CME 213, Introduction to parallel computing
Eric Darve
Winter 2020

Stanford University

Neural Networks on CUDA

Part I: Introduction, grading, and tutorial

In this project, you will implement a neural network using CUDA to identify digits from hand-written images (a specific case of image classification problem). Neural networks are widely used in machine learning problems, specifically in the domains of image processing, computer vision and natural language processing. Recently, there has been a lot of excitement regarding deep learning, which basically uses more advanced variants of the simpler neural network (NN) we cover here. Therefore, being able to train large neural networks efficiently is important and is the goal of this project.

The main purpose of this Part I is to introduce you to the project and give you a picture of what needs to be done. Details of the starter code, submission instructions and full details of the grading steps will be handed out in Part II.

1 Schedule and Grading

Date	Percent	Description
Friday March 6th 11 PM	20%	Preliminary Report
Wednesday March 18th 11 PM	80%	Final Report

Late days: For the preliminary report, you have one late day as usual. For the final report, there are **no** late days. So please make sure you submit your work on time.

Preliminary Report: The preliminary report consists of a written report of **1–2 pages,** along with MPI + CUDA computer code. The grading basis for the preliminary report is as follows:

- **Correctness (15%):** We expect a correct, fully working MPI+CUDA parallel code. However you do not need to optimize your code. Submit the simplest code that will give the correct result. The details about how we determine correctness are provided in Part II.
- **Profiling (5%):** Perform a profiling of your unoptimized code. Identify major bottlenecks in your code and make suggestions for what you think is causing these bottlenecks and how you would try to alleviate them. You do not need to write computer code for any optimization. We will not test your code for performance.

Final Report: For the final report, you will have to turn in a **4-page report** and a fully working and optimized code. Your grade will be determined as follows:

- **Correctness (32%):** This is for correctness of your optimized code. The details about how we determine correctness are provided in Part II.

- **Correctness analysis (3%):** In addition to the code, in the report, discuss the correctness of your code. Does the GPU code give the same results as the CPU code? How large is the difference? How does this difference depend on the parameters of your calculation? How can you explain this difference?
- **Performance (20%):** This grade will be based on the performance of your code for a number of specific benchmarks. See Part II.
- **Profiling and analysis (20%):** the results are presented in the final report. Your tasks include:
  - Perform a detailed profiling of your optimized code. Include plots, charts, and data visualization from your profiling. Discuss the performance of your code before and after optimizations. What are the key bottlenecks? What is slowing down your code? How does the performance of your code depend on various parameters, like input data size, and number of neurons?
  - Explain the steps you have taken to improve performance. What bottleneck in the hardware was addressed by your algorithmic changes? Can you explain the impact of the changes you made in the code?
- **Overall quality of report (5%):** this criterion will evaluate the quality of the report, the clarity and correctness of the explanations, the organization and presentation, the completeness of the explanations, and the overall quality of the project.
- **Bonus assignment (+ 15 points):** There is an extra credit part to the project worth 15 points. This is described in Section 4.3. If you decide to do the bonus assignment, use an extra half page to describe your approach and findings. The final report will then be 4 ½ pages.

2 Data and Notation

We will be using the MNIST [1] dataset, which consists of greyscale 28 × 28 pixel images of handwritten digits from 0 to 9. Some examples of this dataset are shown in Figure 1.

Figure 1: Examples of MNIST digits

The dataset is divided into two parts: 60,000 *training* data and 10,000 *test* data. We will use the *training* data to learn the parameters of our neural network (described later), and the *test* data to measure the performance of the learned network on an unseen dataset.

In the training process, one issue is overfitting on the training data. To avoid this, a common practice is to perform a cross-validation—a technique to measure how the model will generalize on

an unseen dataset. Cross-validation is carried out by further dividing the *training* data into two sets, a *training* set and a *validation* set. The *validation* set is a small portion (usually 10%) of the *training* dataset. We then perform the training on the *training* set (excluding the *validation* set) and evaluate our model on the *validation* set. There are different types of cross-validation, and we will only do a single holdout for validation because of computational issues. We use insights from this validation to improve our model.

The *test* data is used to evaluate our tuned model on the unseen data. It is not meant to be used in the training process.

We denote the i^{th} image sample in the *training* set as $(x^{(i)}, y^{(i)})$, where $x^{(i)}$ denotes the image and $y^{(i)}$ denotes the class label, which is the digit shown in the image.

3 Neural Networks

3.1 Neurons

To describe neural networks, we will begin by describing the simplest neural network, one which comprises a single "neuron." Figure 2 illustrates a single neuron.

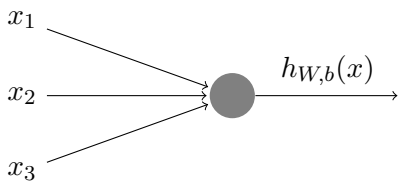

Figure 2: Single Neuron

This "neuron" is a computational unit that takes as input (x_1, x_2, x_3), and outputs

$$h_{W,b}(x) = f(Wx + b) = f\Big(\sum_{i=1}^{3} W_i x_i + b\Big)$$

where $f : \mathbb{R} \to \mathbb{R}$ is the activation function, W is the weights of the neuron, and b is the bias term. (W, b) together form the parameters of this neuron.

For this project, we set $f(\cdot)$ to be the sigmoid function:

$$f(z) = \frac{1}{1 + \exp(-z)}$$

Other common activation functions include $f(z) = \tanh(z)$ and the rectified linear unit (ReLU), $f(z) = \max(0, z)$. These are illustrated in Figure 3.

This is a good time for us to discuss the calculation of the derivative of the sigmoid function with respect to its input, since we are going to use it significantly in the following sections.

$$f(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$
$$\frac{\partial \sigma(x)}{\partial x} = -\frac{1}{(1 + \exp(-x))^2} \frac{\partial \exp(-x)}{\partial x} = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \sigma(x)(1 - \sigma(x))$$

The neuron performs a linear transform on its input followed by a non-linear transformation (sigmoid, in this case).

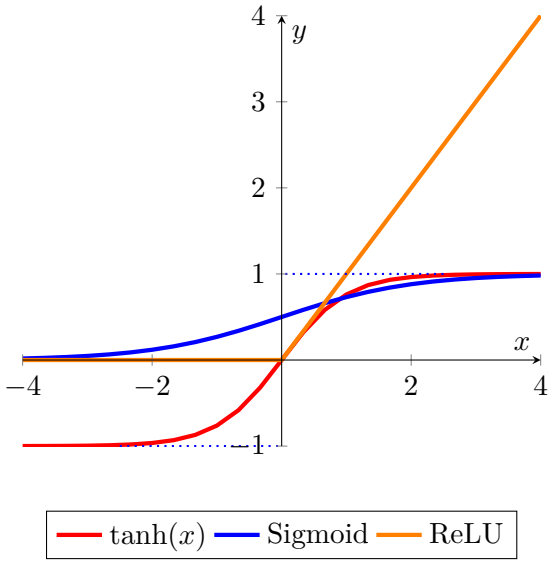

Figure 3: Examples of three activation functions: $\tanh(x)$, $1/(1+\exp(-x))$ (sigmoid), and the rectified linear unit (ReLU).

When a single neuron is used, we are limited to a binary classification. Take the example of a cancer tumor. We may build a neural network that takes as input the size of the tumor, its location, and the length of time it has been there. Based on these three pieces of information, the network needs to determine whether the tumor is benign or malignant. This is a true/false type of determination. In our previous model using the sigmoid function, we can interpret the output of the network as follows: if $f(Wx+b) > 0.5$, we consider that the tumor is malignant, otherwise it is benign.

If you think about it, the value of $f(Wx+b)$ depends on the sign of $Wx+b$. What the network is doing is that it is partitioning the space x using a d-dimensional hyperplane of the input space. On one side of the hyperplane, $f(Wx+b) > 0.5$, and on the other $f(Wx+b) < 0.5$.

Our task is therefore to learn the parameters of the network W and b so that it achieves the best accuracy or precision on the *test* dataset.

In the project we need to extend this concept to multiple classes. Instead of a simple true/false output, we need to decide which digits from 0 to 9 is shown on the input image. This requires a full neural network.

3.2 Fully-connected feedforward neural network

Figure 4 shows a fully-connected feedforward neural network with 1 input layer, 1 hidden layer, and 1 output layer. We call such a network to be a two-layer neural network (ignoring the input layer as it is trivially present). Let us denote the input as $x \in \mathbb{R}^{1\times d}$, the number of neurons in layer i as H_i, and the parameters of layer i as $(W^{(i)}, b^{(i)})$.

In our problem, we are trying to determine the digit associated with each image. We will call this digit the "label" associated with the image (using the neural network terminology). The total number of labels is denoted C. In our case $C = 10$, since we are trying to determine digits 0 to 9.

Recall that the parameters of a single neuron are $W \in \mathbb{R}^{1\times d}$ and $b \in \mathbb{R}$, i.e., W is a vector of the same dimensionality as the input and the bias is simply a scalar. Therefore, we can represent the

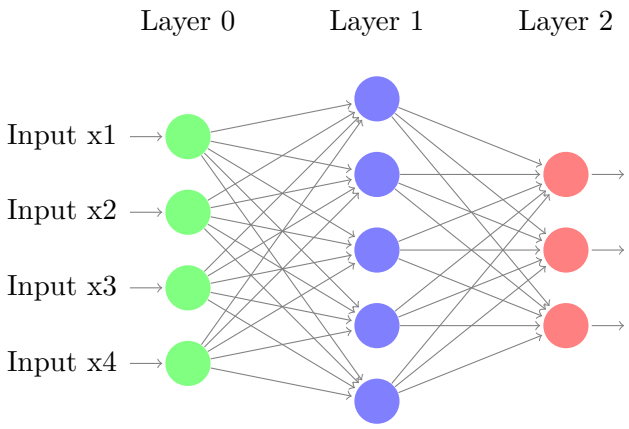

Figure 4: Fully Connected Feedforward Neural Network with 2 layers

parameters of layer i of the neural network in matrix form as $W^{(i)} \in \mathbb{R}^{H_i \times H_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{H_i \times 1}$. Similarly, if we had N input vectors, we denote the input collectively as $X \in \mathbb{R}^{N \times d}$.

In Figure 4, $d = 4$, $H_1 = 5$, $H_2 = 3$, $W^{(1)} \in \mathbb{R}^{5 \times 4}$, $b^{(1)} \in \mathbb{R}^{5 \times 1}$, $W^{(2)} \in \mathbb{R}^{3 \times 5}$, $b^{(2)} \in \mathbb{R}^{3 \times 1}$.

The last layer is special. This is the output of our network. In the project, we have $C = 10$ output nodes. Each node represent a possible digit. We will see later on how the output vector $\hat{y}$ can be interpreted to determine the digit that is predicted by the network for an input image.

3.3 Feed-forward

The nice thing about neural networks is that they are highly modular. Layer L_i does not need to know whether its input is the input layer itself or the output of L_{i-1}. L_i computes its activations as $a^{(i)} = f^{(i)}(Wa^{(i-1)} + b^{(i)})$, with $a^{(0)} = x$, where $f^{(i)}$ is the non-linearity used by L_i (sigmoid, by default). Feed-forward is the process of computing the activations of all neurons in the network layer-by-layer, from $i = 1$ to $i = 2$ in our case.

Let us perform the feed-forward for the network in Figure 4:

$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$a^{(1)} = \sigma(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$
$$\hat{y} = a^{(2)} = \text{softmax}(z^{(2)})$$

$\hat{y}$ is the output of the network. Note that we have represented the linear transformation of the L_i by $z^{(i)}$. This will help us in the following sections. The softmax function is defined by:

$$\text{softmax}(z^{(2)})_j \overset{\text{def}}{=} P(\text{label} = j|x) \overset{\text{def}}{=} \frac{\exp(z_j^{(2)})}{\sum_{i=1}^{C} \exp(z_i^{(2)})}$$

This equation is saying that the probability that the input has label j (i.e., in our case, the digit j is handwritten in the input image) is given by $\text{softmax}(z^{(2)})_j$. Therefore, our predicted label for the input x is given by:

$$\text{label} = \text{argmax}(\hat{y})$$

This is basically the digit the network believes is written in the input image.

3.4 Training

Recall that our objective is to learn the parameters of the neural network such that it gets the best accuracy on a set of data points, which we call the *dev* set. Let y be the one-hot vector denoting the class of the input, i.e., $y_c = 1$ if c is the correct label, 0, otherwise. We want $P(\text{label} = c|x)$ to be the highest.

Without going into the mathematical details, we will use the following general expression to determine the error of our neural network. This expression turns out to be the most convenient for our purpose:

$$\text{CE}(y, \hat{y}) = -\sum_{i=0}^{C-1} y_i \log(\hat{y}_i)$$

CE stands for cross-entropy. Since y is a one-hot vector, this simplifies to

$$\text{CE}(y, \hat{y}) = -\log(\hat{y}_c)$$

We can observe that CE is 0 when we have the optimal answer, e.g., $\hat{y}_c = y$. Similarly, CE is maximal ($+\infty$) when $\hat{y}_c$ is 0. This corresponds to a neural network that is "sure" that the digit is *not* c (maximally wrong).

The total cost for N input data points (such that the cross-entropy of the i^{th} training vector is denoted as $CE^{(i)}$):

$$\text{cost} = J(W, b; x, y) = \frac{1}{N}\sum_{i=1}^{N} CE^{(i)}(y, \hat{y})$$

The above cost measures the error or our "dissatisfaction" with the output of the network. The more certain the network is about the correct label (high $P(y = c|x)$), the lower our cost will be.

Clearly, we should choose the parameters that minimize this cost. This is an optimization problem, and is usually solved using the method of Gradient Descent (described below).

Our neural network applies a non-linear function to the input because of the sigmoid and softmax functions. However, if we make W very small, the network becomes nearly linear because Wx is itself very small. To optimize the neural network, we often add a penalization term for the magnitude of W in order to control the non-linearity of the network. There is no rigorous justification for this penalization. It is found to work well in practice and is easy to use. With the penalization term, the cost function becomes

$$\boxed{J(W, b; x, y) = \frac{1}{N}\sum_{i=1}^{N} CE^{(i)}(y, \hat{y}) + 0.5\ \lambda\ \|p\|^2} \tag{1}$$

where $\|p\|^2$ is the sum of the l^2-norm of all the weights W of the network, and λ needs to be tuned for best performance.

3.5 Gradient Descent

Gradient Descent is an iterative algorithm for finding local minima of a function. For our case,

$$\boxed{p \leftarrow p - \alpha\ \nabla_p J} \tag{2}$$

where α is the learning rate that controls how large the descent step is. $\nabla_p J$ is the gradient of J with respect to the network parameters p.

In practice, we often do not compute J using all the input images: $J = \sum_{i=1}^{N} \text{CE}^{(i)}$, where N are all the images. Instead, we subdivide the input into mini-batches containing M images. We process one mini-batch at a time. For each mini-batch, we calculate J, and update the network coefficients. Then, process the next batch, until all images are processed. This algorithm is also called Stochastic Gradient Descent. See Listing 5 for the pseudo-code. In the code, an epoch (in the machine learning lingo) is an iteration over the entire data set of N images.

This approach usually leads to faster convergence because we update the network coefficients more often and are able to learn faster.

```
epoch = 0
while epoch < MAX_EPOCHS:
  batches = split(training_samples, M)
  for batch in batches:
    p = p - step * gradient(batch)
  epoch += 1
```

Figure 5: Stochastic Gradient Descent with mini-batches

3.6 Backpropagation

Backpropagation is the process of updating the neural network coefficients. This involves computing the gradient of multi-variable functions using the chain rule, to obtain $\nabla_p J$.

Let's compute the gradient for the parameters in the last layer (2) of our network:

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z_k^{(2)}} = -\frac{\partial \log \left(\dfrac{\exp(z_c^{(2)})}{\sum_{i=0}^{C} \exp(z_i^{(2)})} \right)}{\partial z_k^{(2)}} = -\frac{\partial \left[z_c^{(2)} - \log \left(\sum_{i=0}^{C} \exp(z_i^{(2)}) \right) \right]}{\partial z_k^{(2)}}$$

There are two cases here:

1. Case I: $k = c = y_i$, i.e., k is the correct label

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z_k^{(2)}} = -1 + \frac{\exp(z_k^{(2)})}{\sum_{i=0}^{C} \exp(z_i^{(2)})} = -1 + \hat{y}_k = \hat{y}_k - y_k$$

2. Case II: $k \neq y_i$

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z_k^{(2)}} = 0 + \frac{\exp(z_k^{(2)})}{\sum_{i=0}^{C} \exp(z_i^{(2)})} = \hat{y}_k - y_k$$

Therefore, the gradient in vector notation simplifies to

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(2)}} = \hat{y} - y \tag{3}$$

Recall that $z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$, such that $z^{(2)} \in \mathbb{R}^{H_2 \times 1}$, $a^{(1)} \in \mathbb{R}^{H_1 \times 1}$ and $W^{(2)} \in \mathbb{R}^{H_2 \times H_1}$. Therefore,

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(2)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(2)}} = (\hat{y} - y){a^{(1)}}^T} \tag{4}$$

Similarly,

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial b^{(2)}} = \hat{y} - y} \tag{5}$$

Going across L_2:

$$\frac{\partial z^{(2)}}{\partial a^{(1)}} = {W^{(2)}}^T$$
$$\frac{\partial \text{CE}(y, \hat{y})}{\partial a^{(1)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} = {W^{(2)}}^T (\hat{y} - y)$$

Going across the non-linearity of L_1:

$$\begin{aligned} \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}} &= \frac{\partial \text{CE}(y, \hat{y})}{\partial a^{(1)}} \frac{\partial \sigma(z^{(1)})}{\partial z^{(1)}} \\ &= \frac{\partial \text{CE}(y, \hat{y})}{\partial a^{(1)}} \circ \sigma(z^{(1)}) \circ (1 - \sigma(z^{(1)})) \end{aligned}$$

Note that we have assumed that $\sigma(\cdot)$ works on matrices by applying an element-wise sigmoid, and $\circ$ is the element-wise (Hadamard) product.

That brings us to our final gradients:

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(1)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(1)}} = \Big(\frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}}\Big) x^T} \tag{6}$$

Similarly,

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial b^{(1)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}}} \tag{7}$$

The above equations have been derived for a single training vector, but they extend seamlessly to a matrix of N column vectors.

4 Your Tasks

You task is to implement a parallel training process for our two layer neural network utilizing both CUDA and MPI. You should focus on parallelizing each iteration that processes a batch of images. The work should be parallelized among a certain number of MPI nodes (e.g., 4 MPI nodes), and the computation on each node should be accelerated by GPU kernels. You'll need to carefully analyze the neural network training algorithm, the overhead of the MPI and CPU-GPU communication and decide the granularity of parallelism. For example, you may choose to fork (distribute data

among MPI nodes) and join (collect, reduce on one MPI node) on every matrix operation, or fork at the beginning, keep intermediate results local at each node and join at the end, or a combination of both.

Possible optimizations may include:

- **GPU accelerated matrix operation**: The training process contains GEMM (General Matrix-matrix Multiplication) and element-wise matrix operations, which should be accelerated by GPU. The key to optimizing the GPU kernels involves a wise choice of blockSize and gridSize, memory coalescing, usage of shared memory, etc.
- **Communication overhead**: Both MPI and GPU computation introduce communication overheads. While some of the communication are required, some can be avoided.

4.1 Outline of parallelization strategies for CUDA and MPI

- **GEMM CUDA implementation:** A GEMM operation can be expressed as $D = \alpha * A * B + \beta * C$:

$$d_{ij} = \alpha \sum_k a_{ik} b_{kj} + \beta c_{ij} \tag{8}$$

  Some BLAS libraries perform in-place computation that saves the result D in the memory space of C, as cuBLAS does. This is possible if C is no longer used after the computation.

  Please read the "CUDA C Programming Guide" written by NVIDIA:

  `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

  `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

  We first outline the basic implementation ("Algorithm 1"). The simplest implementation is to have one thread calculate a single element in the result D. Each thread reads the required row of A, column of B, and an element of C to compute the output element in D. This is equivalent to having each thread calculates the entire sum of k for a given (i, j) in Equation 8. This is equivalent to Figure 9 on p. 25 (CUDA PG, `PG-02829-001_v10.2`).

  You are free to refer to online documentation and code for your project. **However, all the work you are submitting must be entirely your own. You are not allowed to copy code written by someone else.**

  For Algorithm 2, you need to use shared memory and propose a better implementation. For example, use a blocking algorithm and take advantage of the shared memory. One approach is to have each thread block (e.g., a block of 32×32 threads) compute a sub-matrix (of size 32×32) in the output matrix. Blocks from matrices A and B can be loaded into shared memory, with each thread reading one element of each sub-matrix. Each thread then updates its entry in the sub-matrix of D. A loop is used to multiply all the required entries in A and B. See Figure 10 p. 28 (CUDA Programming Guide).

  We intentionally do not explain the details of these algorithms. It's for you to fill the blanks and perhaps come up with better ideas!

- **MPI implementation:** For each batch of images, you should subdivide the input images into smaller image batches and use MPI communication methods to distribute the input data and Neural Network parameters among MPI nodes, and perform GEMM and other computations in parallel. The resulting network coefficient updates should be reduced and sent to all nodes.

4.2 An even better GEMM implementation

Here is a suggestion for a more advanced implementation with higher performance. Each block can have size 16×4 (i.e., `BlockDim.x = 16`, `BlockDim.y = 4`), for a total of 64 threads. This block computes a sub-matrix of size 64×16 in D. Similar to the shared memory implementation, all 64 threads in the thread block work cooperatively to loop over the corresponding rows in A and columns in B. Within each iteration, a 4×16 sub-matrix of B is loaded into the thread block's shared memory, with each thread reading one element. Each thread then reads in a 1×4 sub-matrix of A into a local array `a[4]`, not into shared memory (this is not useful here). In total a 64×4 sub-matrix of A is read, while each one of the 64 threads only uses one row of this sub-matrix. Each thread computes a row of size 1×16 in the output D.

You may also consider the implementation with a hierarchical tiling documented at
`https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/`
It is slightly different from the implementation above but can also can give you excellent performance. The relevant sections on the web page are from the top down to the section called `WMMA GEMM` (not included).

Explain why the approach you picked is advantageous compared to Algorithm 1.

4.3 Improved blocking to reduce MPI communications

This part of the project is worth an extra 15 points as bonus. This part is more challenging and should be attempted only if you are done with the other assignments in the project and feel comfortable with exploring a more advanced topic.

Let's simplify the problem by considering the following simplified situation. We can think of the feedforward phase as a series of matrix-matrix multiplications of the form WA where A are activations from layer $l-1$ and W are the weights of layer l. Since we are processing multiple input images at the same time, the number of columns of A is the size of the batch. The number of columns in W is the number of activations in layer $l-1$ and the number of rows is the number of activations in layer l.

The backpropagation phase corresponds to a series of products of the form $W^T d$ where d is the "gradient" vector from layer l and $W^T d$ is the "gradient" for layer $l-1$. The NN weights are updated using a sum of rank-1 matrices of the form $\Delta W = \sum_i d_i a_i^T$, where the sum is over all the input images. The vector d_i (from image sample i) is the "gradient" vector from the backpropagation at layer l while a_i is the activation for layer $l-1$.

The goal is then to subdivide or block the data and operations in such a way that communication is minimized. In the current approach, we split the input sample A into column blocks and assign one block to each processor. Then we do a reduction of all the $d_i a_i^T$ contributions from all processors. Since the size of the matrix ΔW is quite significant, this approach leads to a large MPI overhead.

To reduce the communication time, a common optimization is to partition the NN coefficient matrices W across different processors. This is called model parallelism as opposed to data parallelism. Data parallelism corresponds to splitting the input "data" across different processors, while model parallelism correspond to splitting the NN coefficients W because the NN is our model in this problem. Model parallelism typically splits the weights along a certain dimension, and synchronization occurs when outputs from other parts of the model are required. Splitting weights across different dimensions would result different operations needed when synchronizing. In a more abstract manner, let's focus on a product WA and write it out as

$$\sum_k W_{ik} A_{kj}$$

So for each triple (i, k, j) there is one multiplication and addition to perform. We can map all the triplets onto a parallelepiped Ω whose dimensions are given by the dimensions of W and A (namely the number of rows in W, the number of columns in W, and the number of columns in A). The question is then, generally speaking, how to partition this set of triplets Ω across the different processors to minimize communication.

You can experiment with different ideas to reduce the amount of data that needs to be communicated using MPI. For example, as we perform many iterations, the input data is fixed across iterations (the input images). Therefore for the first layer you can experiment with splitting $W^{(1)}$ along the rows (index i) and sharing the input images across all processors (which needs to be done only once). Similarly, $W^{(2)}$ is a very fat matrix with only 10 rows (the 10 labels). You can therefore experiment with splitting this matrix along its columns (index k). Experiment with different types of blocking and determine which give the best performance. You can estimate with hand calculations the communication cost for each scheme in order to guide you towards the best combination. Look at the matrix and data set sizes used in the code.

You can search the literature for more details on these ideas. For example, see Google Scholar papers, and Krizhevsky [2014].

References

[1] Yann LeCun et. al. MNIST. `http://yann.lecun.com/exdb/mnist/`. [Online].