnt by this layer is 32.

Layer 2-

Convolutional layer which accepts BATCH\_SIZE number of input examples. Each example has a dimension of cl1.output\_h X cl1.output\_w which are computed by the previous layer depending on the kernel size, convolution stride, padding, and the size and stride of the pooling layer if any. The number of input channels is equal to the number of feature maps learnt by the previous layer which is 32.

This layer pads the input image with by a factor of 2 vertically and horizontally.

The convolution filter windows slides across the image with a stride of 1 and the size of each filter is 5x5 over all 3 channels.

The number of filters or feature maps learnt by this layer is 64.

Layer 3-

Fully connected Layer with ReLU activations accepting inputs from the previous convolutional layer. Each example in the batch taken as input by this layer has a size which is the product of the number of feature maps, output width and output height. This layer has 64 outputs which are fed in to another fully connected layer.

Layer 4-

Fully connected Layer with ReLU activations accepting inputs from the previous Fully connected layer. Each example in the batch taken as input by this layer has a size which is the number of output neurons in the previous layer. This layer has 32 outputs.

Layer 5-

Fully connected Softmax Layer with ReLU activations accepting inputs from the previous Fully connected layer. Each example in the batch taken as input by this layer has a size which is the number of output neurons in the previous layer. This layer has 10 outputs which is the number of classes in the CIFAR-10 dataset.

The training loop code pertaining to this is also simple and is presented below –

while (1) {

readBatch(fp, x, y);

cl0.LoadData(x, false);

cl0.Convolve();

cl1.LoadData(cl0.d\_out, true);

cl1.Convolve();

cl2.LoadData(cl1.d\_out, true);

cl2.Convolve();

fcl0.LoadData(cl2.d\_out, true);

fcl0.ForwardProp();

fcl1.LoadData(fcl0.d\_out, true);

fcl1.ForwardProp();

fcl2.LoadData(fcl1.d\_out, true);

fcl2.ForwardProp();

fcl2.ComputeSoftmaxGradients(y);

fcl1.ComputeLayerGradients(fcl2.d\_prev\_layer\_derivatives);

fcl0.ComputeLayerGradients(fcl1.d\_prev\_layer\_derivatives);

cl2.ComputeLayerGradients(fcl0.d\_prev\_layer\_derivatives);

cl1.ComputeLayerGradients(cl2.d\_prev\_layer\_derivatives);

cl0.ComputeLayerGradients(cl1.d\_prev\_layer\_derivatives);

fcl2.UpdateWeights(fcl2.d\_gradients);

fcl1.UpdateWeights(fcl1.d\_gradients);

fcl0.UpdateWeights(fcl0.d\_gradients);

cl2.UpdateWeights(cl2.d\_filter\_gradients, cl2.d\_bias\_gradients);

cl1.UpdateWeights(cl1.d\_filter\_gradients, cl1.d\_bias\_gradients);

cl0.UpdateWeights(cl0.d\_filter\_gradients, cl0.d\_bias\_gradients);

}

In the code snippet shown, a training loop has been implemented. As is clearly evident, the first block of code does the forward propagation part taking the outputs of the previous layer and computing output activations. The second block starts with the computation of output derivatives of the softmax layer by using the training labels in one-hot encoded format. The line fcl2.ComputeSoftmaxGradients(y) does this part. This basically does an element-wise subtraction of the classifier softmax output and the training label one-hot encoded matrix pertaining to the batch and scales it by dividing the result with the batch size.

Next, each layer (including the softmax) use the input derivatives from the arguments to compute their respective gradients for the weights and biases. The gradients can be computed by firstly doing an element-wise multiply of the layer’s local derivatives of the output activations and the derivatives received from the layer in front. Do note that both these matrices have dimensions BATCH\_SIZE x OUTPUT\_NEURONS. Subsequently, a dot product of the resulting matrix and the input data matrix containing the activations from the previous layer (with a 1’s column appended to the left) is computed which gives the gradients for the weights and biases of the given layer.

Take note that the data matrix pertaining to a layer has dimensions BATCH\_SIZE x (INPUT\_NEURONS + 1) (the +1 is the extra 1’s column for biases). Now when the dot product of the transpose of the data matrix and the previous layer derivatives are taken, a matrix with dimensions (INPUT\_NEURONS + 1) x OUTPUT\_NEURONS results which is consistent with the dimensions of the weight matrix of each layer. Each gradient maps element-wise to its corresponding weight and hence for the weight update operation, a simple scaled (by the learning rate) in-place subtraction of the layer weights is performed.

**5. Intermediate convolution layer outputs**

This section gives a visual idea as to how the convolutional layer performs and what transformations are done to an input image of resolution 1920 x 1080 by this layer. The input images used for this illustration are as follows –



Let us term them & respectively.

The result of convolution operations on Red, Green and Blue channels with a 50x50 kernel filled with 1’s with a horizontal and vertical stride of 1 and 0 padding are presented below (1 output feature map)

Red Channel –





Blue Channel –





Green Channel –





All Channels –

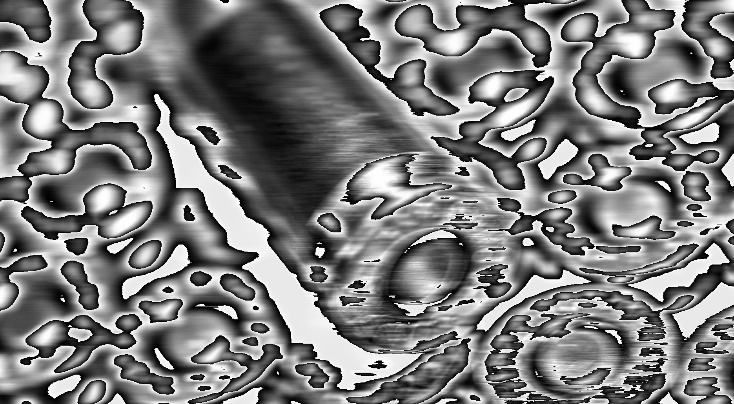




Do note that all of these convolutions have been done with a 0 bias. Experiment results with different bias values are presented as follows –

Bias = -20,000





Bias = +20,000





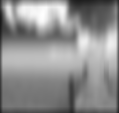
As is evident, a washout effect is visible with high positive bias and artifacts start to appear with high negative bias.

Multiple experiments were also carried out to test max pooling. The results are as follows with a 50x50 pooling window with no padding –





The results appear to be whitened at multiple spots which is in line with the max pooling function heuristically verifying the working. The same experiments were also repeated with hand designed matrices and results were found to correspond with the expected outputs. The result of applying 2 convolutions (1st one with max pooling) are as follows –





Furthermore, to monitor what the network has learnt, the following formula can be used to map the output activations on to the pixel space –

This equation corresponding to 2 fully connected layers with the input layer consisting of features ranging from to and the output layer consisting of neurons & the activations of each of these are denoted by

is the activation function being used at each neuron (Sigmoid, ReLU etc); therefore is the inverse of the activation function.

is the weight of the link connecting input neuron & output neuron.

is the final activation output of output neuron.

is the input from the input neuron.

This function basically helps visualize the what a given network has learnt for a given classification. Therefore, if we would like to map the learned weights to pixel space for a given label , we can set and the others to and compute this function for every for and visualize what the network “imagines” to be that class.

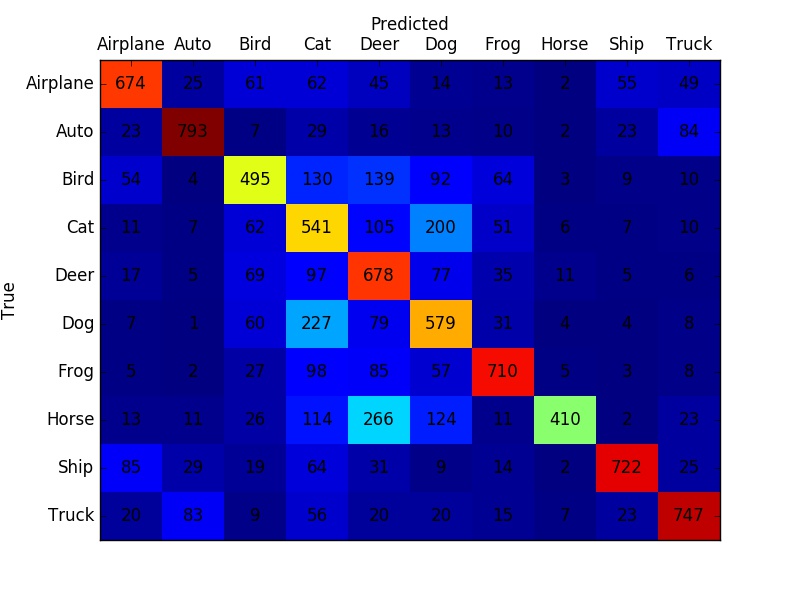
Do note that this is only for two layers. For deeper networks, this function may be performed in a chained fashion to extend to deeper networks.

**6. Results**

Tests were carried out to train on the CIFAR-10 dataset with a batch size of 128. The neural network architecture used is the same as the AlexNet type design explained in Section 4. The following results were obtained –

Test Accuracy = 63.49 %

Confusion matrix –



The training was done over epochs consisting of 59,904 images each. Testing was done over 10,000 images. The CIFAR-10 dataset consists of images from the following classes –



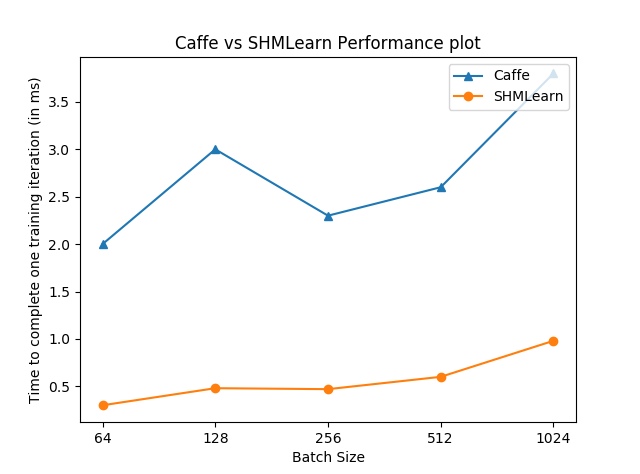
The library training speed revolved around 0.0045 seconds per batch. The same training on caffe yielded a speed of around 0.2 seconds per batch. The hardware used involved a NVIDIA GTX 1070 GPU (mobile).

However, the library currently lacks support for distributed deployment and the support for other training strategies apart from stochastic gradient descent like Adagrad, Adam etc. Also, as of now, it supports only convolutional and fully connected layer implementations.

The speedup can mostly be attributed to factors from the parallel design patterns and from the removal of unnecessary overhead incurred in the libraries.

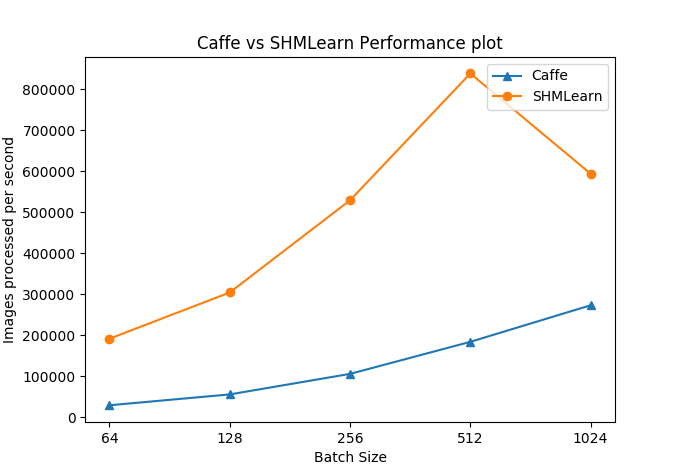
The performance results from training on MNIST training are presented below. The training conditions are as follows –

60,000 images of the handwritten digits from 0 to 9 were used in one training epoch with different batch sizes of 64, 128, 256, 512 & 1024 utilizing Stochastic Gradient Descent as training strategy. A constant learning rate of 0.05 with 0.01 regularization strength (L2) and 0.0 momentum was utilized. The GPU used is a GTX 1070 (laptop edition) with 8 GB of GPU memory and the CPU used is an Intel core i7 6700-HQ with 16 GB of RAM. The neural network trained is a 2 layer fully connected neural network with 64 hidden neurons with sigmoid activation and softmax output. The performance plots are presented as below –



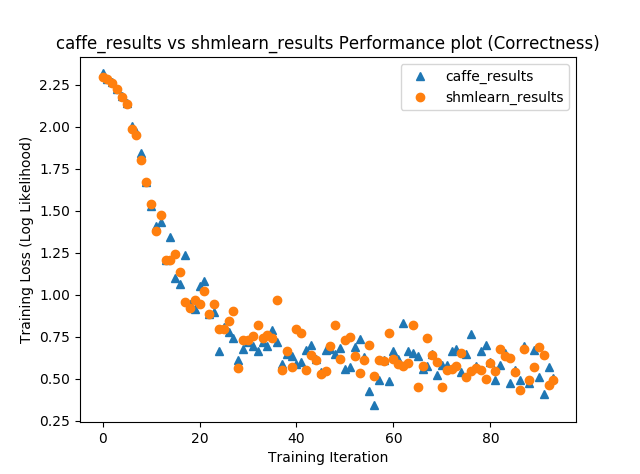
The plot above demonstrates strong scaling or the time taken to complete a constant amount of work (one training iteration). Here, the comparison is made with an equivalent implementation in Caffe with the GPU. As can be inferred from the plot above, the library provides an average speedup of around 6x for this network.

The Number of images processed / sec vs batch size plot is presented below –



This plot demonstrates the amount of work (Number of training iterations) completed in a constant time (1 second). An average speedup of around 6x can be noticed here.

The correctness of the outputs produced by the library is explicated by the plot below which shows how the loss changes with iterations as compared to the same caffe implementation. The second plot is the same plot with time as X-axis.



**7. Future Work**

The future roadmap primarily includes support for execution in distributed environments leveraging MPI support. MPI support with GPUDirect would be explored as it provides significant performance gains by allowing the passing of GPU pointers directly to MPI methods which bypass the data from the host and direct communicate GPU data between nodes.

Furthermore, other novel training strategies will be explored by researching different training schemes and strategies.

**8. References**

1. A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In NIPS, 2012.
2. Tal Ben-nun, A CUDNN minimal deep learning training code sample using LeNet, <https://github.com/tbennun/cudnn-training/>
3. Andrej Karpathy et. Al., CS231n: Convolutional Neural Networks for Visual Recognition, Stanford University.
4. A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. 2009
5. Y. Lecunn, C. Cortes, MNIST dataset, <http://yann.lecun.com/exdb/mnist/>