A Machine Learning Method For Unmixing 4-D Ptychographic Images And Its Implementation on GPU

Zhen ZHANG
The Chinese University of Hong Kong,
zzhang3913@gmail.com

August 3, 2017

Abstract

This study focuses on unmixing 4-D ptychographic images which incorporate different linear combinations of basic modes. Least square method for this problem gives poor results. In this study, a machine learning method is proposed to achieve better accuracy. The training data, instead of being collected from experiments, are generated synthetically. Performances of different data generation methods are compared and the decreases of the cost function are shown. The neural network is tested with all data we have and the result is satisfactory. The algorithm is implemented on `C`, with LAPACK for CPU code as well as MAGMA for GPU code. Generally the CPU code works well with smaller number of training examples, while GPU code is faster when the matrices are larger. The nature of this algorithm inspires me to work out an algorithm to compute the matrix inverse based on neural network. The convergence of the mentioned algorithm is well studied in this paper.

1 Introduction

Fast electron detectors are gaining ground in traditional high-resolution microscopy studies. In particular, 4D ptychographic datasets collected over a range of real and reciprocal space coordinates are believed to contain a wealth of information about structure and properties of materials. Currently, There are three basic modes known, M_0, M_1, M_2, each of which is a 2688 by 2688 image. The M_0 is the mode with high symmetry and M_1, M_2 are two distortion modes. M_0, M_1, M_2 are shown in Figure 1. In future computations, let $M_1 = M_1 - M_0, M_2 = M_2 - M_0$.

Test structure incorporates different linear combinations of the two distortion modes. The aim is, for each input image I of the same size as the three basic modes, to find a representation of I with the three basic modes. The input image can be mostly represented as a linear combination of the three basic modes, namely,

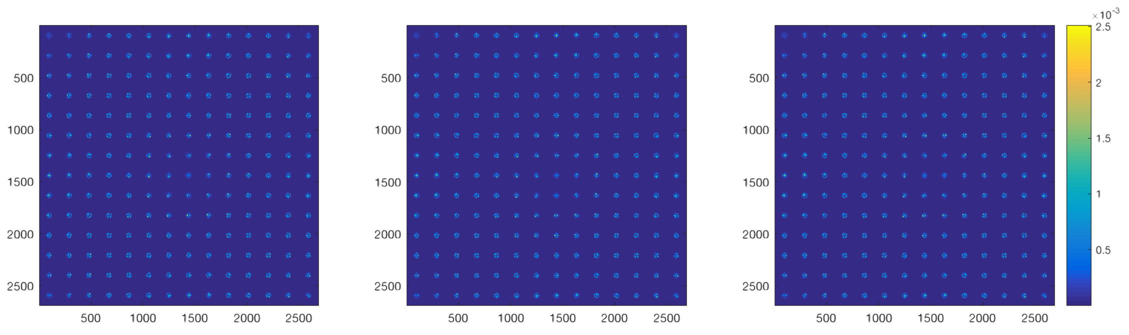

Figure 1: M_0, M_1, M_2

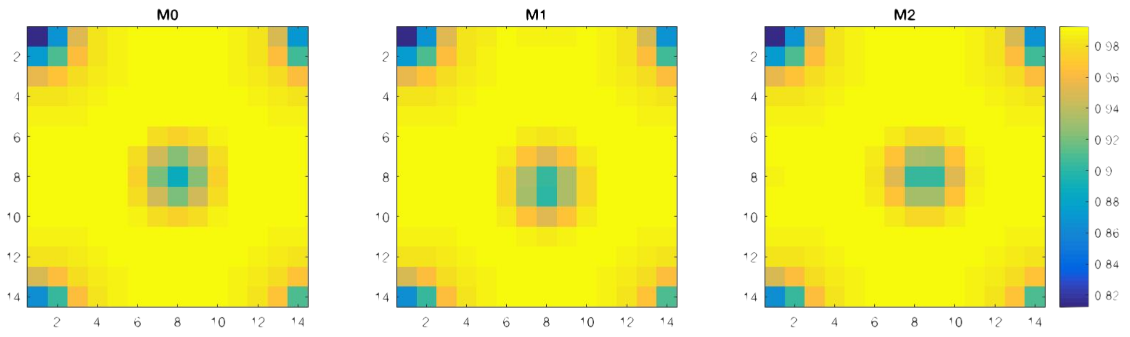

Figure 2: Transformed M_0, M_1, M_2

$$I \approx \alpha M_0 + \beta M_1 + \gamma M_2.$$

Currently, 16 input images are provided. Every four of the images share the same set of coeffcients (α, β, γ). $\alpha = 1$ for all images. β, γ take value of either -1 or 1.

The result of least square is quite far away from what we desire. Instead of trying to solve this problem by completely analysing the structures and features of the images, we use a machine learning method. Since the experimental data size is merely 16, a larger number of synthetic data are generated by interpolating the bias of the linear approximation.

The output is satisfactory when the input is one of the 16 images. To test the algorithm more rigorously, M_0, M_1, M_1 are input and the outputs are still good.

This algorithm is also implemented with `C` language, both on CPU and GPU. Some details of implementation will be mentioned and some acceleration techniques will be discussed.

The nature of the algorithm is to find a linear approximation of the input image with a nonlinear bias. In other words, it is solving a large overdetermined nonlinear system. Therefore this idea can possibly be used to solve other linear algebra problems based on the NN formulation, such as to compute the matrix inverse. An argument will be given to prove the spectral radius of the mentioned algorithm.

2 Description of the Algorithm

2.1 Image Simplification

In practice, it is known that we can add up all pixel values of every 192 by 192 block and transform the image into 14 by 14, while preserving the features of the image. The transformed three basic modes are shown in Figure 2. This simplification puts a 2688×2688 large image into a 14×14 transformed image. This will greatly reduce the amount of memory needed and improve computation efficiency.

2.2 Bias Interpolation

For each of the input images, I, the bias of the linear approximation can be written as:

$$B = I - (\alpha M_0 + \beta M_1 + \gamma M_2).$$

Recall that (α, β, γ) is known for each of the 16 input images.

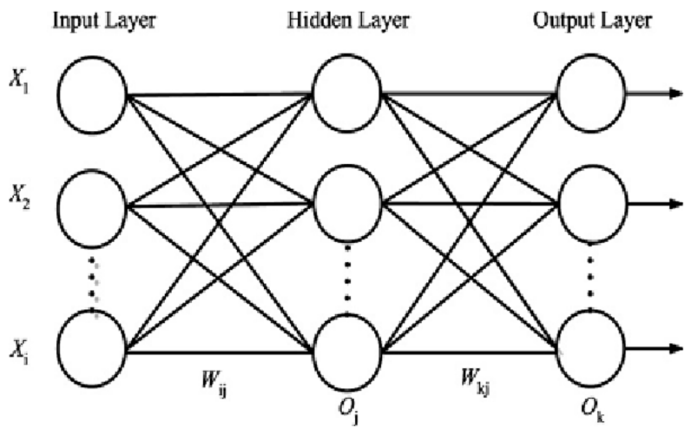

Figure 3: A fully connected neural network

It is assumed that the bias is the result of the interactions between the basic modes. Therefore, for each pixel (x, y) in B, we have

$$B(x, y) = B_{x,y}(\beta, \gamma). \tag{1-1}$$

Since for all input images at hand we have $\alpha = 1$, we temporarily omit α in (1). Then $B_{x,y}(\beta, \gamma)$ is a function whose value at $(1, 1), (1, -1), (-1, 1), (-1, -1)$ are known (though for each set of coefficients there are four bias values, these values are actually very close). Therefore, we can use **interpolation** to find $B_{x,y}(\beta, \gamma)$. Multiple interpolation methods are used. If we take the M_1 and M_2 (original mode image without subtracting M_0) also as input images, we will have two more points, $(1, 0)$ and $(0, 1)$, for interpolation. This paper will mainly focus on 4-point interpolation. Different interpolation methods will be compared in section 3.

Fix $\alpha = 1$. Pick β and γ randomly from the interval $[-1, 1]$. Compute the bias $B_{x,y}(\beta, \gamma)$ for each pixel (x, y). Then a synthetic input image is given by:

$$I_{synthetic} = \alpha M_0 + \beta M_1 + \gamma M_2 + B(\beta, \gamma). \tag{1-2}$$

2.3 Neural Network

Supervised learning is used. Figure 3 shows an example of neural network, from [1]. The network is comprised of layers. The first is the input layer, the last is the output layer and in between are the hidden layers. Size and number of the hidden layers are manually chosen. Between layers are weights, which indicate the strength of connection between the nodes belonging to adjacent layers. An input is given to the neural network and propagated forward. For each hidden layer there is an activation function, which can be `sigmoid()`, `tanh()` or `ReLu()`. After forward propagations, we define a cost function which measures the difference between the network's outputs and the desired outputs. Then the machine modifies its internal adjustable parameters (weights) to reduce this error, using backpropagation.[2] Backpropagation is an efficient way to compute the gradient of the cost function and is therefore crucial for machine learning.

The inputs to the network are the synthetic input images I. The activation functions are `tanh()` except for the output layer, which is linear. The cost function is the sum of the square of the 2-norm of the difference between the output vectors and the training examples, plus a regularisation term. Namely, the cost function is:

$$J(\Theta) = \Sigma_{i=1}^{m} ||o_i - y_i||_2^2 + \lambda ||\Theta||_2^2, \tag{1-3}$$

where Θ is the weights. m is the number of training examples. When the input is the ith example, o_i is the output of the neural network and y_i is the desired output. λ is the regularisation parameter, a hand-chosen constant. The regularisation term prevents overfitting.

2.4 Training

'Training', or 'learning', means to minimise the cost function (1-3). A trained neural network is supposed to output correct results for data other than the training examples. A popular training method is the stochastic gradient descent method (SGD). Some other useful training methods will be mentioned in section 5.

In this project, to minimise the cost function, the Polack-Ribiere flavour of conjugate gradients is used to compute search directions, and a line search using quadratic and cubic polynomial approximations and the Wolfe-Powell stopping criteria is used together with the slope ratio method for guessing initial step sizes. Additionally a bunch of checks are made to make sure that exploration is taking place and that extrapolation will not be unboundedly large.[3]

3 Computations and Results

The 16 input images are arranged as a 4×4 block matrix. The true coefficients corresponding to M_0 (namely, α) are shown as follows:

$$\alpha : \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

Each entry in the matrix is the α corresponding to an input image. Note that for all input images, $\alpha = 1$. Similarly all β and γ are as follows:

$$\beta : \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix},$$

$$\gamma : \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix}.$$

3.1 Least Square Results

Recall: for linear system $Ax = b$, A an $m \times n$ real matrix, when $m > n$, a good approximate solution is given by $x = (A^T A)^{-1} A^T b$. This is least square method. Its result for this problem is as follows:

$$\alpha_{LSQ} : \begin{bmatrix} 0.9950 & 0.9924 & 0.9700 & 0.9712 \\ 0.9927 & 0.9982 & 0.9652 & 0.9645 \\ 0.9678 & 0.9631 & 0.9320 & 0.9467 \\ 0.9716 & 0.9660 & 0.9409 & 0.9426 \end{bmatrix},$$

Method	Cost
`linear`	0.0712
`v4`	0.4695
`cubic`	0.0620
`natural`	0.0590

Table 1: Cost of different interpolation methods in 4-point interpolation

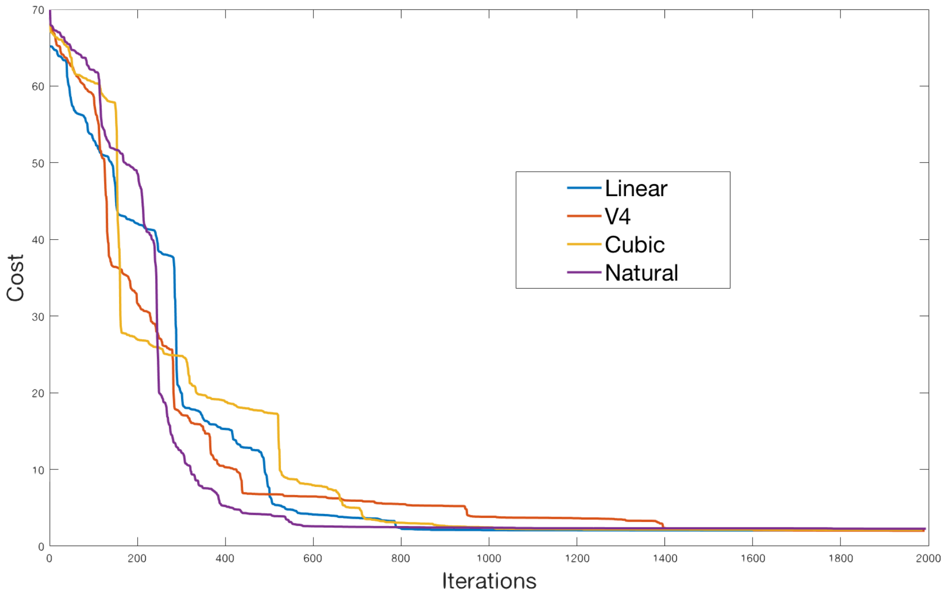

Figure 4: Learning in different interpolation methods. Omit the first 5 iterations for clearance

$$\beta_{LSQ}: \begin{bmatrix} 0.8284 & 0.8349 & 0.9280 & 0.9544 \\ 0.8117 & 0.8117 & 0.9676 & 0.9881 \\ -0.4186 & -0.4692 & -0.3302 & -0.2990 \\ -0.4570 & -0.5001 & -0.3538 & -0.3590 \end{bmatrix},$$

$$\gamma_{LSQ}: \begin{bmatrix} 0.7945 & 0.8472 & -0.4802 & -0.4999 \\ 0.8798 & 0.8754 & -0.4627 & -0.4921 \\ 0.9440 & 0.9474 & -0.3569 & -0.4003 \\ 0.9379 & 0.9591 & -0.3422 & -0.3582 \end{bmatrix}.$$

The least square method does not output accurate results. The total error (cost) is 5.8837.

3.2 Choice of Interpolation Method

Experiments are carried out to find a good interpolation method, with 2 hidden layers, 15 nodes in a hidden layer, 200 synthetic training examples (as are generated by (1-2)), regularization parameter=0.05 and 4-point interpolation. Table 1 shows for each interpolation method the total cost for all 16 input images after 2000 iterations. Note that all interpolations give better result than the least square method.

Figure 4 is the graph for the learning process of the four interpolation methods.

According to the graph and the error table, all methods performs well, except the `v4` (biharmonic spline) interpolation.

3.3 Predictions in 4-point Case

With the parameters same as in section 3.2, the prediction results of 4-point interpolation are as follows:

With **linear** interpolation:

$$\alpha_{4-point-linear}: \begin{bmatrix} 0.9974 & 0.9969 & 1.0013 & 1.0017 \\ 0.9966 & 0.9987 & 1.0009 & 0.9983 \\ 0.9975 & 0.9984 & 1.0001 & 1.0021 \\ 0.9978 & 0.9975 & 0.9994 & 1.0005 \end{bmatrix},$$

$$\beta_{4-point-linear}: \begin{bmatrix} 0.9371 & 1.0653 & 0.9578 & 1.0160 \\ 1.0354 & 0.9152 & 0.9609 & 1.0103 \\ -0.9511 & -0.9957 & -0.9371 & -1.0068 \\ -0.9632 & -1.0313 & -0.9725 & -1.0379 \end{bmatrix},$$

$$\gamma_{4-point-linear}: \begin{bmatrix} 0.9420 & 1.0472 & -1.0165 & -1.0059 \\ 1.0105 & 0.9775 & -0.9926 & -0.9606 \\ 0.9569 & 0.9146 & -1.0376 & -0.9786 \\ 1.0284 & 1.0436 & -0.9849 & -0.9727 \end{bmatrix}.$$

With **cubic** interpolation:

$$\alpha_{4-point-cubic}: \begin{bmatrix} 0.9863 & 0.9880 & 0.9872 & 0.9877 \\ 0.9864 & 0.9867 & 0.9872 & 0.9887 \\ 0.9886 & 0.9892 & 0.9826 & 0.9866 \\ 0.9900 & 0.9901 & 0.9842 & 0.9851 \end{bmatrix},$$

$$\beta_{4-point-cubic}: \begin{bmatrix} 0.9228 & 1.0626 & 0.9539 & 1.0231 \\ 1.0188 & 0.8923 & 0.9583 & 0.9958 \\ -0.9593 & -1.0013 & -0.9705 & -0.9772 \\ -0.9475 & -1.0325 & -0.9663 & -1.0108 \end{bmatrix},$$

$$\gamma_{4-point-cubic}: \begin{bmatrix} 0.9163 & 1.0083 & -1.0100 & -1.0174 \\ 1.0067 & 0.9658 & -0.9789 & -0.9377 \\ 0.9545 & 0.9097 & -1.0054 & -0.9915 \\ 1.0069 & 1.0326 & -0.9709 & -0.9524 \end{bmatrix}.$$

With **natural neighbour** interpolation:

$$\alpha_{4-point-natural}: \begin{bmatrix} 1.0007 & 0.9995 & 0.9980 & 0.9973 \\ 1.0009 & 1.0016 & 1.0000 & 0.9987 \\ 0.9999 & 1.0012 & 1.0026 & 0.9996 \\ 1.0002 & 0.9989 & 1.0007 & 0.9997 \end{bmatrix},$$

$$\beta_{4-point-natural}:\begin{bmatrix}0.9235 & 1.0840 & 0.9809 & 1.0461\\ 1.0215 & 0.8881 & 0.9576 & 1.0013\\ -0.9542 & -1.0203 & -0.9665 & -1.0122\\ -0.9681 & -1.0178 & -0.9774 & -1.0330\end{bmatrix},$$

$$\gamma_{4-point-natural}:\begin{bmatrix}0.9266 & 1.0565 & -0.9880 & -0.9788\\ 1.0019 & 0.9853 & -0.9769 & -0.9969\\ 0.9775 & 0.9345 & -0.9928 & -0.9836\\ 1.0283 & 1.0734 & -0.9756 & -0.9830\end{bmatrix}.$$

The errors of these predictions are already shown in Table 1.

A more rigorous test is to input to the neural network M_0, M_1 and M_2(with no subtraction), whose coefficients were not taken into account during 4-point interpolation. Note that the coefficients for M_0, M_1 and M_2 are $(1,0,0), (1,1,0)$ and $(1,0,1)$. Following is the prediction for these three basic modes by different interpolation methods.

With **linear** interpolation:

$$M_0:\begin{bmatrix}0.9995\\ 0.0424\\ -0.0590\end{bmatrix}, M_1:\begin{bmatrix}1.0021\\ 1.1248\\ -0.0478\end{bmatrix}, M_2:\begin{bmatrix}1.0007\\ 0.0359\\ 1.0546\end{bmatrix}.$$

With **cubic** interpolation:

$$M_0:\begin{bmatrix}0.9987\\ 0.0365\\ -0.0759\end{bmatrix}, M_1:\begin{bmatrix}0.9985\\ 1.1300\\ -0.0463\end{bmatrix}, M_2:\begin{bmatrix}0.9992\\ 0.0134\\ 1.0478\end{bmatrix}.$$

With **natural** interpolation:

$$M_0:\begin{bmatrix}1.0095\\ 0.0561\\ 0.0055\end{bmatrix}, M_1:\begin{bmatrix}1.0067\\ 1.1245\\ -0.0216\end{bmatrix}, M_2:\begin{bmatrix}0.9996\\ 0.0441\\ 1.0848\end{bmatrix}.$$

The prediction is accurate, up to tolerable error.

3.4 Predictions in 6-point case

As mentioned in section 2.2, the interpolation can also be done with 6 points. With all parameters the same as before, the predictions in 6-point interpolation are as follows:

With **linear** interpolation:

$$\alpha_{6-point-linear}:\begin{bmatrix}0.9946 & 0.9935 & 1.0026 & 1.0022\\ 0.9931 & 0.9957 & 1.0009 & 1.0007\\ 1.0002 & 0.9971 & 0.9972 & 0.9973\\ 0.9980 & 0.9980 & 0.9972 & 0.9982\end{bmatrix},$$

$$\beta_{6-point-linear} : \begin{bmatrix} 0.8075 & 0.9706 & 0.9275 & 0.9862 \\ 0.9102 & 0.8161 & 0.8821 & 0.9099 \\ -0.8930 & -0.9665 & -0.9575 & -0.9912 \\ -0.9347 & -0.9748 & -1.0008 & -1.0374 \end{bmatrix},$$

$$\gamma_{6-point-linear} : \begin{bmatrix} 0.8503 & 0.9651 & -0.9411 & -0.9491 \\ 0.9281 & 0.8809 & -0.9320 & -0.8894 \\ 0.8858 & 0.8440 & -1.0163 & -0.9977 \\ 0.9431 & 0.9792 & -0.9656 & -0.9733 \end{bmatrix}.$$

With **cubic** interpolation:

$$\alpha_{6-point-cubic} : \begin{bmatrix} 1.0050 & 1.0059 & 1.0012 & 1.0016 \\ 1.0047 & 1.0055 & 1.0001 & 1.0007 \\ 0.9964 & 0.9964 & 0.9841 & 0.9864 \\ 0.9952 & 0.9965 & 0.9846 & 0.9852 \end{bmatrix},$$

$$\beta_{6-point-cubic} : \begin{bmatrix} 0.8202 & 0.9913 & 0.9054 & 0.9709 \\ 0.9383 & 0.8269 & 0.8814 & 0.9541 \\ -0.9104 & -0.9759 & -0.9896 & -0.9997 \\ -0.9449 & -0.9895 & -0.9959 & -1.0120 \end{bmatrix},$$

$$\gamma_{6-point-cubic} : \begin{bmatrix} 0.8217 & 0.9477 & -0.9498 & -0.9429 \\ 0.9027 & 0.8829 & -0.9440 & -0.9305 \\ 0.8897 & 0.8466 & -1.0121 & -0.9681 \\ 0.9330 & 0.9783 & -0.9813 & -0.9693 \end{bmatrix}.$$

With **natural neighbour** interpolation:

$$\alpha_{6-point-natural} : \begin{bmatrix} 1.0023 & 0.9941 & 0.9917 & 0.9900 \\ 1.0010 & 0.9985 & 0.9954 & 1.0000 \\ 0.9985 & 0.9969 & 0.9968 & 0.9924 \\ 0.9995 & 0.9969 & 0.9954 & 0.9953 \end{bmatrix},$$

$$\beta_{6-point-natural} : \begin{bmatrix} 0.7670 & 0.9237 & 0.8879 & 0.9701 \\ 0.8511 & 0.7487 & 0.8611 & 0.9047 \\ -0.9608 & -1.0219 & -0.9916 & -0.9466 \\ -0.9533 & -1.0203 & -0.9864 & -0.9997 \end{bmatrix},$$

$$\gamma_{6-point-natural} : \begin{bmatrix} 0.8195 & 0.9419 & -0.9591 & -0.9587 \\ 0.8916 & 0.8750 & -0.9569 & -0.9521 \\ 0.8831 & 0.8396 & -1.0153 & -0.9705 \\ 0.9464 & 0.9839 & -0.9741 & -0.9661 \end{bmatrix}.$$

The error of these predictions shown in Table 2.
Predictions for M_0, M_1, M_2 are as follows:

With **linear** interpolation:

$$M_0 : \begin{bmatrix} 1.0035 \\ 0.0021 \\ -0.0188 \end{bmatrix}, M_1 : \begin{bmatrix} 0.9997 \\ 1.0625 \\ -0.0258 \end{bmatrix}, M_2 : \begin{bmatrix} 0.9997 \\ -0.0134 \\ 1.0527 \end{bmatrix}.$$

Method	Cost
`linear`	0.2373
`cubic`	0.2198
`natural`	0.3116

Table 2: Cost of different interpolation methods in 6-point interpolation

With **cubic** interpolation:

$$M_0 : \begin{bmatrix} 1.0023 \\ -0.0097 \\ -0.0367 \end{bmatrix}, M_1 : \begin{bmatrix} 1.0067 \\ 1.0433 \\ -0.0386 \end{bmatrix}, M_2 : \begin{bmatrix} 1.0004 \\ -0.0183 \\ 1.0376 \end{bmatrix}.$$

With **natural** interpolation:

$$M_0 : \begin{bmatrix} 1.0019 \\ 0.0896 \\ 0.0028 \end{bmatrix}, M_1 : \begin{bmatrix} 0.9980 \\ 1.0754 \\ -0.0516 \end{bmatrix}, M_2 : \begin{bmatrix} 0.9983 \\ -0.0092 \\ 1.0749 \end{bmatrix}.$$

The prediction is still accurate.

4 Acceleration

The algorithm is also implemented with `C` language, with both CPU code with LAPACK, and GPU code with MAGMA. Generally for small number of training examples the CPU code is faster while the GPU version works better for larger datasets. Large matrices fully utilize the parallel-oriented design of GPU and thereby the performance is much better than on CPU.

4.1 Communication Control

It is noted that communication between CPU and GPU is extremely time-consuming. In the implementation this communication should be reduced. To do this, all large data structures, including the training examples, outputs of layers, weights and the gradient should be initialized on CPU and be communicated to GPU before learning. This communication control greatly improves the efficiency of the code.

4.2 New CUDA Routines

New CUDA files are written to implement the Hadamard product and element-wise functions. These new routines apply the corresponding operations directly on GPU data and therefore helps to reduce CPU-GPU communication.

Following are prototypes of the routines:

```
void magmablas_dlatanh (int m, int n, double* dA, int ldda, double*
dB, int lddb);
void magmablas_dla_pmult_dtanh( int m, int n, double* dA, int ldda,
double* dB, int lddb);
```

`magmablas_dlatanh()` applies `tanh()` to a device matrix `dA` and the output matrix is stored in another device matrix `dB`.

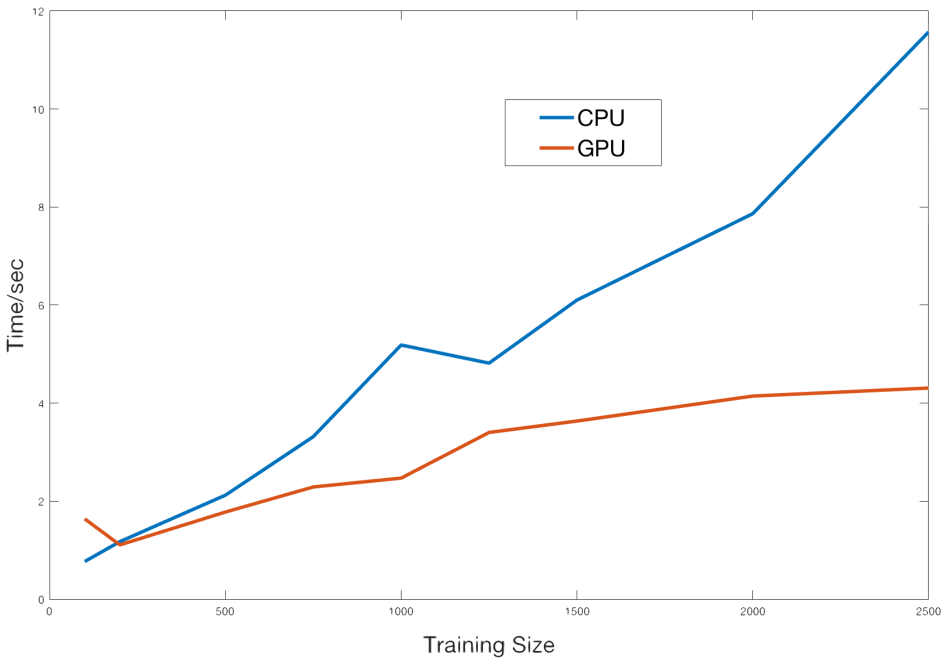

Figure 5: Learning time cost of CPU and GPU code

`magmablas_dla_pmult_dtanh()` applies the derivative of `tanh()` to a device matrix `dA` and multiply the result elementwisely with another device matrix `dB` and the output is stored in `dB`.

4.3 Performances

The experiment is carried out with neural network the same as in section 3.2. The interpolation is 4-point linear and the learning is iterated 2500 times. Training examples are generated by MATLAB.

Computation is carried out on `Bridges` with `interact -gpu` command, which starts an interactive job on a P100 node in the GPU-shared partition with 1 GPU and for 60 minutes. By the user guide of `Bridges`, each P100 node contains:

- 2 NVIDIA P100 GPUs
- 2 Intel Xeon E5-2683 v4 CPUs, each with
  - 16 cores, 2.1 GHz base frequency and 3.0 GHz max turbo frequency
  - 40MB cache
- 128GB RAM, DDR4 2400.

Table Figure 5 and 3 show the performances of the GPU and the CPU code at different training data sizes.

5 Future Work

5.1 Neural Network

Note that in section 3.4, in the 6-point case the predictions on the 16 input images are not as good compared to the 4-point case. Therefore more work can still be done for the details of the neural network, such as optimal regularisation

Training Size	CPU learning time(sec)	GPU learning time(sec)
100	0.768925	1.639293
200	1.175271	1.108230
500	2.122196	1.778217
750	3.315735	2.289727
1000	5.186406	2.470949
1250	4.816086	3.402146
1500	6.102730	3.634838
2000	7.864992	4.144583
2500	11.571694	4.308255

Table 3: Performances of CPU and GPU implementation under different training sizes. Note that the GPU learning time at the training size of 200 is shorter than at 100, and similarly for CPU at 1250 and 1000.

parameter, hidden layer size, deeper network, etc. Other interpolation methods can also be tested and compared.

5.2 Learning Method

Current CG method does not give fast convergence. Other learning methods may also be used. Following are some methods from [4] :

5.2.1 Momentum

Momentum is an improved version of stochastic gradient descent (SGD). The idea behind is, for each iteration, a part of the step change in the last iteration is inherited. The update rule is:

$$\Delta x_t = \rho \Delta x_{t-1} - \eta g_t,$$

where η is the learning rate, ρ is the inertia rate, g_t is the batched gradient.

5.2.2 ADAGRAD

For ADAGRAD, the update rule is as follows:

$$\Delta x_t = -\frac{\eta}{\sqrt{\Sigma_{\tau=1}^{t} g_\tau^2}} g_t,$$

where the denominator computes the $l2$ norm of all previous gradients on a per-dimension basis and η is a global learning rate. Note that the sum in the denominator may diverge to infinity and slow down the learning.

5.2.3 ADADELTA

ADADELTA resolves the learning rate shrinking problem of ADAGRAD. Define iterative relation:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2,$$

where ρ is a hand-chosen decay rate. This definition is similar to the exponential moving average (EMA) in technical analysis.

Define RMS:

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon},$$

where ϵ is a small number preventing RMS from vanishing. Then the update rule is:

$$\Delta x_t = -\frac{RMS[\Delta x]_{t-1}}{RMS[g]_t} g_t.$$

Note that in ADADELTA there is not a hand-chosen learning rate.

5.2.4 Second Order Methods

Second order information is also useful.

The following method uses the diagonal part of the Hessian matrix:

$$\Delta x_t = -\frac{1}{|diag(H_t)| + \mu} g_t,$$

where H_t is the Hessian of the cost function, μ is a small constant.

A method combining Hessian with ADAGRAD is the following:

$$\Delta x_t = -\frac{1}{|diag(H_t)|} \frac{E[g_{t-w:t}]^2}{E[g_{t-w:t}^2]} g_t,$$

where $E[y_{t-w:t}]$ denotes the mean of previous w vectors in a vector sequence $\{y_i\}$.

5.3 Linear Algebra with Neural Network

In this model, with interpolation, essentially the neural network is solving a highly overdetermined non-linear system. Therefore this idea can possibly be used to solve linear algebra problems based on the neural network formulation. The following is an algorithm to compute the matrix inverse with neural network.

Let A be an $n \times n$ invertible real matrix. Let A^{-1} be its inverse. The way to compute A^{-1} with neural network is as follows:

In the neural network there is no hidden layer. The only set of weights connects the input layer and the output layer. The size of both layers is n, where n is the dimension of A. The activation function is linear. Let Θ be the weight matrix of the neural network. Δt is the timestep of gradient descent. $\{b_i\}_{i=1}^n$ is a set of randomly-chosen vectors. The cost function of the neural network is :

$$J(\Theta) = \Sigma_{i=1}^n ||A\Theta b_i - b_i||_2^2. \tag{2}$$

It can be shown by direct differentiation that the gradient of J is the following:

$$\frac{\partial J}{\partial \Theta} = \Sigma_{i=1}^n A^T (A\Theta - I) b_i b_i^T. \tag{3}$$

Let $B = \Sigma_{i=1}^n b_i b_i^T$. Provided that we pick n b_i randomly, B is a symmetric positive definite (SPD) matrix with the probability of 1. Then

$$\Theta_{k+1} = \Theta_k - \Delta t \frac{\partial J}{\partial \Theta}(\Theta_k) \tag{4}$$

$$\Rightarrow \Theta_{k+1} = \Theta_k - \Delta t \Sigma_{i=1}^n A^T (A\Theta_k - I) b_i b_i^T, \tag{5}$$

Let $D_k = \Theta_k - A^{-1}$, then

$$D_{k+1} = D_k - \Delta t A^T A D_k B. \tag{6}$$

Let $A = U\Sigma V^T$ be the SVD of A, where U, V are orthogonal. Then

$$A^T A = V\Sigma^2 V^T. \tag{7}$$

Because B is SPD, we have

$$B = QSQ^T, \tag{8}$$

where Q is orthogonal and S is diagonal.
Then by (6),

$$D_{k+1} = D_k - \Delta t V\Sigma^2 V^T D_k QSQ^T. \tag{9}$$

Then

$$D_{k+1}Q = D_k Q - \Delta t V\Sigma^2 V^T D_k QS. \tag{10}$$

Let $E_k = D_k Q$, e_k^j be the jth column of E_k, $\lambda^{(j)}$ be the eigenvalue of B in the jth column of S. Then

$$e_{k+1}^j = (I - \Delta t\lambda^{(j)} V\Sigma^2 V^T)e_k^j. \tag{11}$$

Let $K_j = (I - \Delta t\lambda^{(j)} V\Sigma^2 V^T)$, then the spectral radius of K_j is

$$max_m |1 - \Delta t\lambda^{(j)}\sigma_m^2|, \tag{12}$$

where σ_m are singular values of A.

My mentor, Dr. S. Tomov, suggests the following simplification:

Choose $b_i = e_i$, $\Delta t = 2/(\lambda_{min} + \lambda_{max})$, where $\{e_i\}_{i=1}^n$ is the standard basis and λ_{min}, λ_{max} are the minimal and maximal eigenvalues of $A^T A$. Then the best convergence rate is

$$max_m |1 - \Delta t\sigma_m^2| = |\frac{\kappa^2 - 1}{\kappa^2 + 1}|, \tag{13}$$

where κ is the condition number of A.

The rate of convergence depends on the square of the condition number of A. However, in some cases the convergence rate can be better. Following are some examples.

By [5] , let A be an $n \times n$ real symmetric diagonal dominant matrix with positive diagonal part D, and let $S_1^2 = D^{-1}$ and $H = S_1 A S_1$. Then the spectral radius r of the Jacobi matrix associated to A is proved:

$$(\kappa(H) - 1)/(\kappa(H) + 1) \le r \le (\kappa(H) - 1)/(1 + \kappa(H)/(n - 1)). \tag{14}$$

And the relation between $\kappa(H)$ and $\kappa(A)$ is given by:

$$\kappa(H)/\kappa(D) \le \kappa(A) \le \kappa(H)\kappa(D). \tag{15}$$

The convergence rate depends largely on the condition number itself.

By [6], for a Hermitian positive definite matrix, the OR Krylov subspace iterates x_k are uniquely defined for each k and can be computed using CG method. Then we have

$$\frac{||x - x_k||_A}{||x - x_0||_A} \le 2(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1})^k, \tag{16}$$

where $||u||_A = (u^H A u)^{\frac{1}{2}}$, κ is the condition number of A, and x is the solution to the linear system. In other words, the convergence depends on the square root of the condition number.

Therefore, there is still a lot to do before the neural network linear system solver becomes outstanding. Future work may focus on investigating other structures of neural network and consolidating the theoretical ground for this method.

5.4 Image Preprocessing

It can be noted that the input images and three basics modes look very similar and the image data have high redundancy. Some filters or principal component analysis (PCA), linear or non-linear, may help.

6 Summary

At the beginning, machine learning for solving this problem seems to be prohibited by the shortage of data. However by interpolations, synthetic data are generated for training. After learning, not only are the outputs for the 16 known input images accurate, which is more or less expected, but also in the 4-point case the predictions for the three basic modes, and in the 6-point case the prediction for M_0, are satisfactory as well. This gives us confidence that in the future if more data can be acquired, this model will still perform well in predicting coefficients for new data.

In the `C` implementation, GPU once again shows its outstanding capability of processing big data.

Machine learning has seen its success in facial recognition, classification, speech recognition and many other modern technologies. But what we have explored of machine learning might only be the tip of the iceberg - many other applications are bourgeoning with hope and energy. This paper sheds some light on linear algebra with neural network and I believe in the future there will be more.

References

[1] Chien-Sheng Chen; Jium-Ming Lin. Applying Rprop Neural Network for the Prediction of the Mobile Station Location. *Sensors*, **2011**, *11*, 4207-4230; doi:10.3390/s110404207.

[2] Yann Lecun; Toshua Bengio; Geoffrey Hinton. Deep Learning. *Nature*, **2015**, Vol. 521, pp 436-444; doi:10.1038/nature14539.

[3] Carl Edward Rasmussen. fmincg().m, **2001** and **2002**.

[4] Matthew D.Zeiler. ADADELTA: An Adaptive Learning Rate Method. arXiv:1212.5701v1 [cs.LG], **2012**.

[5] M. Arioli; F. Romani. Relations between condition numbers and the condition convergence of the Jacobi method for real positive definite matrices. *Numerische Mathematik* 46, 31-42 **(1985)**.

[6] Jörg Liesen; Petr Tichý. Convergence analysis of Krylov subspace methods. **MSC(2000)** 15A06, 65F10, 41A10.

Acknowledgements

This project is done under the RECSEM program at JICS, UTK. The program is sponsored by NSF, JICS and the Chinese University of Hong Kong.

Let me extend my gratitude to my mentors: Dr. S. Tomov, Dr. A. Haidar and Dr. K. Wong from UTK and Dr. R. Archibald from ORNL.