THE UNIVERSITY of TENNESSEE KNOXVILLE

香港中文大學
The Chinese University of Hong Kong

UNMIXING 4-D PTYCHOGRAPHIC IMAGES

Mentors: Dr. Rick Archibald(ORNL), Dr. Azzam Haidar(UTK), Dr. Stanimire Tomov(UTK), and Dr. Kwai Wong(UTK)

PROJECT BY:

MICHAELA SHOFFNER(UTK)

ZHEN ZHANG(CUHK)

HUANLIN ZHOU(CUHK)

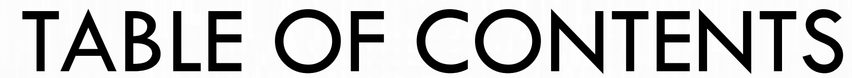
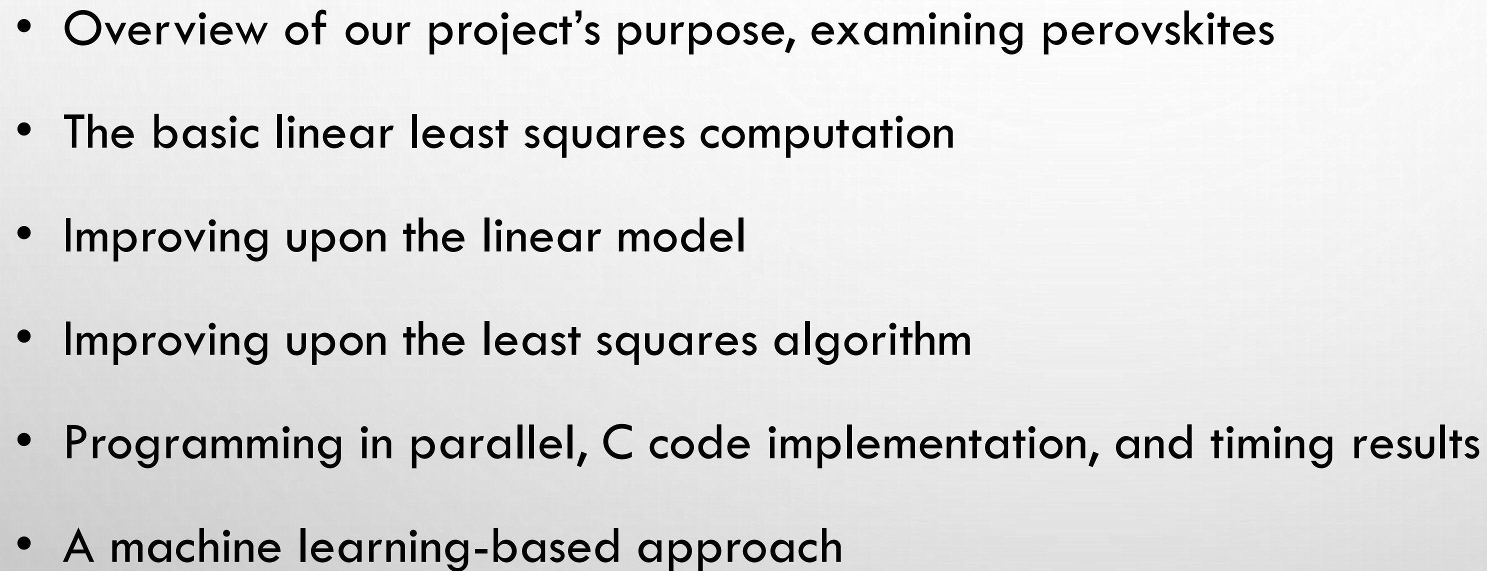


TABLE OF CONTENTS


• Overview of our project's purpose, examining perovskites
• The basic linear least squares computation
• Improving upon the linear model
• Improving upon the least squares algorithm
• Programming in parallel, C code implementation, and timing results
• A machine learning-based approach

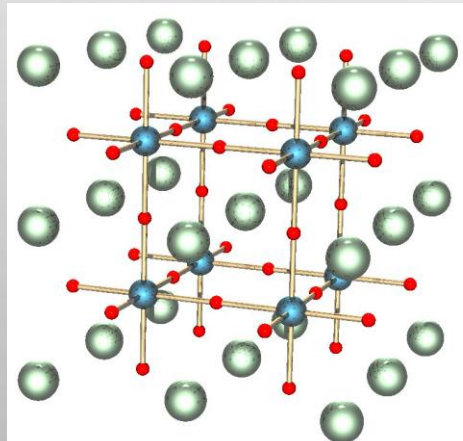
OUR INTEREST

Fast electron detectors are gaining ground in traditional high-resolution microscopy studies, particularly 4D ptychographic datasets which are believed to contain a wealth of information about structure and properties of materials. However, currently available data analysis methods are either too general, only allowing for analysis of simplest objects, or too reductive, effectively recreating traditional detectors from these datasets before interpretation. This project aims to explore the ways that symmetry mode analysis can be adapted to analyze 4D datasets of materials such as multifunctional complex oxides, or perovskites.

INTRODUCTION TO PROJECT - PEROVSKITES

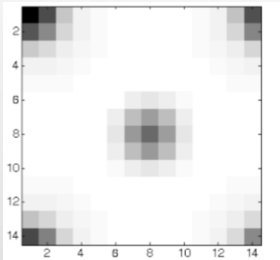
What is a perovskite structure?

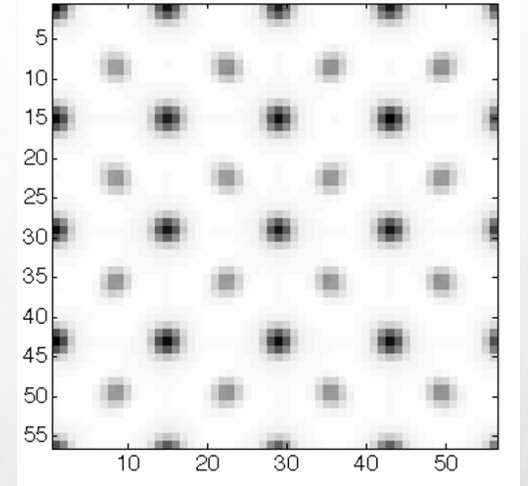
- Any one of many crystalline materials with a structure of ABX_3 , such as calcium titanium oxide, or $CaTiO_3$
- These structures are of interest to material science, and have potentially useful conductivity and dielectric properties for use in solar cell development
- They can have very different properties depending on the sample's exact molecular structure and distortion, with determining the type of distortion present in a particular sample is the goal of this project.



OVERVIEW OF PROBLEM SET UP

- What we have:
  - 3 baseline modes, M_1, M_2, and M_0
  - 16 images, x, each consists of a combination of these 3 modes
  - 3 4x4 true weight matrices





What we want: the true model of the composition of the image, such that the calculated weights of the 3 modes equal the true weights.

Beginning model: linear least squares optimization:

$x = \alpha M_1 + \beta M_2 + \gamma M_0$, with α, β, and γ being the weights of the three modes and x being the resulting image.

LEAST SQUARES SOLUTION

First possible solution executed in Matlab.

$x = \alpha M_1 + \beta M_2 + \gamma M_0 \leftrightarrow ||x-(\alpha M_1+\beta M_2+\gamma M_0)||_2=0 \quad \leftrightarrow \quad Aw = x$

$A=[M_1\ M_2\ M_0]$: 7,225,344-by-3 : a matrix containing the data of the 3 modes.

x: 7,225,344-by-1 : a vector of the data of the resulting image.

Formulation: let $w=[\alpha; \beta; \gamma]$ (3-by1), find w that minimizes $||Aw-x||_2$, to solve $Aw = x$.

4-by-4 unit cells→4-by-4 weights for each mode.

Weight Matrix (Calculated):

0.8284	0.8117	-0.4186	-0.4570
0.8349	0.8117	-0.4692	-0.5001
0.9280	0.9676	-0.3302	-0.3538
0.9544	0.9881	-0.2990	-0.3590

Weight Matrix (True):

1.0000	1.0000	-1.0000	-1.0000
1.0000	1.0000	-1.0000	-1.0000
1.0000	1.0000	-1.0000	-1.0000
1.0000	1.0000	-1.0000	-1.0000

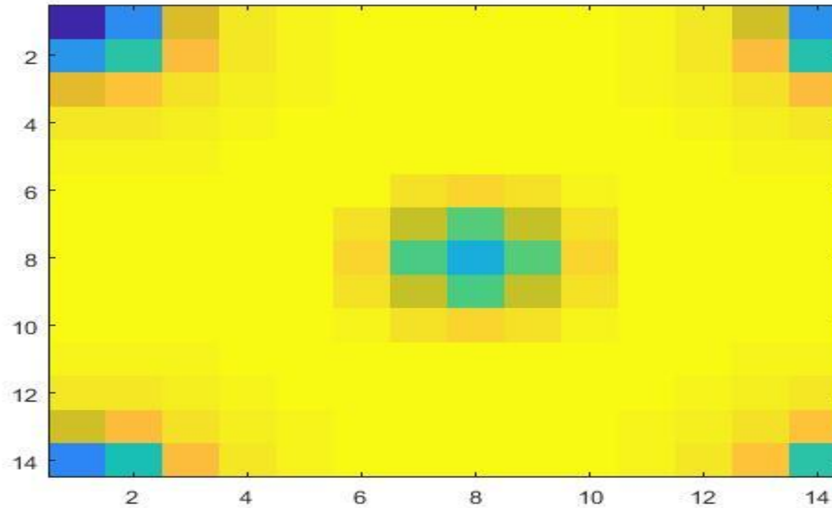
GOALS

Our general goals for this summer consist of:

- Increasing the accuracy of the calculated weights, by using alternative algorithms
- Increase the speed the calculations can be performed, by changing the programming language, using a GPU for calculations, and/or implementing it in parallel
- Explore the possibility in using machine learning to find the correct weights, in place of the standard mathematical approaches

Model Improvement: Gradient

(M_0)



$G_0x(:,1)=M_0(:,2)-M_0(:,1);$

$G_0x(:, j)=M_0(:, j+1)-M_0(:, j);$

…

$G_0x(:,14)=M_0(:,1)-M_0(:,14);$

$G_0y(1, :)=M_0(2,:)-M_0(1,:)$

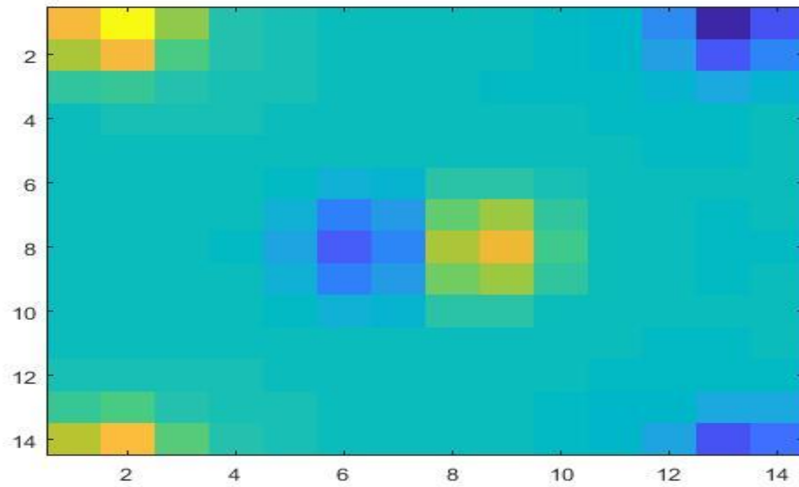
$G_0y(i , :)=M_0(i+1, :)-M_0(i , :);$

…

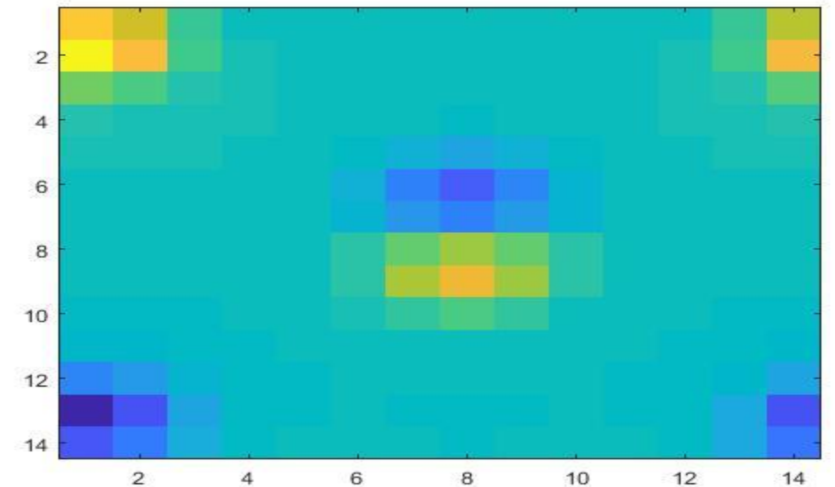
$G_0y(14,:)=M_0(1, :)-M_0(14, :);$

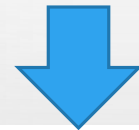
Model Improvement: Gradient

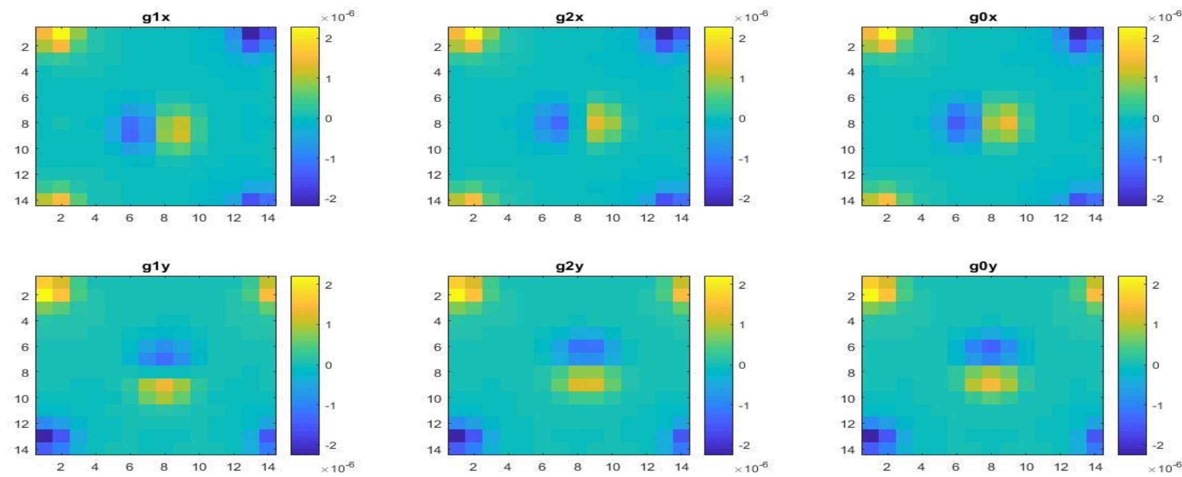
G_0x:



G_0y:



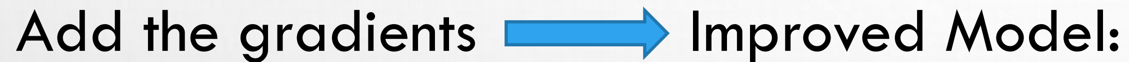




Model Improvement: Gradient

Original Model - Simple linear:

$$x=\alpha M_1+\beta M_2+\gamma M_0 \quad \text{Or} \quad Aw = x$$



Add the gradients → Improved Model:

$$x=\alpha M_1+\beta M_2+\gamma M_0+a*G_1x+b*G_2x+c*G_0x+d*G_1y+e*G_2y+f*G_0y$$

Algorithm: Least Square

- Find α, β, γ, a, b, …, f such that

$$|| \alpha M_1+\beta M_2+\gamma M_0+a*G_1x+b*G_2x+c*G_0x+d*G_1y+e*G_2y+f*G_0y - x||_2$$

is minimized.

→ α, β, γ, the weights of the three modes

Model Improvement: Gradient

Result:

Total Difference=$\sum$|Calculated Weight – True Weight|

From original model: 11.6405

From improved model: 2.9635

M_1(improved):

0.7063	0.8202	-0.9917	-1.1031
0.9238	0.8443	-1.0185	-1.0838
1.0107	0.9142	-0.8604	-0.9573
1.0271	0.9366	-0.9759	-1.0386

M_1(original):

0.8284	0.8117	-0.4186	-0.4570
0.8349	0.8117	-0.4692	-0.5001
0.9280	0.9676	-0.3302	-0.3538
0.9544	0.9881	-0.2990	-0.3590

Algorithm Improvement: L1-Regularized Least Square

Goal: To force the results to be more piecewise constant

Idea: minimize $||Aw-x||_2^2 + |grad(w)|$

where $|grad(w)| = \sum |w(:, j+1)-w(:, j)| + \sum |w(i+1, :)-w(i, :)|$

Formulation: find u that minimizes

$$m||Au-x||_2^2 + |E_1 u|_1 + |E_2 u|_1$$

u : 144x1 unknown.

u_1-u_{48} : weights of the modes themselves;

u_{49}-u_{96}: weights of gradients in x direction;

u_{97}-u_{144}: weights of gradients in y direction.

E_1 (36x144) : gradient calculation in x direction;

E_2 (36x144): gradient calculation in y direction.

Algorithm Improvement: L1-Regularized Least Square

Find u that minimizes $m||Au-x||_2^2 + |E_1 u|_1 + |E_2 u|_1$

$$A = \begin{bmatrix} [M1\ M2\ M0] & [G1x\ G2x\ G0x] & [G1y\ G2y\ G0y] & & & \\ & & \cdots & \cdots & & \\ & [M1\ M2\ M0] & & [G1x\ G2x\ G0x] & & [G1y\ G2y\ G0y] \end{bmatrix} \quad (16*196X144)$$

$\Rightarrow$ $A*u = \alpha * M_1 + \beta * M_2 + \gamma * M_0 + a_1 * G_1 x + b_1 * G_2 x + c_1 * G_0 x + a_2 * G_1 y + b_2 * G_2 y + c_2 * G_0 y$

$$x = \begin{pmatrix} I_1 \\ I_2 \\ \vdots \\ \vdots \\ I_{16} \end{pmatrix}$$

Each I = a single image

(16*196x1)

$\Rightarrow$ $||Au-x||_2^2$: Total Error of all 16 unit cells.

Method : Split Bregman Method.

Algorithm Improvement: L1-Regularized Least Square

Split Bregman Method:

Initialization: $d_1^0, d_2^0, b_1^0, b_2^0 = 0$ (36x1)

For k=0:N

$u^{k+1}=(mA^TA+\lambda E_1^TE_1+\lambda E_2^TE_2)^{-1}(mA^Tb+\lambda E_1^T(d_1^k-b_1^k)+\lambda E_2^T(d_2^k-b_2^k))$;

$(d_i^{k+1})_i = \text{shrink}((E_iu^{k+1})_i+(b_i^k)_i, 1/\lambda)$, i=1,2;

where shrink(z, a)=(z/|z|)*max(|z|-a, 0)

$$b_1^{k+1} = b_1^k + E_1u^{k+1} - d_1^{k+1};$$
$$b_2^{k+1} = b_2^k + E_2u^{k+1} - d_2^{k+1};$$

Algorithm Improvement: L1-Regularized Least Square

Results:

M_1(Split Bregman):

0.7333	0.7971	-0.9936	-1.0665
0.9198	0.8443	-0.9936	-1.0511
0.9885	0.9051	-0.9082	-0.9804
0.9992	0.9051	-0.9433	-1.0020

M_1(original):

0.8284	0.8117	-0.4186	-0.4570
0.8349	0.8117	-0.4692	-0.5001
0.9280	0.9676	-0.3302	-0.3538
0.9544	0.9881	-0.2990	-0.3590

Total Difference=$\sum$|Calculated Weight – True Weight|

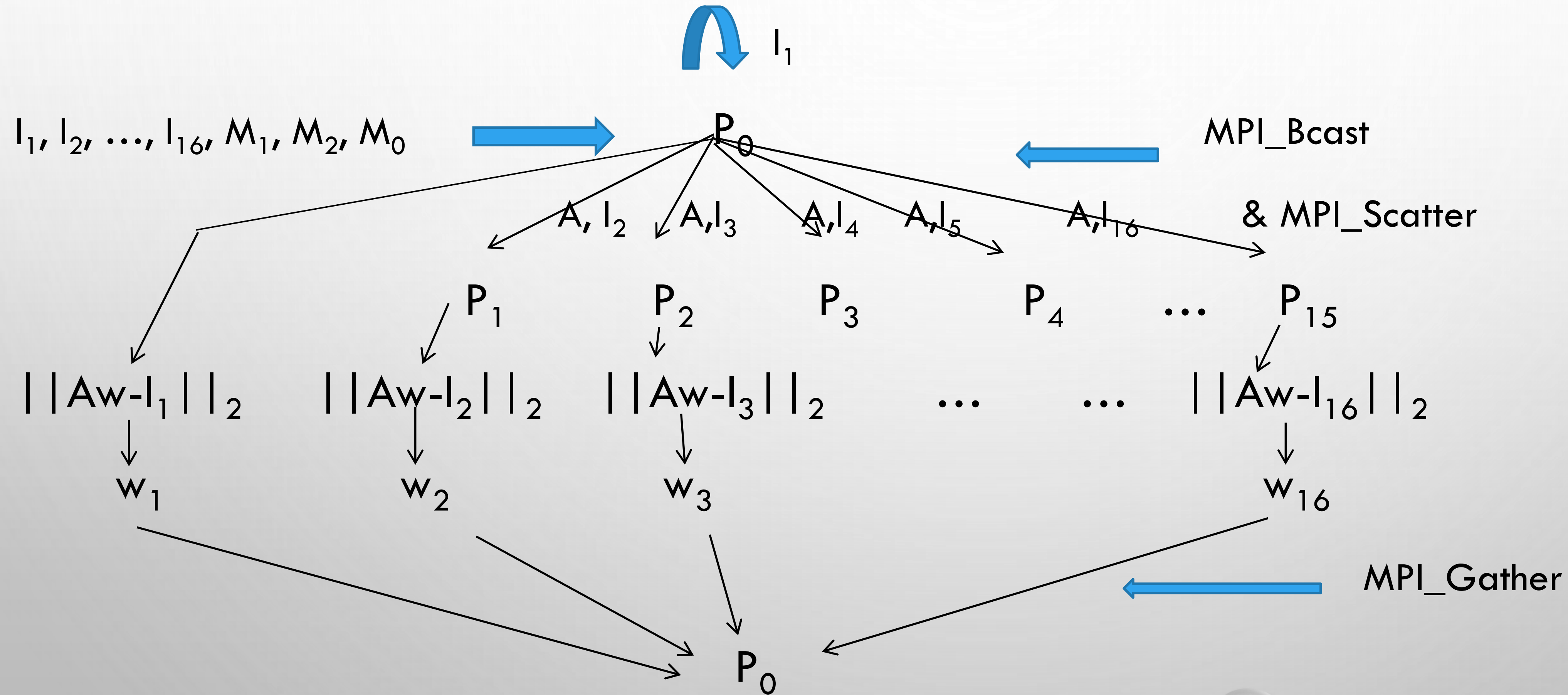
Original Model: 11.6405

Improved Model with Simple Least Square: 2.9635

Improved Model with L1-Regularized Least Square: 2.8833

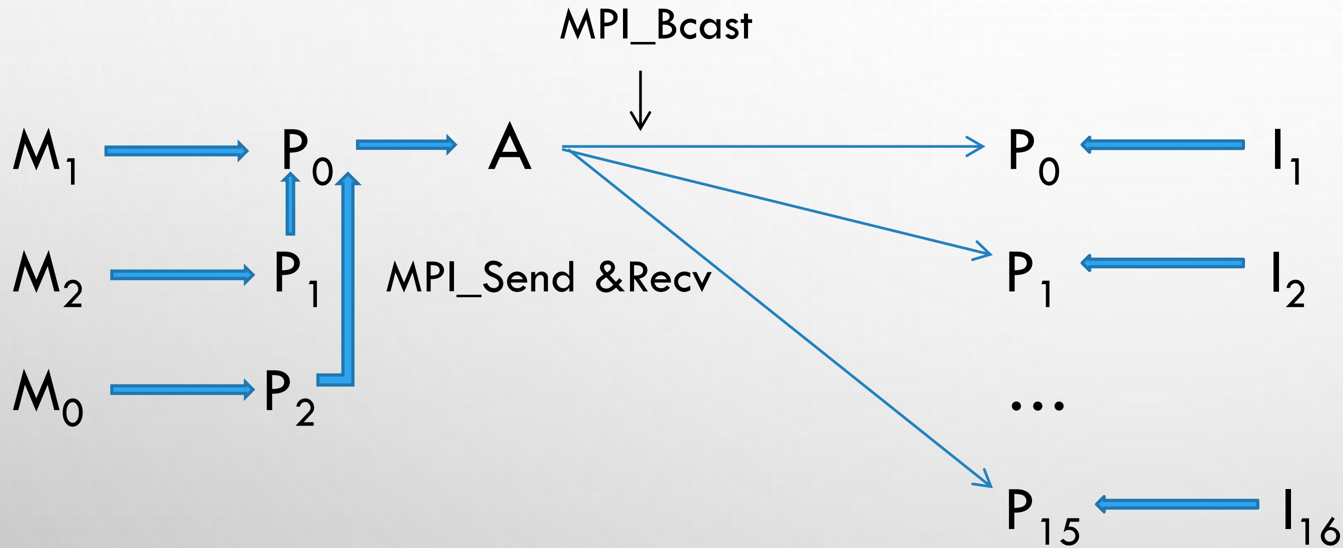
Implementation: Parallel code -> Version 1

Improved Model with Simple Least Square:



Implementation: Parallel code -> Version 2

Improved Model with Simple Least Square



P_i : $||Aw-I_{i+1}||_2$

LAPACK

Using LAPACK was the first of our attempts at improving the time necessary to complete calculations. Since this linear algebra library was made with speed and efficiency in mind, simply using some of these pre-made functions (namely dgels and dgemm) works very well to provide speed up, with a minimum of trouble.

Using Bridges PSC	Basic Least Squares	Simplified LS with Gradient	Split Bregman (20 iterations)
Size of matrix	7,225,344 by 3	196 by 9	144 by 144
Matlab	~ 33 seconds	~ 21 seconds	~ 21 seconds
CPU, using LAPACK	~ 7 seconds	~ 0.75 seconds	~ 0.76 seconds

MAGMA

The next framework we tried was MAGMA, an implementation of LAPACK meant to run on a GPU. Because the main part of the program was still run on a CPU, all the data had to be moved over to the GPU for calculations, and the results had to be moved back to be available to display. All this data movement added considerable overhead, which would be counter-balanced by hopefully reduced time needed for making the calculations.

MAGMA

The idea was that it could work calculations on large matrices much faster, but it turned out that since our matrices were far longer than wide, it couldn't help with calculations much, and in fact took longer than before, likely due to having to copy data to and from the GPU.

Using Bridges PSC	Basic Least Squares	Simplified LS with Gradient	Split Bregman (20 iterations)
Size of matrix	7,225,344 by 3	196 by 9	144 by 144
GPU, using MAGMA	~ 50 seconds	~ 4 seconds	Not made

MAKING PARALLEL

The third basic version attempted was finding some means of running the program in parallel. For this purpose, we tried:

- Message Passing Interface (MPI), which uses various functions for sending and receiving data across multiple processors
- OpenMP, which uses compiler directives to generate parallel executables without the user having to interfere or be too specific
- ScaLAPACK, an implementation of LAPACK made for parallel processing.

Using Bridges PSC	Basic Least Squares	Simplified LS with Gradient	Split Bregman (20 iterations)
Size of Matrix	7,225,344 by 3	196 by 9	144 by 144
Using LAPACK and MPI	~ 53 seconds	~ 1 second	~ 1.5 seconds
Using LAPACK and OpenMP	~ 7.4 to 26 seconds	~ 0.7 to 3 seconds	Not made
Using ScaLAPACK	Not made	~ 30s to 6min	Not made

CONCLUSIONS OF ALGORITHMIC METHOD

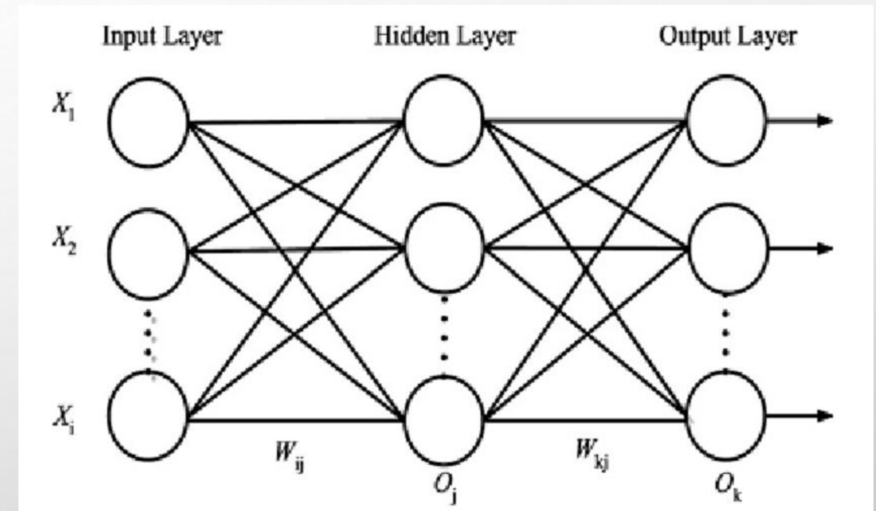
From the algorithmic approach side of this project, the current best approach, as judged by accuracy of results, speed of calculations, and ease of understanding how it works, would have to be the least squares plus gradient implemented with LAPACK.

- It improves upon the basic least squares' accuracy almost as best as we were able
- It ties for fastest program
- And it is a highly understandable solution, without adding many additional complications

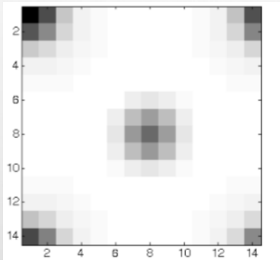
Had the data been better suited to GPU processing, or we had far more than 16 data sets to work through, one of the other methods could have come out ahead, but as it now lies, this is the clear winner.

NEURAL NETWORK

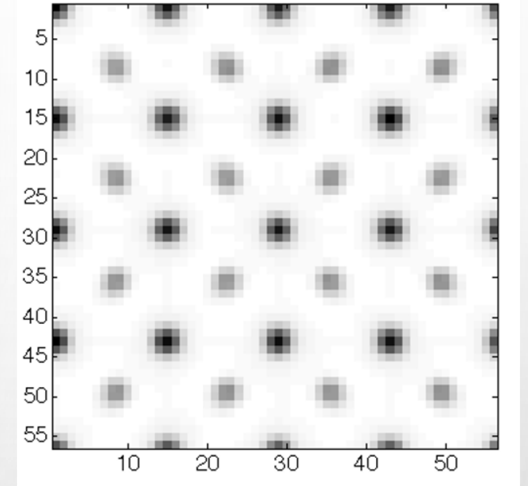
- Neural network is essentially a feed-back mechanism.
- Vectors are propagated forward to produce outputs, and outputs are propagated back to adjust the neural network.
- The behaviours of the layers are similar to the neurons in a biological brain.
- The process of propagation is essentially **matrix multiplication**.



OVERVIEW OF PROBLEM SET UP



- What we have:
  - 3 baseline modes, M_1, M_2, and M_0
  - 16 images, x, each consists of a combination of these 3 modes
  - 3 4x4 true weight matrices



What we want: the true model of the composition of the image, such that the calculated weights of the 3 modes equal the true weights.

Beginning model: linear least squares optimization:

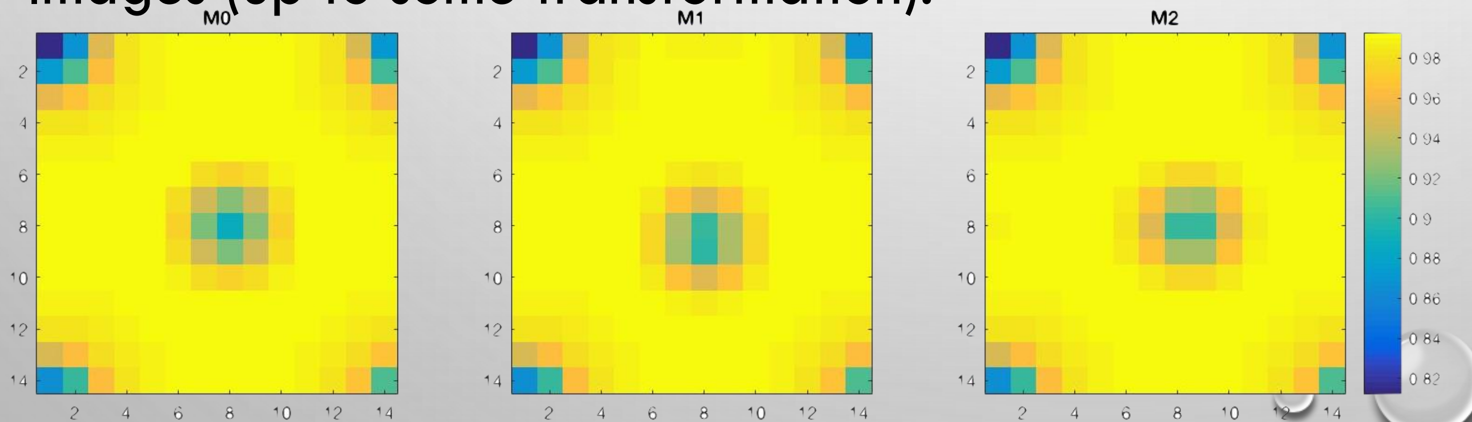
$x = \alpha M_1 + \beta M_2 + \gamma M_0$, with α, β, and γ being the weights of the three modes and x being the resulting image.

MACHINE LEARNING: GOAL

- Let M_0, M_1, M_2 be three modes, and I be a target image. We try to find a linear representation of the input image with the three modes, namely,

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

- The goal: find the coefficients alpha, beta and gamma.
- Images (up to some transformation):



PROBLEM

- The problem is that if you solve the three coefficients directly, using least square method, the result you have will be very bad.
- The reason is that there is a non-linear bias in the dependence. Namely,

$$I = \alpha M_0 + \beta M_1 + \gamma M_2 \ + bias$$

- So working out the bias is an important part of this project.

BIAS

- The method to work out the bias is by interpolation.
- Assume that the bias is caused by the interactions between the M1 and M2 modes
- From the assumption above, for each pixel (x, y) in I , the bias for this pixel is

$$B_{x,y}(\beta, \gamma)$$

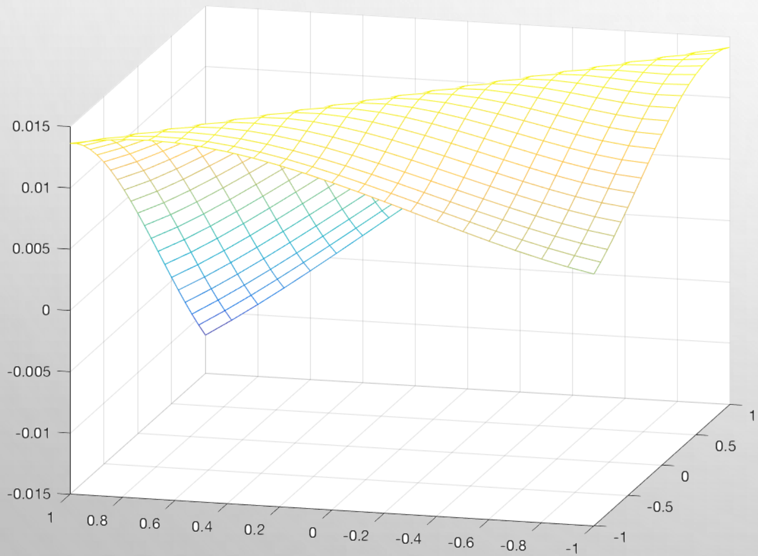
- Which is a function of beta and gamma, though the form of this function is not known.
- We can interpolate bias from the data we know.

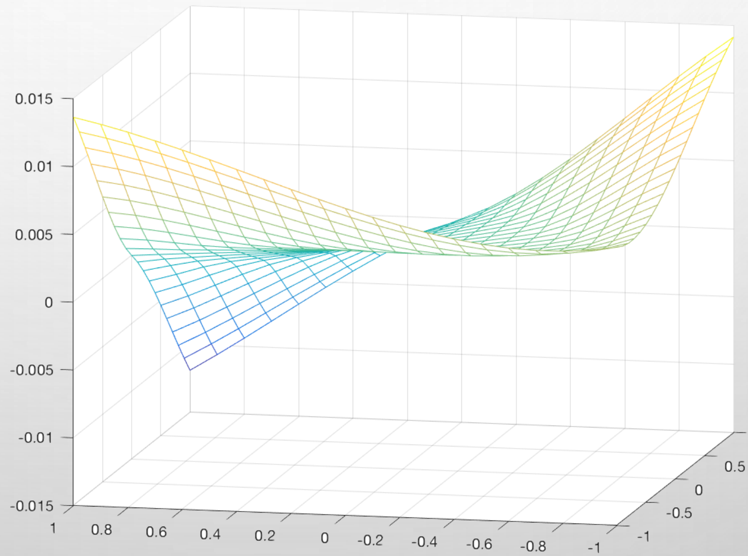
INTERPOLATION

- Recall we have 16 input images, every four of them share the same coefficients.
- Therefore from this dataset we can know the value of the bias function at 4 points, i.e.,
- $B(1,1), B(-1,1), B(1,-1), B(-1,-1)$
- If we take the coefficients of M1 and M2 into account during interpolation, we also know the values of the bias function at two more points:
- $B(0,1), B(1,0)$
- Therefore, we have two kinds of interpolation: 4-point and 6-point.

INTERPOLATION

- 4-point interpolation(cubic): 6-point interpolation(cubic):





Note that in the 4 point interpolation, The surface is more smooth; for the 6 point interpolation, more points makes the interpolation more complicated, so the surface is not that smooth.

CHOICE OF INTERPOLATION METHOD

- Synthetic training examples are generated by:
  - 1. Pick β, γ randomly from the interval (-1,1) and fix $\alpha = 1$.
  - 2. Generate the example by taking the linear combination and then adding the bias term:

$$I_{synthetic} = \alpha M_0 + \beta M_1 + \gamma M_2 + B(\beta, \gamma)$$

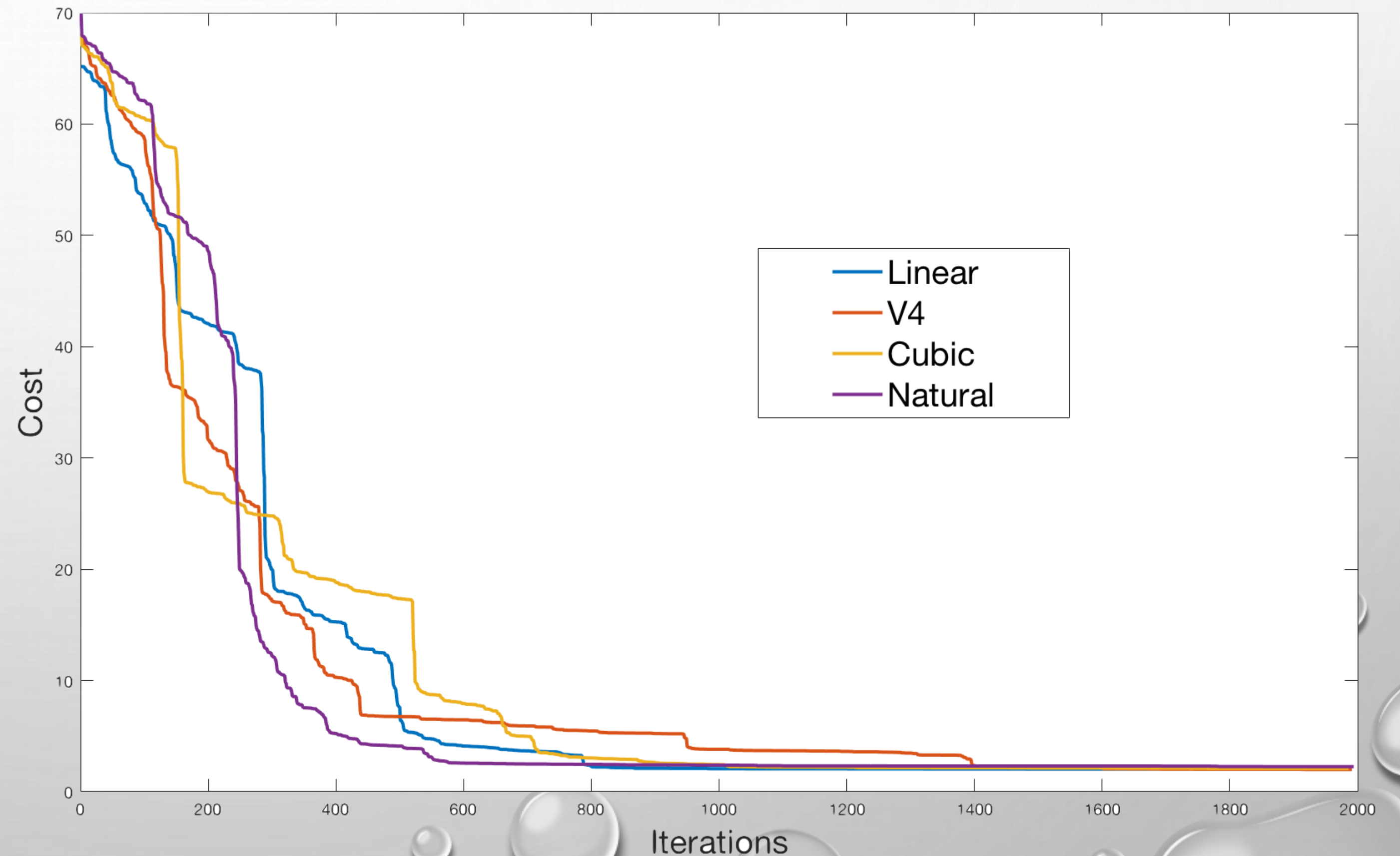
- The computations are carried out with following parameters:
  - 2 hidden layers, hidden layer size = 15;
  - Regularisation parameter = 0.05;
  - 200 synthetic examples;
  - A CG method for learning.

COMPARISON BETWEEN INTERPOLATION METHODS

- Use 4-point interpolation.

Method	Cost
Linear	0.0712
V4	0.4695
Cubic	0.0620
Natural	0.0590

Note that all interpolation methods works fine except the v4 interpolation(biharmonic). So in the future computations, linear, cubic and natural neighbour interpolation will be used.



PREDICTION RESULTS

- The prediction with 4-point cubic interpolation:

$$\alpha : \begin{bmatrix} 0.9863 & 0.9880 & 0.9872 & 0.9877 \\ 0.9864 & 0.9867 & 0.9872 & 0.9887 \\ 0.9886 & 0.9892 & 0.9826 & 0.9866 \\ 0.9900 & 0.9901 & 0.9842 & 0.9851 \end{bmatrix}, \qquad \alpha_{true} : \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\beta : \begin{bmatrix} 0.9228 & 1.0626 & 0.9539 & 1.0231 \\ 1.0188 & 0.8923 & 0.9583 & 0.9958 \\ -0.9593 & -1.0013 & -0.9705 & -0.9772 \\ -0.9475 & -1.0325 & -0.9663 & -1.0108 \end{bmatrix}, \qquad \beta_{true} : \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix}$$

$$\gamma : \begin{bmatrix} 0.9163 & 1.0083 & -1.0100 & -1.0174 \\ 1.0067 & 0.9658 & -0.9789 & -0.9377 \\ 0.9545 & 0.9097 & -1.0054 & -0.9915 \\ 1.0069 & 1.0326 & -0.9709 & -0.9524 \end{bmatrix}. \qquad \gamma_{true} : \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \end{bmatrix}$$

MORE TEST

- With 4-point natural interpolations:

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

- The true coefficients are:

$$M_0 : \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad M_1 : \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad M_2 : \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

ON GPU

- CPU-GPU communication is very time consuming.
- Trick: All large data structures should not leave GPU, to reduce such communication.
- New CUDA routines implemented to keep data on GPU.
- Elementwise functions and elementwise product is implemented.
- Part of the code:

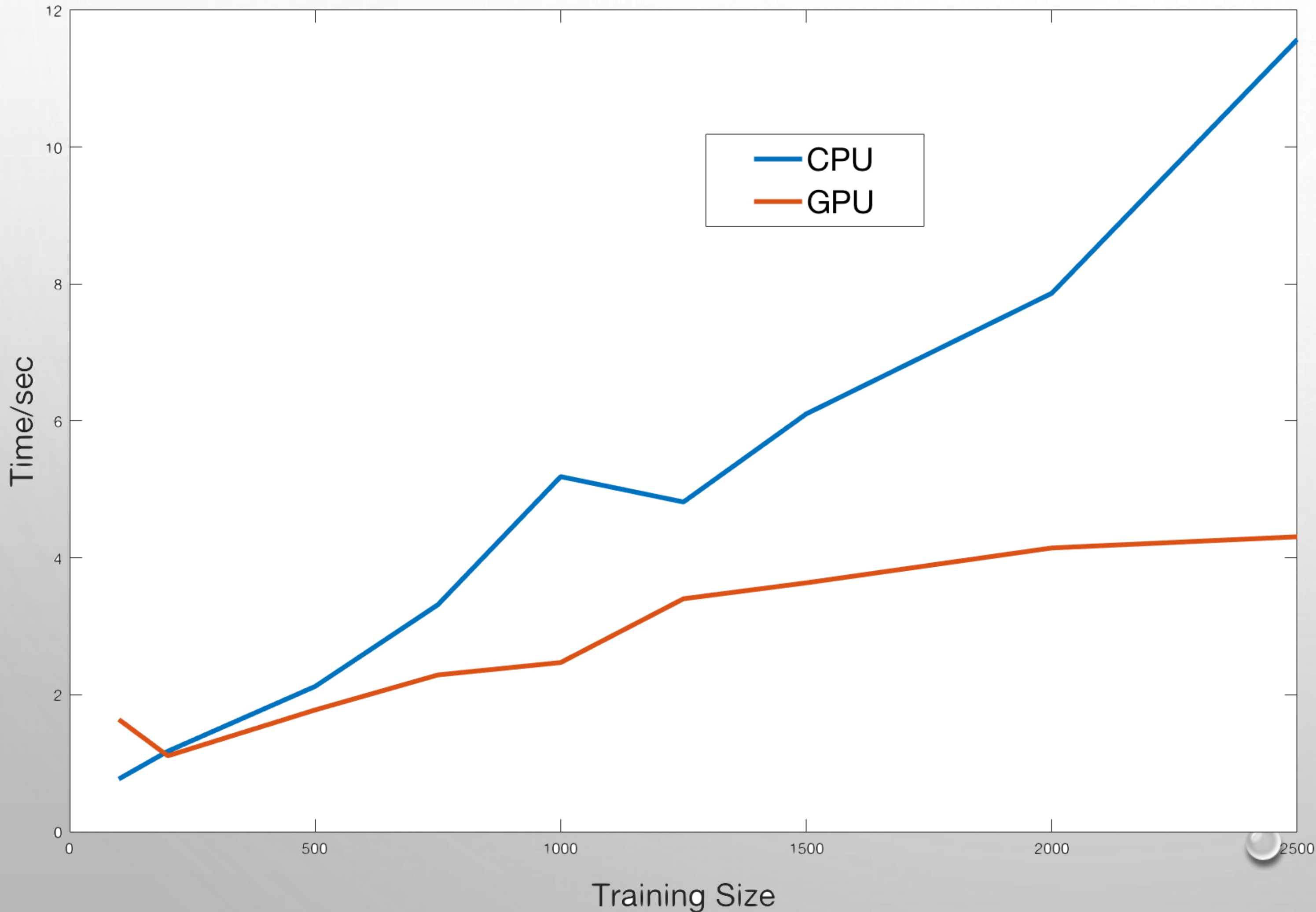
```
if ( ind < m ) {
    dA += ind + iby*ldda;
    dB += ind + iby*lddb;
    if ( full ) {
        // full block-column
        #pragma unroll
        for( int j=0; j < BLK_Y; ++j ) {
            dB[j*lddb] = MAGMA_D_TANH(dA[j*ldda]);
        }
    }
    else {
        // partial block-column
        for( int j=0; j < BLK_Y && iby+j < n; ++j ) {
            dB[j*lddb] = MAGMA_D_TANH(dA[j*ldda]);
        }
    }
}
```

PERFORMANCES

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

NVIDIA P100 GPU is used.

PERFORMANCES

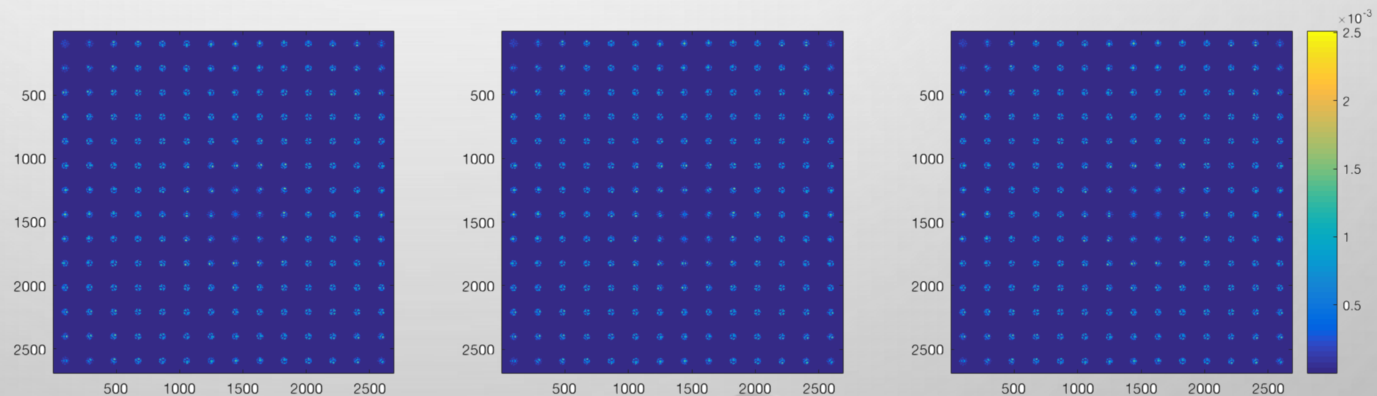


Note: the CPU curve is increasing sharply, looking like a straight line. While the GPU curve is smooth and the value is not high even for large datasets.

MACHINE LEARNING :SUMMARY

- Why use machine learning method for this problem?
- This way we don’t have to analyse the detailed structure and the features of the image.
- It is really troublesome to analyse this kind of image.

Three basic modes:



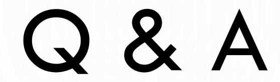
MORE…

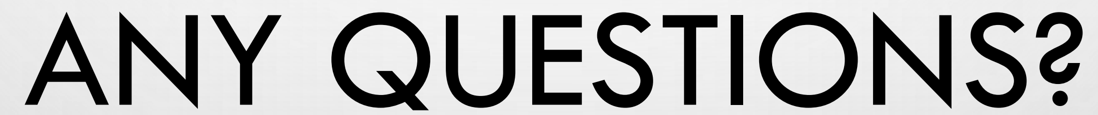
- More thinking: essentially what this neural network is doing is to solve an overdetermined nonlinear system.
- Then why don't we extend this idea further? Maybe we can solve a big linear system with neural network.
- Or maybe even other linear algebra problems may be solved with neural network.

COMPUTE MATRIX INVERSE

- For invertible matrix A ,
- Cost function: $J(\Theta) = \Sigma_{i=1}^{n} ||A\Theta b_i - b_i||_2^2$
- Gradient $\Sigma_i A^T (A\Theta b_i - b_i) b_i^T$
- For the gradient, I prove that the spectral radius is
-
- $max_{m,j}|1 - \Delta t \lambda^{(j)} \sigma_m^2|$
-

Δt is the time step,
$\lambda^{(j)}$ are eigenvalues of $\Sigma_i b_i b_i^T$
σ_m are singular values of A


Q & A


ANY QUESTIONS?