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

Souham Biswas
Illinois Institute of Technology
sbiswas7@hawk.iit.edu

Abstract

Most Deep Learning frameworks available today (TensorFlow, Caffe, CNTK etc) involve a pythonic interface to experiment with, or deploy deep learning algorithms. However, little effort has been made towards developing GPU-optimized frameworks with native C++ support. This library leverages support of cuDNN (CUDA Deep Neural Network) and cuBLAS (CUDA Basic Linear Algebra Subprograms) packages by NVIDIA to provide for a highly-optimized GPU powered deep learning library without the overhead of pythonic interfaces. Maximum care has been taken to provide a clean C++ interface which closely parallels the convenience provided by pythonic interfaces of most deep learning frameworks while providing for greater control of model architecture and functionality.

1. Introduction

The main strategy for performance optimization followed involves reduction of most operations to intelligent matrix multiply operations of the form $C = \alpha(A \cdot B) + \beta C$ where A & C are matrices and α, β are scalars. This strategy stems from the fact that cuBLAS (most BLAS packages in general) contains optimized primitives to perform this type of operation. The library at present, includes the implementation of Convolutional & Fully connected Layers with support for ReLU & Sigmoid Activation Functions, Max & Average Pooling, L1 & L2 Regularization.

A crucial factor which often poses a bottleneck for GPU-powered applications like this one include memory access operations between the host (CPU) and the device (GPU). This has been minimized by keeping all the training and inference related variables on the GPU by allocating and initializing them only once in the beginning of any inference or training procedure.

The remaining part of the paper is organized in the following way: Section 2 shall discuss about the various optimization design patterns implemented in this library which contribute to performance improvement on GPUs. Section 3 discusses the memory layout of the data which is used as the norm for all operations and functions implemented in SHM-Learn. Section 4 gives a brief overview of the functionalities provided in the library and provides usage examples. Section 5 briefly discusses the intermediate outputs produced by the convolutional layers over some sample images. Section 6 presents characteristics and plots of test benchmarks and a discussion of the performance of the library and its present limitations. Finally, section 7 cites the future work.

2. Optimization Design Patterns

Various design patterns to improve upon the training and inference performance were implemented. These design patterns are mostly specific to NVIDIA GPU architectures and hence, yield performance boosts when run on GPU hardware. They are mentioned as follows –

1. Reduction to $C = \alpha(A \cdot B) + \beta C$ form
   In most BLAS packages (Intel MKL, cuBLAS, etc) the matrix multiplication

subroutine is popularly known as GEMM which stands for General Matrix-Matrix Multiply. These subroutines for the most part, have a function signature of the form $C = \alpha(A \cdot B) + \beta C$. These primitives are heavily optimized. Therefore, most mathematical operations (like gradient computations, weight updating etc) have been implemented as a composition of this operation. For example –

To compute gradients at the softmax layer, the following steps need to be taken theoretically –

For a softmax output at index k, gradient vector G_1^k must be computed per the following formula –

$$G_1^k = \{s_k^i - 1(k = c)\ \forall\ i \in [0, m)\}$$

s_k^i is the softmax output of output neuron at index k for input example at index i in the batch.

m is the number of examples in the batch currently being trained on.

c is the correct class label (index of correct output neuron which should fire the strongest) of example i within the batch.

$1(k = c)$ is 1 only when k equals c and 0 otherwise.

Subsequently, to find out the gradient for weight w_j^k which is the weight of the connection between neuron j in the previous layer and neuron k in the softmax layer, we need to compute the sum of product of vectors G_k^1 and X_j where X_j is the vector containing the output activations of neuron j in the previous layer for all examples in the batch. Finally, this quantity needs to be divided by m to obtain the final gradient for w_j^k. This can be achieved in multiple ways –

- *Method 1-*
  We can selectively subtract 1 from the output matrix containing softmax outputs for each example in the batch from the output at indices corresponding to the class labels from the training data to obtain G_1^k.
  Following this, we do a matrix multiply between G_1^k previously obtained and X_j to finally obtain the gradient $(X_j^T \cdot G_1^k)$.

- *Method 2-*
  We perform the following operation –
  $$\frac{1}{m}\left(X^T \cdot Y + X^T \cdot S\right)$$
  It turns out, that this computation is much faster than the former as the work has been composed into a series of matrix-matrix operations which have highly optimized implementations in cuBLAS.

2. <u>GPU Kernels designed to support invocation as multiples of GPU Warp size</u>
   A warp is a collection of GPU threads which are executed simultaneously on a given SM (Streaming Multiprocessor) at one time. Therefore, for maximum GPU throughput, it has been observed that launching of GPU kernels divided into equal sized thread blocks, with each thread block's size as a multiple of the warp size (32 threads for NVIDIA GPUs) provides significant performance gains

even if this involves some threads doing redundant work. A simple example shall illustrate this further –

Suppose we have to fill an array of size 38 with 1's; we have two options –

- Launch 1 thread block with 38 threads with each thread setting its respective array cell whose index is computed using the `threadIdx` directly.
- Launching 2 thread blocks with 32 threads each with each thread setting index `(blockIdx * blockDim + threadIdx) mod 38`. This way, all threads with `(blockIdx * blockDim + threadIdx) >= 38` ranging from index 38 to 63 will be setting array indices 0 to 25 once again.

Even though it might appear that case 1 should be faster as it is doing lesser work (1 thread block with 38 threads vs 2 thread blocks with 32 threads each for a total of 64 threads), it is observed that case 2 is faster than case 1. The reason behind this is because in case 2, every thread block is executed on a SM (Streaming Multiprocessor); therefore, in the second case, 2 SMs are used simultaneously and each thread block launches 1 warp (32 threads) each and all of this happens simultaneously. Whereas, in case 1, there is 1 thread block which must execute on 1 SM; now since the number of threads in the thread block isn't a multiple of 32 (warp size), the first 32 threads are launched while the remaining 6 threads wait for their completion which are launched later.

Therefore, care has been taken to ensure similar GPU kernel invocation semantics wherever possible.

3. Minimization of Thread Divergence in GPU kernels

   GPUs are SIMD (Single Instruction Multiple Data) devices. This means, their instruction set allows for assembly instructions which perform the same operation on multiple data elements simultaneously. Therefore, they are most efficient when there is minimal conditional branching and when the GPU kernels follow a similar path of execution.

   Therefore, the GPU kernels have been designed such that code-branching is minimum; even if that implies some redundant computation.

4. Bias related computations combined with weight matrix computation to reduce matrix-multiply operations

   The bias related matrix operations have been combined with weight matrix computations by efficiently appending a column of 1's to the input of a layer. This problem is much harder than it seems when performed on an NVIDIA GPU. It is explained as follows –

   We have the following matrix which we want to transform the following way –

$$\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & a_0 & a_1 \\ 1 & a_2 & a_3 \end{bmatrix}$$

   On a GPU, every matrix is represented as a linear flattened array. So, the problem depicted in 2D above for illustration actually translates into the following representation visually –

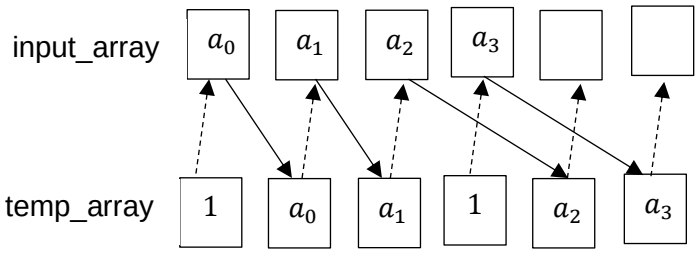
$$[a_0, a_1, a_2, a_3] \rightarrow [1, a_0, a_1, 1, a_2, a_3]$$

Name: Souham Biswas **CWID**: A20365242 **Email**: sbiswas7@hawk.iit.edu

Please note, here it is assumed that the pointer to input array has sufficient space to contain the resulting expanded matrix with the 1's column.

This can be tackled in two ways-

a. Temporarily storing whole of the original array in a temporary array, making the necessary shifts in the temporary array and writing the new result back onto the input array. It is represented below –



Solid arrows are copy operations and dashed operations are write operations. This requires the whole matrix to be duplicated; and input matrices in any layer can grow very quickly with features and batch size. Therefore, this severely limits the size of the input matrix this method can handle as GPU memory is limited. It is desired that the work be split up into thread blocks to comply with design pattern explained at 2.2, which states that work should be divided in to thread blocks, where each executes a multiple of the warp size number of threads, even if some threads perform redundant work for maximum performance.

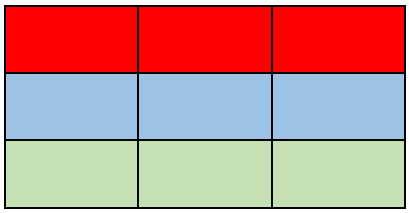
b. The second method consists of two steps, is scalable to any work size (divides input in to thread blocks) and uses much lesser GPU memory

However, this type of algorithm is feasible only when the following condition holds –

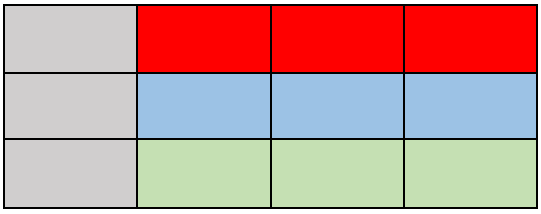
$$\frac{2n}{m+1} > 1$$

Here, n is the number of features, or columns of the input matrix and m is the batch size or the number of rows of the input matrix. It's derivation will be presented in the text. The outline of this method is presented below –

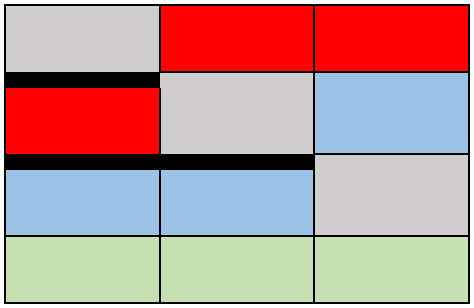
Suppose we have the following matrix with m rows and n columns where each row has been color coded with red, blue and green colors –



We wish to transform this into the following-



The grayed out column represents 1's. When the original matrix is modified to incorporate the same transformation while keeping the same number of columns, the following pattern emerges-



This representation captures the locations in the matrix (represented by bands at top) which could have been non-deterministically changed by the multiple GPU threads and produced Write-After-Read race conditions had a naïve in-place shift been implemented to append the ones column.

However, it can also be noticed that if we somehow stored *these* elements instead of the entire matrix in a temporary location, we can get the 1 vector appended matrix by simply reading from the temp location while writing these elements back into the input array in these locations. As far as the other elements are concerned, they are not being over-written so they can be moved in-place.

The derivation of the $\frac{2n}{m+1} > 1$ rule is presented as follows. Suppose the input matrix has a dimension of m rows and n columns. Therefore, in the naïve case, the memory complexity would be $O(mn)$. In the improved case, the memory complexity would be $O(m^2)$ or $\theta\left(\frac{m(m+1)}{2}\right)$. Therefore, solving the following inequality gives us the solution –

$$\frac{m(m+1)}{2} < mn \Rightarrow \frac{2n}{m+1} > 1$$

Also, it can further be noted that $\frac{2n}{m+1}$ actually denotes the extent of memory usage savings we are achieving when we use this method.

3. Memory Layout

The cuDNN library has two modes of representations for the data it operates on, which is highly tailored to be compatible with image type data. The two modes are –

- NCHW – Number of Images, Channels, Height, Width
- NHWC – Number of Images, Height, Width, Channels
- CHWN – Channels, Height, Width, Number of Images

The mode utilized in this library is NCHW as it is stated as the most efficient for NVIDIA GPUs.

The memory representation of an image batch of b images with each image having c channels, with h height and w width in GPU memory is as follows (Here $\left(p_{chan}^{i,j}\right)_{batch}$ corresponds to a pixel at i^{th} row, j^{th} column in channel index $chan$ of an image at index $batch$ in the batch; all indices are 0-indexed) –

$$\begin{gathered}
(p_0^{0,0})_0, (p_0^{0,1})_0 \ldots, (p_0^{0,w-1})_0, (p_0^{1,0})_0, (p_0^{1,1})_0 \ldots, (p_0^{h-1,w-1})_0, \\
(p_1^{0,0})_0, (p_1^{0,1})_0 \ldots, (p_1^{0,w-1})_0, (p_1^{1,0})_0, (p_1^{1,1})_0 \ldots, (p_1^{h-1,w-1})_0, \\
\vdots \\
(p_{c-1}^{0,0})_0, (p_{c-1}^{0,1})_0 \ldots, (p_{c-1}^{0,w-1})_0, (p_{c-1}^{1,0})_0, (p_{c-1}^{1,1})_0 \ldots, (p_{c-1}^{h-1,w-1})_0, \\
\vdots \\
\vdots \\
(p_0^{0,0})_1, (p_0^{0,1})_1 \ldots, (p_0^{0,w-1})_1, (p_0^{1,0})_1, (p_0^{1,1})_1 \ldots, (p_0^{h-1,w-1})_1, \\
(p_1^{0,0})_1, (p_1^{0,1})_1 \ldots, (p_1^{0,w-1})_1, (p_1^{1,0})_1, (p_1^{1,1})_1 \ldots, (p_1^{h-1,w-1})_1, \\
\vdots \\
(p_{c-1}^{0,0})_1, (p_{c-1}^{0,1})_1 \ldots, (p_{c-1}^{0,w-1})_1, (p_{c-1}^{1,0})_1, (p_{c-1}^{1,1})_1 \ldots, (p_{c-1}^{h-1,w-1})_1, \\
\vdots \\
\vdots \\
\vdots \\
(p_0^{0,0})_{b-1}, (p_0^{0,1})_{b-1} \ldots, (p_0^{0,w-1})_{b-1}, (p_0^{1,0})_{b-1}, (p_0^{1,1})_{b-1} \ldots, (p_0^{h-1,w-1})_{b-1}, \\
(p_1^{0,0})_{b-1}, (p_1^{0,1})_{b-1} \ldots, (p_1^{0,w-1})_{b-1}, (p_1^{1,0})_{b-1}, (p_1^{1,1})_{b-1} \ldots, (p_1^{h-1,w-1})_{b-1},
\end{gathered}$$

$$\vdots$$
$$(p_{c-1}^{0,0})_{b-1}, (p_{c-1}^{0,1})_{b-1} \ldots\ldots, (p_{c-1}^{0,w-1})_{b-1}, (p_{c-1}^{1,0})_{b-1}, (p_{c-1}^{1,1})_{b-1} \ldots\ldots, (p_{c-1}^{h-1,w-1})_{b-1}$$

Do note that all the pixels represented above are laid out in a flattened 1D array in GPU memory.

This format of data representation isn't only limited to image data representation; that is, any data can be represented in the NCHW format. For example, the outputs of a fully connected layer for all the inputs in a batch can be represented by a tensor with –
$N = batch\ size,$
$C = Number\ of\ output\ neurons,$
$H = 1$
$W = 1$

Therefore, this data representation format is compatible with a wide range of types of data.

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_INC
LUDE_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_INC
LUDE_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_INC
LUDE_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_derivat
ives);
fcl0.ComputeLayerGradients(fcl1.d_prev_layer_derivat
ives);
cl2.ComputeLayerGradients(fcl0.d_prev_layer_derivati
ves);
cl1.ComputeLayerGradients(cl2.d_prev_layer_derivativ
es);
cl0.ComputeLayerGradients(cl1.d_prev_layer_derivativ
es);

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.

Name: *Souham Biswas* ***CWID***: *A20365242* ***Email***: *sbiswas7@hawk.iit.edu*

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 I_1 & I_2 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)

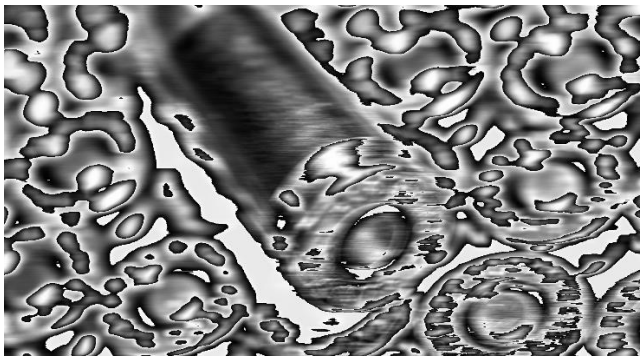
Red Channel –

Blue Channel –

Green Channel –

Name: *Souham Biswas* ***CWID***: *A20365242* ***Email***: *sbiswas7@hawk.iit.edu*

CS 597 – Final Report, Fall 2016

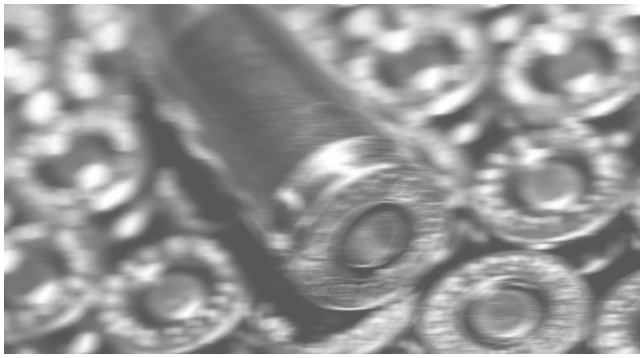


All Channels –

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



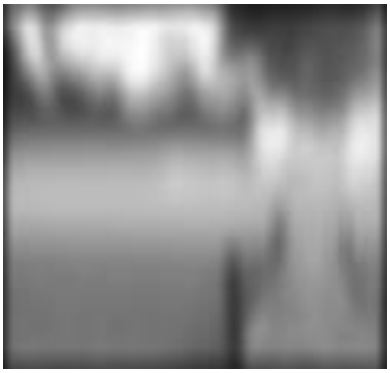
As is evident, a washout effect is visible with high positive bias and artifacts start to appear with high negative bias.

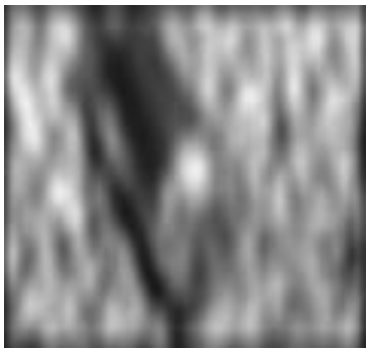
Name: *Souham Biswas* ***CWID***: *A20365242* ***Email***: *sbiswas7@hawk.iit.edu*

CS 597 – Final Report, Fall 2016

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 –

$$x_g = \sum_{i=0}^{n-1} \left(\frac{F^{-1}(O_i)}{\sum_{k=0}^{f-1} w_k^i} \right) w_g^i$$

$$\forall\ 0 \leq g \leq f-1$$

This equation corresponding to 2 fully connected layers with the input layer consisting of f features ranging from x_0 to x_{f-1} and the output layer consisting of n neurons & the activations of each of these are denoted by $O_0 \dots \dots O_{n-1}$

$F(u)$ is the activation function being used at each neuron (Sigmoid, ReLU etc); therefore $F^{-1}(u)$ is the inverse of the activation function.

w_k^i is the weight of the link connecting k^{th} input neuron & i^{th} output neuron.

O_i is the final activation output of i^{th} output neuron.

x_g is the input from the g^{th} 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 i, we can set $O_i = 1.0$ and the others to 0.0 and compute this function for every x_g for $g \in [0, f-1]$ and

visualize what the network "imagines" to be that class.

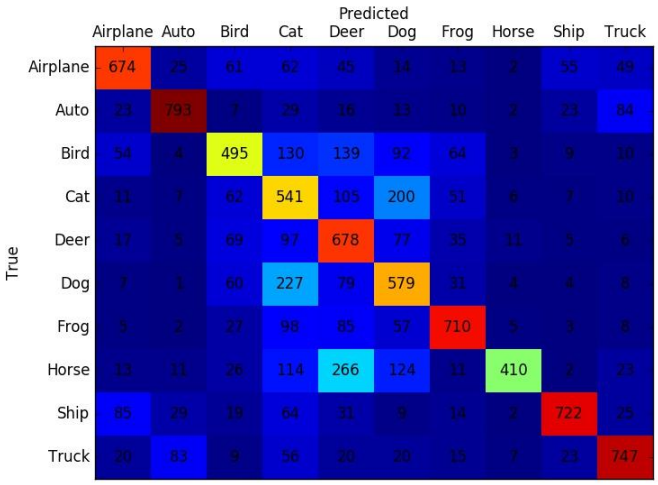
Do note that this is only for two layers. For deeper networks, this function may be performed in a chained fashion to extend to deeper networks.

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 –



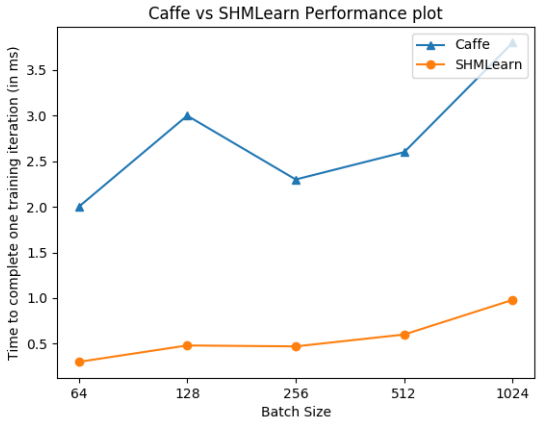
The training was done over epochs consisting of 59,904 images each. Testing was done over 10,000 images. The CIFAR-10 dataset consists of images from the following classes –

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 –

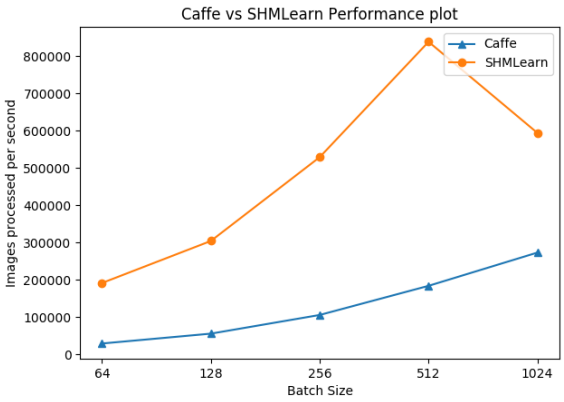


Name: *Souham Biswas* ***CWID***: *A20365242* ***Email***: *sbiswas7@hawk.iit.edu*

CS 597 – Final Report, Fall 2016

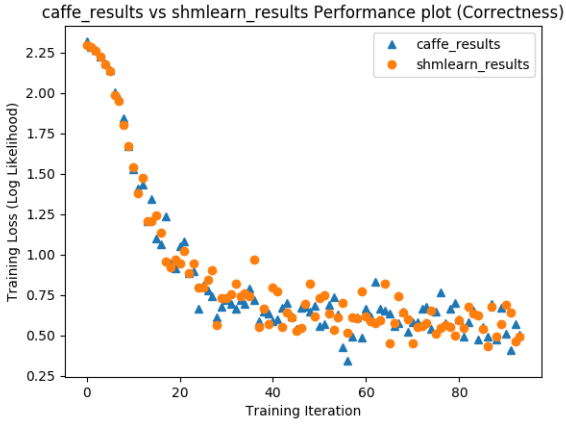
The plot above demonstrates strong scaling or the time taken to complete a constant amount of work (one training iteration). Here, the comparison is made with an equivalent implementation in Caffe with the GPU. As can be inferred from the plot above, the library provides an average speedup of around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

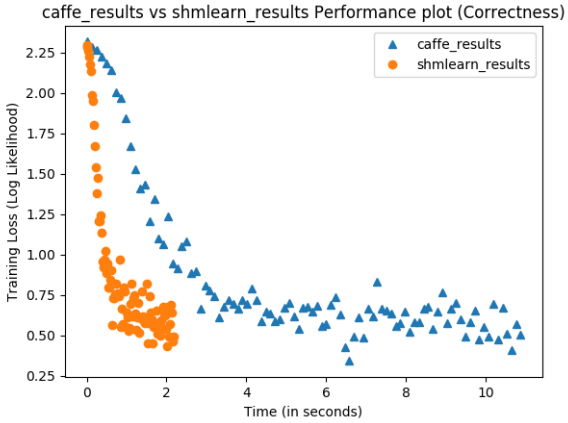
The Number of images processed / sec vs batch size plot is presented below –



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/