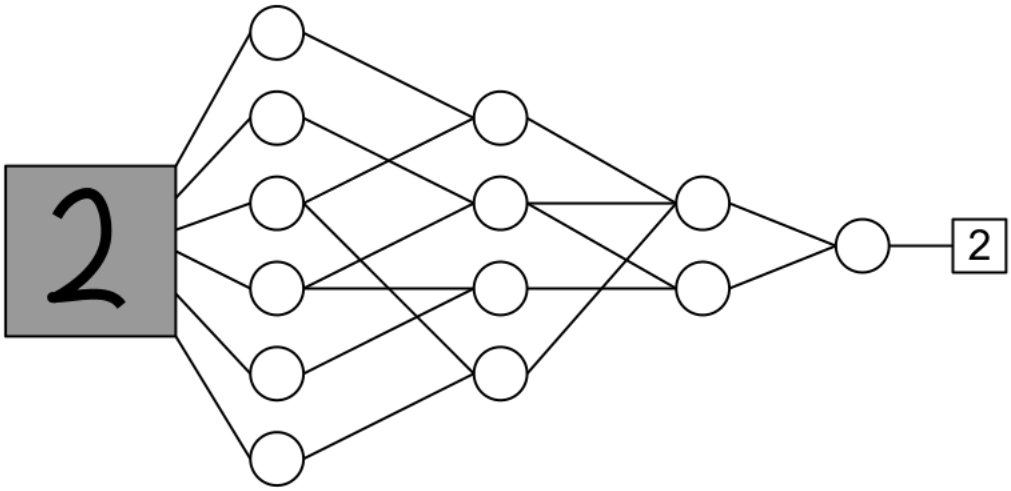
CS 789 Parallel Programming
Parallel Feed-forward Neural Network: Handwritten Digit Recognition

Lucas Bang



Contents

1 Background

Neural networks are known to be a reliable method for automatic handwritten digit recognition. These systems are currently used by the United States Postal Service to sort mail by zip code and to recognize dollar amounts on hand written checks in ATMs. For this project I have implemented a neural network that accomplishes this same task in parallel using The MNIST Database of Handwritten Digits. While the size of the network required for this task is small enough to run quickly when implemented sequentially, larger neural networks for more complicated tasks could easily require parallelization to classify images quickly.

The basic component of a neural network is the perceptron. A perceptron can be thought of as an abstract model of a neuron in a brain. A perceptron takes weighted inputs, computes an activation function on the total weighted input and outputs an appropriate value. For this network, we use the sigmoid activation function given by $sigmoid(x) = \frac{1}{1+e^{-x}}$, as in the figure, and output that value from the perceptron as illustrated below. The value of the sigmoid function is always between 0 and 1, and may be be thought of as a probability that the perceptron is correctly classifying its input.

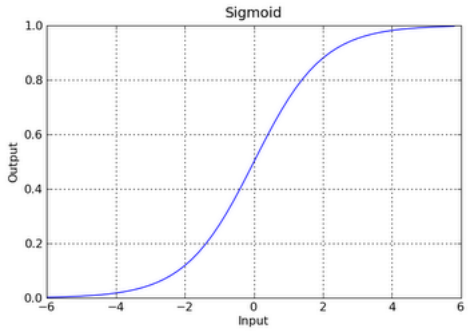


Figure 1: The sigmoid function.

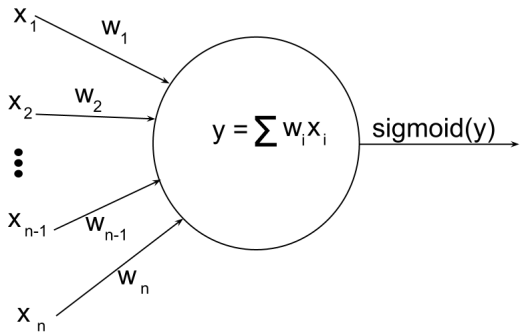


Figure 2: How a perceptron computes.

2 The Architecture of Our Digit Recognizing Network

We can construct a neural network from a large set of perceptrons as in the following figure. In order to correctly operate, the neural network must determine the weights to place on the input edges. This involves a very complicated back propagation algorithm. This algorithm was previously implemented as part of a project to learn about neural networks. Implementing the back propagation algorithm in parallel is beyond the scope of this project and possibly beyond my debugging ability given the time constraints. Thus, here we implement only the feed forward classification pass of the algorithm, utilizing the weights learned using previously run sequential code.

Each image is 20 pixels by 20 pixels fro a total of 400 pixels. The lead process will break of the image into rows, and send a set of rows to a each Layer 1 process. When a layer 1 process receives a set of pixels, it simulates the activation of the neurons it is in charge of, and sends the activations to all 25 processes in the following layer. We think of the connectivity graph between two layers as a complete bipartite graph, $K_{m,n}$. Then, each activation in Layer 2 is computed and passed to Layer 3. Layer 3 contains 10 neurons. Each neuron in Layer 3 uses the information from Layer 2 to compute a probability of each possible digit. The final neuron, held by the lead simply computes the maximum of these values and outputs that number as the

chosen classification of the image. The code for this is included in Appendix A.

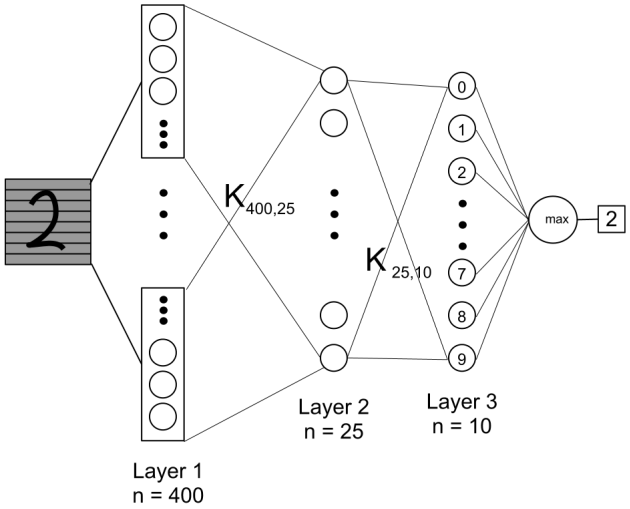


Figure 3: Our Nerual Network Architecture.

3 Results

The neural network performed well. Out of 5000 examples, it was able to correctly classify 4955, a rate of 99.1%. The following image contains all 45 images that were incorrectly classified. A condensed copy of the output is shown here:

```
45  Theta 1 has dimensions 25 by 401
45  Theta 2 has dimensions 10 by 26
45  Finished Loading Data
45  Finished Loading Images
45  4955 CORRECT!
```

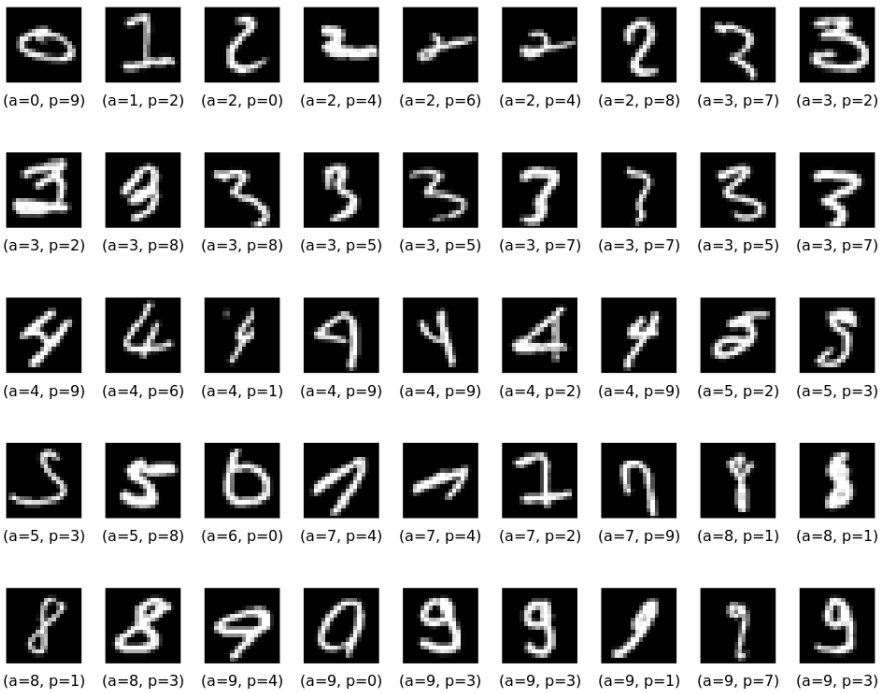


Figure 4: Incorrectly classified images. The label p represents the digit predicted by the network and the label a represents the actual digit.

A Parallel Code for Feed Forward Neural Network

Listing 1: Feed Forward Neural Network

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <sys/time.h>
#include <math.h>

void printMatrix(float * matrix, int n, int m){
  int i, j;
  for(i = 0; i < n; i++){
    for(j = 0; j < m; j++){
      printf("%f ", matrix[i*m + j]);
    }
    printf("\n");
  }
  printf("\n");
}

void copy_row_out(float * matrix, float * row, int row_num, int n, int m){
  int i, j;

  for(i = 0; i < m; i ++){
    row[i] = matrix[row_num*m + i];
  }

}

void readMatrix(char * file_name, float * matrix, int n, int m){
  FILE *f;
  int i,j;
  f = fopen(file_name,"r");

  for(i = 0; i < n; i++){
    for(j = 0; j < m; j++){
      fscanf(f,"%f ", &matrix[i*m + j]);
    }
  }
  fclose(f);
}

void pad_image(float * image, float* paddedimage, int n){
  paddedimage[0] = 1;
  int i;

  for(i = 0; i < n; i++)
    paddedimage[i+1] = image[i];

}

void print_row(float * row, int n, int rank){
  int i;
  for(i = 0; i< n; i++){
    printf("%3d\t: %3d: %3.6f \n ", rank, i, row[i]);
  }
  printf("\n");
}

float sigmoid(float x){
  return 1.0 / (1.0 + exp(-x));
}

int main(int argc, char *argv[]) {
  int   size, rank, i, j, k, masterID;;
  int   n1, m1, n2, m2, layer2size;
  int   test_set_size = 5000;
  int   neurons_per_layer1 = 40;
  int   neurons_per_layer2 = 1;
  int   imgsize = 20;
  int   iterations = 5000;
```

```
  FILE *  f;

  MPI_Status status;
  char  hostname[256];
  gethostname(hostname,255);

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  masterID = size-1;


  n1  = atoi(argv[2]);
  m1  = atoi(argv[3]);
  n2  = atoi(argv[5]);
  m2  = atoi(argv[6]);
  layer2size = n1;

  if(rank == masterID){
    float *   theta1;
    float *   theta2;
    float *   image;
    float *   allImages;
    float *   padded_image;
    float *   Y;
    float     layer3activation;
    float     max_activation = -1;
    int     img_h = 20;
    int     img_w = 20;
    int     num_pix = img_h * img_w;
    int     their_start, their_end, amt_to_send;
    int     layer2recipient, layer3recipient;
    int     msg_size, prediction, actual;
    int     num_correct = 0;


    theta1    = (float *) malloc(sizeof(float)*n1*m1);
    theta2    = (float *) malloc(sizeof(float)*n2*m2);
    image     = (float *) malloc(sizeof(float)*num_pix);
    Y       =(float * ) malloc(sizeof(float)*test_set_size);
    padded_image= (float *) malloc(sizeof(float)*(num_pix + 1));
    allImages   = (float *) malloc(sizeof(float)*test_set_size*(m1-1));


    printf("%3d\t: Theta 1 has dimensions %d by %d \n", rank, n1, m1);
    printf("%3d\t: Theta 2 has dimensions %d by %d \n", rank, n2, m2);
    printf("%3d\t: Theta 1 stored in %s \n", rank, argv[1]);
    printf("%3d\t: Theta 2 stored in %s \n", rank, argv[4]);

    readMatrix(argv[1], theta1, n1, m1);
    readMatrix(argv[4], theta2, n2, m2);
    readMatrix(argv[7], allImages, test_set_size, (m1-1));
    readMatrix(argv[8], Y, test_set_size, 1);

    //printMatrix(theta2, n2, m2);

    printf("%3d\t Finished loading data\n", rank);

    //load an image and pad with 1 for bias
    for(k = 0; k < iterations; k++){
      max_activation = -1;
      copy_row_out(allImages, image, k, test_set_size, (m1-1));
      //print_row(image, num_pix);
      pad_image(image,padded_image, num_pix);
      //print_row(padded_image, num_pix + 1);
      //printf("%3d\t Finished loading image %d\n", rank, k);


      // break up image and send a piece to the process
      // that is in charge of those neurons in the first layer
      for(i = 0; i < (m1 - 1) / neurons_per_layer1; i++){
        if(i == 0){their_start = 0;}
        else{their_start = i*neurons_per_layer1 + 1;}
        their_end = (i + 1) * neurons_per_layer1 + 1;
        amt_to_send = their_end - their_start;
        MPI_Send(&image[their_start], amt_to_send, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
```

```
      }

      // distribute Theta1 to second layer each node gets one row
      for(i = 0; i < n1 ; i++){
        layer2recipient = i + (m1 - 1) / neurons_per_layer1;
        MPI_Send(&theta1[i*m1], m1, MPI_FLOAT, layer2recipient, 0, MPI_COMM_WORLD);
      }

      for(i = 0; i < n2 ; i++){
        layer3recipient = i + (m1 - 1) / neurons_per_layer1 + n1 ;
        MPI_Send(&theta2[i*m2], m2, MPI_FLOAT, layer3recipient, 5, MPI_COMM_WORLD);
      }

      for(i = n1+n2; i < n1 + 2 * n2; i++){
        MPI_Recv(&layer3activation, 1, MPI_FLOAT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if(layer3activation > max_activation){
          max_activation = layer3activation;
          prediction = i-34;
        }
      }
      actual = (int)Y[k];
      //printf("%3d\t: prediction = %d actual = %d\n", rank, prediction, actual);
      if(prediction == actual){ num_correct++;}
      else{
        printf("Incorrect: Prediction = %3d Actual = %3d Image = %3d \n", prediction, actual, k+1);
      }

    } //end for loop
    printf("%3d\t %d CORRECT!\n", rank, num_correct);
  }
  //=================================================================
  // FIRST LAYER
  else if (rank < (m1-1)/neurons_per_layer1){
    float *   theta1;
    float *   theta2;
    float *   image;
    float *   my_activations;
    int     my_start, my_end, amt_to_recv, amt_to_send;
    int     layer2recipient;

    if(rank == 0){my_start = 0;}
    else{my_start = rank*neurons_per_layer1 + 1;}
    my_end = (rank + 1) * neurons_per_layer1 + 1;

    // get a piece of the image from the master
    amt_to_recv = my_end - my_start;
    amt_to_send = amt_to_recv;

    image  = (float *) malloc(sizeof(float)*amt_to_recv);
    my_activations = (float *) malloc(sizeof(float)*amt_to_recv);

    for(k = 0; k < iterations; k++){
      MPI_Recv(image, amt_to_recv, MPI_FLOAT, masterID, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      for(i = 0; i < amt_to_recv; i++){
        my_activations[i] = (image[i]);
      }
      for(i = 0; i < n1  ; i++){
        layer2recipient = i + (m1 - 1) / neurons_per_layer1;
        MPI_Send(my_activations, amt_to_send, MPI_FLOAT, layer2recipient, 0, MPI_COMM_WORLD);
      }
    } // end for loop
  }
  //=================================================================
  // SECOND LAYER
  else if (rank < (m1-1)/neurons_per_layer1 + layer2size  ){
    float * theta1;
    float * theta2;
    float * image;
    float * activations_level1;
    float * my_activations;
    int amt_to_recv;
    int their_start, their_end;
    float my_activation = 0.0;
    float * padded_image;

    theta1 = (float *) malloc(sizeof(float)*m1);
    activations_level1 = (float*) malloc(sizeof(float)*m1);
```

```
    theta2 = (float *) malloc(sizeof(float)*n2*m2);
    image  = (float *) malloc(sizeof(float)*imgsize*imgsize);
    padded_image  = (float *) malloc(sizeof(float)*(imgsize*imgsize + 1));

    MPI_Recv(theta1, m1, MPI_FLOAT, masterID, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    for(k = 0; k< iterations; k++){
      for(i = 0; i < (m1 - 1) / neurons_per_layer1; i++){
        if(i == 0){
          their_start = 0;
        }
        else{
          their_start = i*neurons_per_layer1 + 1;
        }
        their_end = (i + 1) * neurons_per_layer1 + 1;
        amt_to_recv = their_end - their_start;
        MPI_Recv(&activations_level1[their_start], amt_to_recv, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &status);
      }

      pad_image(activations_level1, padded_image, m1-1);
      for(i = 0; i < m1; i++){
        my_activation += padded_image[i]*theta1[i];
      }
      my_activation = sigmoid(my_activation);

      for(i = 35; i < 45; i++){
        MPI_Send(&my_activation, 1, MPI_FLOAT, i, 6, MPI_COMM_WORLD );
      }
    }//end for loop
  }//================================================================
  // THIRD LAYER
  else{
    float * theta2;
    theta2 = (float *) malloc(sizeof(float)*m2);
    MPI_Recv(theta2, m2, MPI_FLOAT, masterID, 5, MPI_COMM_WORLD, &status);
    float activation_layer2;
    float my_activation = theta2[0];
    for(k = 0; k < iterations; k++){
      my_activation = theta2[0];
      j = 1;
      for(i = 10; i < 35; i++){
        MPI_Recv(&activation_layer2, 1, MPI_FLOAT, i, 6, MPI_COMM_WORLD, &status);
        my_activation += theta2[j]*activation_layer2;
        j++;
      }
      MPI_Send(&my_activation, 1, MPI_FLOAT, masterID, 0, MPI_COMM_WORLD);
    }//end for loop
  }
  MPI_Finalize();
  return 0;
}
```