Handwritten digit recognition

Stefan Niculae Ionut Ciocoiu

Parallel & Concurrent Programming Lab Report
February 9, 2017

Abstract

We implement a neural network to classify handwritten digits with training done in parallel, using MPI. We present the dataset, model architecture, parallelization strategy and an interactive interface. Then we discuss model accuracy evolution and time performance benchmarks. Our implementation achieves good prediction performance with fast training time.

1 Introduction

The project aims to highlight the role of parallelism in programming, especially in Machine Learning tasks. Considering the vast amount of data today's applications have to deal with, it is paramount to run tasks concomitantly. To underline improvements bought by parallelism, we implemented a neural network training module in MPI.

In the following sections we will talk about the dataset used; detail model architecture and reason for choosing it along with its training performance. Parallelism has a dedicated section with a high-level overview of the solution and time performance benchmarks. The last sections contain an outline of the graphical interface and implementation details for the entire project.

2 Problem statement

Given a picture of a handwritten digit, label it accordingly 0 - 9. This is a classic problem in Computer Vision on which we apply a neural network trained using gradient descent. We seek to reach satisfactory model performance and fast training time, using parallelism.

3 Dataset

We demonstrate our model performance on the MNIST database, widely used dataset for training and testing in the field of machine learning. It contains 70 000 examples of labeled handwritten digits. The digits have been size-normalized and centered in a fixed-size image [1].

Figure 1: Sample MNIST images [2]

The images are black-and-white, with a resolution of 28×28 pixels, represented as an array of 784 values. Each value ranges from 0 to 1 indicating the amount of blackness in the pixel.

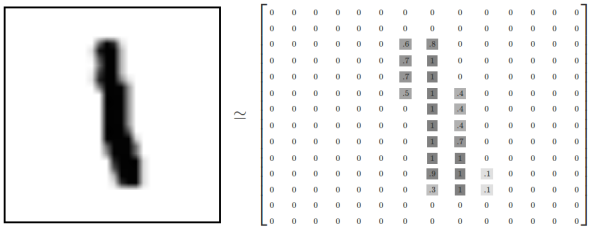


Figure 2: Matrix representation of a digit [2]

4 Model

In order to recognize a given image, we need to model to understand the data. It needs to be able to be queried for a new instance in order to provide an interactive demo.

We use a feed-forward neural network with an one fully connected layer:

$$y = f(Wx + b)$$
$$label = argmax(y)$$

Where W is the weights matrix, 10×784 – the number of classes(one for each digit) and the dimension of an image (784). b is the bias vector, 1×10, corresponding to the free element. y is a vector of probabilities, from which argmax extracts the predicted labe

The activation function output a vector of probabilities:

$$f(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

Where $exp(x)$ represents e^x, Euler's constant. It *normalizes* the vector, causing the greatest elements to have logarithmic relative value.

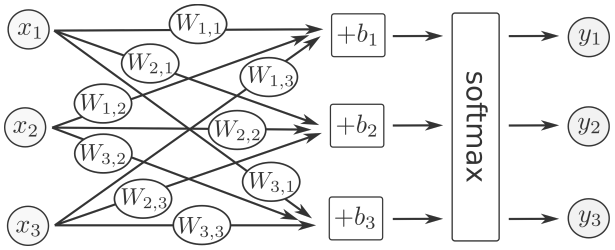


Figure 3: Simplified representation of model graph [2]

The cost function is cross-entropy:

$$H(y_t, y_p) = -\sum_x y_t log(y_p)$$

Where y_t is the true, correct label of an image and y_p is the label predicted by the model.

L_2 regularization is used:

$$\frac{1}{\lambda} \sum_i \sum_j W_{i,j}^2$$

Where λ is the regularization parameter. This limits over-fitting by imposing a penalty on big weights.

The model uses an adaptive learning rate:

$$\eta \frac{W^{(k)} - W^{(k-1)}}{\nabla^{(k)} - \nabla^{(k-1)}}$$

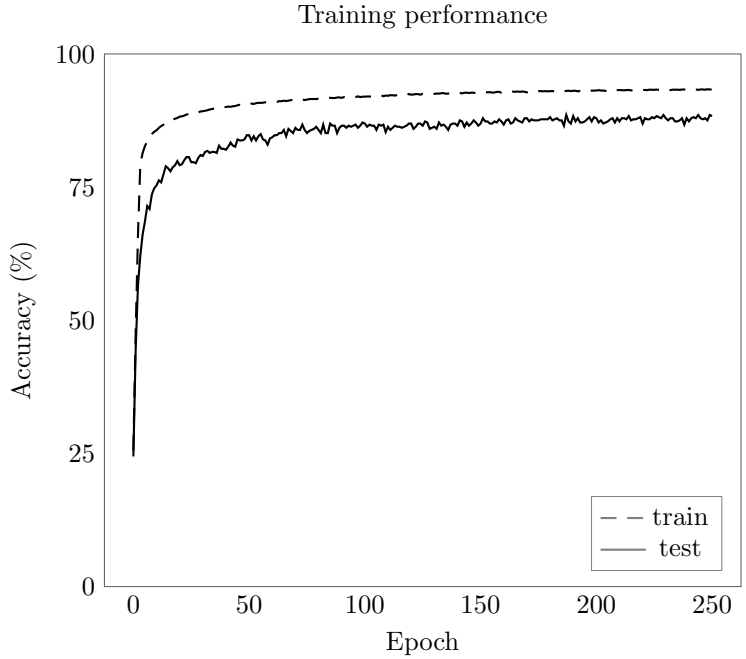
Where η is the momentum parameter. This helps with stability in late epochs.

Training We employ gradient-descent on mini-batches.

$$W^{(k)} = W^{(k-1)} + \alpha \nabla_W^{(k-1)}$$
$$b^{(k)} = b^{(k-1)} + \alpha \nabla_b^{(k-1)}$$

Where W is the weights matrix, 10×784 – the number of classes(one for each digit) and the dimension of an image (28×28). b is the bias vector, 1×10, corresponding to the free element. $W^{(k)}$ denotes the weights at step k. ∇ represents the update which must produce to improve prediction.
Early-stopping prevents the model from further learning when performance on the validation split starts decreasing. Random sampling averts cycles when iterating through the dataset.

Evaluation Model performance is measured on a separate testing set using the accuracy metric – ie: the percentage of images correctly classified.
The following chart illustrates model evolution, after each pass through the dataset, on the entire MNIST training (60 000) and test (10 000) dataset.



The training performance is intentionally dashed, as what interests us is the model's generalization power. It quickly learns in the first 50 epochs and reaches a plateau around the 100 mark. It achieves 93.3% performance on the training set and 88.3% on unseen data.

5 Parallelization

The phase that needs parallelization is the training of the model. Predicting is not computationally intensive. The bulk of the time is spent iterating over the entire training set multiple times. In one epoch, the network looks at every image, in batches and does the following:

- compute the change that must happen in order to better recognize – called gradient,
- updates its internal weights based on the gradient.

Computing the gradient takes the most time. The expensive operations are two dot-products on the weights matrices and the a matrix-wise softmax. This is the portion that we will parallelize.

When looking at a batch of images:

- the batch is assigned evenly among workers,
- each worker computes the gradient on its chunk of the batch,
- partial gradients are sent from each worker to the master,
- the master process accumulates all the partial gradients and aggregates them,
- the master updates the model weights based on the total gradient,
- updated weights are sent back to every worker,
- a new batch is ready to be processed: each worker now knows the effects of all the others.

In our case, aggregating the partial gradients requires nothing more then a summation.

Naturally, there appears a running time – model accuracy trade-off. Updating the weights after every image yields better accuracy than updating once every 1,000 samples. In the case of a greater batch size, the model has not evolved after looking at the first image, nor after the second or the third. It still analyzes them knowing just as much as it did when looking at the first sample.

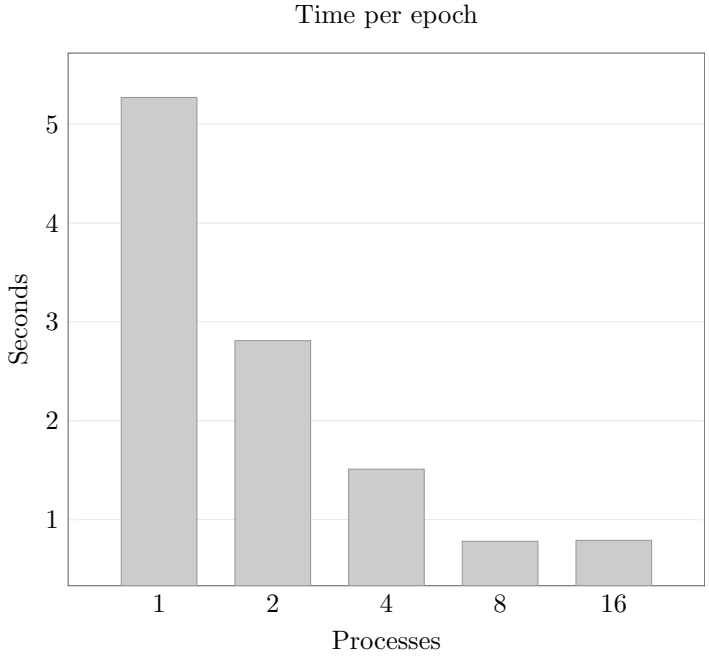
In contrast, a greater batch size produces better parallelization performance. If each worker is assigned a single image, the communication cost of transferring gradient and weight matrices far outweighs the time taken to compute them. Efficiency improvement is present only when the time taken to transfer the results is lower than the time taken to compute them. Greater batch sizes incur more infrequent updates thus reducing the communication overhead.

The trade-off is now clear: a lower batch size leads to greater accuracy while a greater batch size causes faster training.

Another approach would be to let the algorithm run sequentially and parallelize only the matrix multiplication operation. In light of the previously presented considerations, we can conclude that this would be grossly inefficient.

The transfer overhead is much higher than the benefit gained by splitting the problem.

The following chart illustrates the improvement brought by parallelization on one pass through the data:



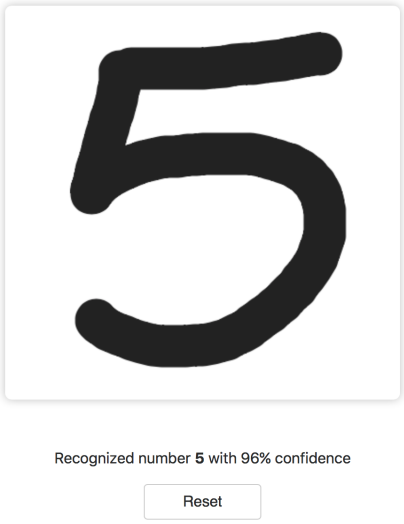
The benchmarks were done on the MNIST training set of 60 000 images with a batch size of 1000, on a machine with 20 processors @ 2.60GHz.

The training time is almost halved when going from one (5.27s) to two (2.81s) processors: 88% improvement. Further 86% improvement is achieved on four processors (1.51s), less than a third (29%) of the single-process time. Doubling the number of processes to eight yields a further 94% improvement (0.78s), making it only 15% of the single-process time. This is notable when considering the perfect improvement possible of 12.5%.

For more than eight cores, the communication overhead starts to take its toll. No further performance improvement is achieved. Chunk sizes get increasingly small (1000, 500, 250, 125, 63), hitting an inflection point at the eight-processes mark.

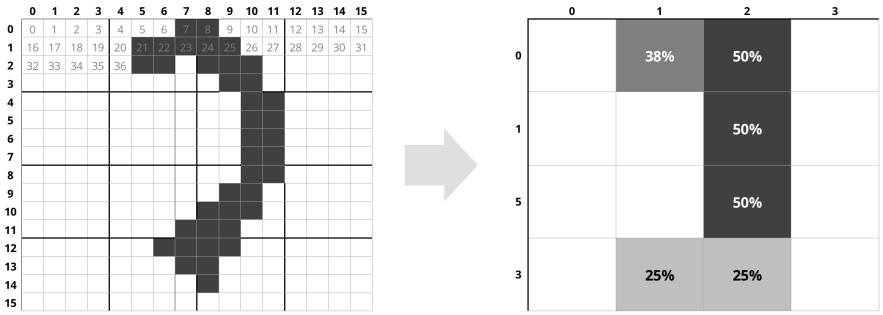
6 Graphical Interface

The user is presented with a blank box in which to draw a digit to be recognized. Upon request, the prediction along with the model's confidence in it is displayed. Since the model was trained on 28×28 images, we need to pass the input in the same format. On today's screens, an input box of 28×28 pixels would look comically small. The raw input must be resized accordingly.



The simplest down-sizing method is taking each block of pixels in the original image and averaging it to form a single pixel in the resulting image. For simplicity, we assume square images and pixels with only one channel – grayscale. Eg: an image with n^2 original pixels is condensed to s^2 pixels, meaning a factor of $f = \frac{n}{s}$. That means each of the s^2 pixels is a sum of a $f \times f$ block divided by f^2.

The trouble appears with encoding the image – both the raw image and the model's inputs are in array form instead of a matrix. Ie: the pixels are all placed one after another starting from the top left corner.



The transformation can be computed by *compacting* the columns: $\sum_i Orig_i$ for column at index $\lfloor \frac{i}{f} \rfloor$. Then for rows: $\sum_i Col_i$ for pixel at index $(i \bmod s) + \lfloor \frac{i}{s \times f} \rfloor$. Where $Orig$ denotes the the original image Col the column-wise averages.

7 Implementation

After going through the high-level overview of the project, we now delve into implementation details.

Predictive model The core of the project is implemented in C++. This includes the neural network training and prediction. Model weights and image samples are kept in memory using 2D-arrays.

Matrices build as vector of vectors (using the standard library implementation), while providing greater flexibility, cannot be sent efficiently using MPI. MPI deals with contiguous blocks of memory, while STL vectors are, by definition, dynamic. A work-around for this is to send a matrix row-by-row. Our initial implementation used this method but unfortunately it provides little to no increase in parallelization performance and caused a general-rewrite.

The most expensive matrix operations are dot products and exponentiation. For matrix multiplication, we explored the Strassen algorithm, $O(n^{2.8})$, and the Coppersmith–Winograd algorithm, $O(n^{2.37})$. Although they achieve high asymptotic performance, hidden constants make them impractical in our case. We went with the Naive, $\Theta(n^3)$, algorithm with the ikj optimization. For exponentiation, we rely on STL's implementation, although this can be further optimized.

Graphical Interface The GUI is implemented as a web page in HTML, with a canvas for the input box. The drawing is facilitated by the Sketch library [3]. Interactivity and picture resizing is done through JavaScript.

Communication Server The glue between the front-end and the model is a web-server done in Python, using the Flask library [4]. An array of 784 pixels is received through from the GUI and is passed to the compiled executable. The output (predicted digit and prediction confidence) is then passed back through AJAX.

The entire source code is available at `https://github.com/stefan1niculae/mnist-in-mpi`.

References

[1] C. J. B. Yann LeCun, Corinna Cortes, "Mnist handwritten digit database," 1998.

[2] Y. Tang, "Tensorflow - mnist for ml beginners," 2015.

[3] M. Bleigh and Intridea, "Sketch javascript library," 2011.

[4] M. Grinberg, "Flask python library," 2014.