Project 3: Classification

CSE 574 – Machine Learning

Karanjeet Singh (Person No: 50169916); karanjee@buffalo.edu

Overview:

The project was to implement classification and evaluate performance of models: Logistic Regression, Single Hidden Layer Neural Network and Convoluted Neural Networks. We were provided with a real world set data from MNIST digit images and the three models were to be run using mini-batch gradient descent.

Dataset:

The dataset consisted of 60000 datapoints as the training set and 10000 points as testing set. The original black and white (bilevel) images from MNIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

The images were converted into 784 features using the following code in matlab:

```
filename1='C:\Users\karanjeet\Downloads\train-images-idx3-ubyte\train-images.idx3-ubyte';
fp1 = fopen(filename1, 'rb');
assert(fp1 ~= -1, ['Could not open ', filename1, '']);
magic = fread(fp1, 1, 'int32', 0, 'ieee-be');
assert(magic == 2051, ['Bad magic number in ', filename1, '']);
numImages = fread(fp1, 1, 'int32', 0, 'ieee-be');
numRows = fread(fp1, 1, 'int32', 0, 'ieee-be');
numCols = fread(fp1, 1, 'int32', 0, 'ieee-be');
train_images = fread(fp1, inf, 'unsigned char');
train_images = reshape(train_images, numCols, numRows, numImages);
train_images = permute(train_images,[1 2 3]);
fclose(fp1);
train_images = reshape(train_images, size(train_images, 1) * size(train_images, 2), size(train_images, 3));
train_images = double(train_images) / 255;
```

The labels/actual_output is also in the form of images and is to be converted to the computational form:

```
fp2 = fopen(filename2, 'rb');
assert(fp2 ~= -1, ['Could not open ', filename2, '']);
magic = fread(fp2, 1, 'int32', 0, 'ieee-be');
assert(magic == 2049, ['Bad magic number in ', filename2, '']);
numLabels = fread(fp2, 1, 'int32', 0, 'ieee-be');
train_labels = fread(fp2, inf, 'unsigned char');
assert(size(train_labels,1) == numLabels, 'Mismatch in label count');
```

Hyper-parameters: **Logistic Regression**

We initialize the hyper-parameters to some value and then work on them to get the best results. The parameters are:

```
b_size = 30;                          // batch size
iterations = 20;                      // iterations over the data
w0 = randn(features,k)*0.1;           // initial random weights
n_a = 1;                              // parameterizing the learning rate
n_b = 1;                              // second parameter for learning rate to make it adaptive
w = w0;                               // wt vector
rand_train = randperm(train_size);    // randomizing the weight vector for better performance
counter = 0;                          // initializing the counter for the iterations
neta = n_a / (n_b+counter);           // formula for adaptive learning rate
b_start = 1;                          // batch start position for each iteration
train_size=60000;                     // training data size, it was 55000 earlier as 10% was for validation
test_size=10000;                      // testing data size
features=784;                         // total number of features for each datapoint
k=10;                                 // number of possible values of classification
```

We then convert the output labels to the k format:

eg: 3 is repersented as 1by10 vector with value [0 0 0 1 0 0 0 0 0 0]

```
testlabels=zeros(1,test_size);
for i=1:test_size
   [~,testlabels(i)] = max(test_output_k_format(i,:));
end
```

Training using Logistic Regression:

We train till all the iterations are over and we iterate over each datapoint in a batch and change the weights after each batch is done. Thus this is a mixture of stochastic and linear gradient descent.

```
while(counter < iterations)                          // to continue till all iterations are over

   b_stop = min(train_size,b_start+b_size-1);        // the end point of each batch
   curr_train_design_mat = train_design_mat(rand_train(b_start:b_stop),:);
// current training design matrix
   curr_train_output_k_format = train_output_k_format(rand_train(b_start:b_stop),:);
// current train output in k format
   curr_train_size = size(curr_train_design_mat,1);
// train size of batch input vector
   curr_train_output_k_format_size = size(curr_train_output_k_format,2);
// train size of batch output vector
   logit_err = zeros(size(w));
// initializing the error
```

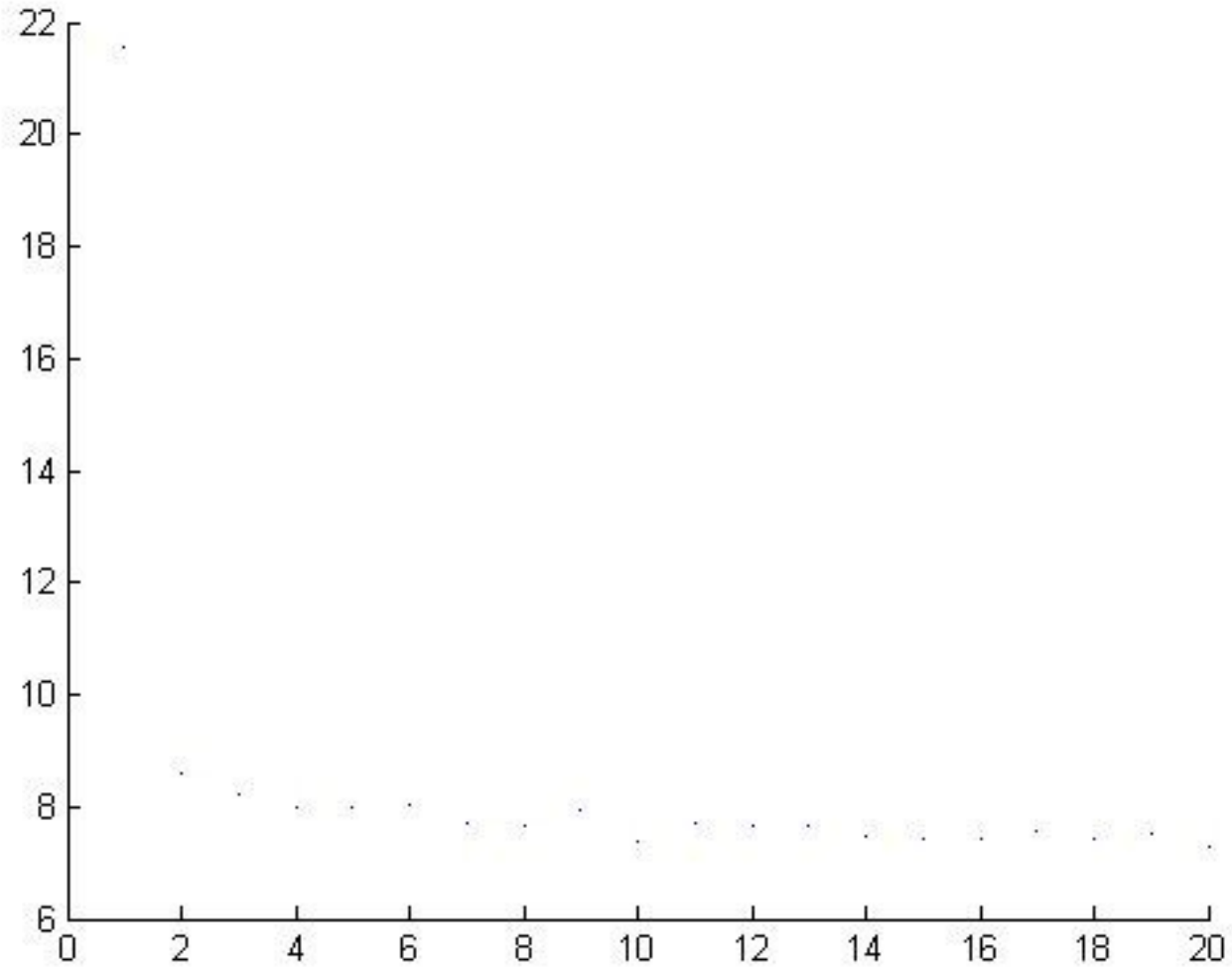
```
// for each of the training sample in the batch we apply the formulas for logistic regression and find the error for each datapoint and cumulate it.
   for n = 1:curr_train_size
      pn = exp(w'*curr_train_design_mat(n,:)');
      yn = pn/sum(pn);
      for i=1:curr_train_output_k_format_size
         logit_err(:,i)=logit_err(:,i)+(yn(i)–curr_train_output_k_format(n,i))*
                        curr_train_design_mat(n,:)'
      end
   end

   logit_err = (1/curr_train_size)*logit_err;          // normalizing error
   dw = logit_err;
   w = w – neta*dw;                                    // changing the wt vector according to the error and eta
   b_start = b_start+b_size;                           // moving to the next batch

// finding the error on the training set after each iteration
// we loop over the input of the testing/validation set and apply the weights to get the desired value.
// we then check the corresponding actual output value. We club all the mis-classifications and return the error rate for the particular iteration.

   if(b_start>train_size)
      trained_val=zeros(1,test_size);
      for i=1:test_size
         as = w'*test_design_mat(i,:)';
         [~,trained_val(i)] = max(as);
      end
      errorrate = 1 - sum(trained_val(:)==testlabels(:)) / size(test_design_mat,1);
      fprintf('Test error = %.1f%%, neta = %f, iteration = %d\n', errorrate*100, neta, counter+1);

      b_start = 1;                                     // reinitializing the batch to start from first datapoint
      counter = counter+1;                             // increasing the counter for the iterations
      neta = n_a / (n_b+counter);                      // adapting the learning rate
      rand_train = randperm(train_size);               // randomizing the datapoints for the next iteration
   end
end

Wlr=w;
blr=zeros(1,10);                    // variables for the mat file
```

Variations are done for the hyper-parameters and the best value is picked which gives us the least error.

For me the tuned hyper-parameters values are:

Batch size: 30
Iterations: 20
learning rate: adaptive (decreasing) starting from 1.

Tuned the batch size and iterations, after 20 iterations there was not any change in the error.



Single Hidden Layer Neural Network:

Hyper-parameters:

```
hidden_layer = 500;                 // perceptrons in the hidden layer
neural_neta = 0.5;                  // learning rate
neural_b_size = 100;                // batch size
neural_b_start = 1;                 // starting row of batch for each iteration
neural_iterations = 5;              // number of iterations
hidden_wts = rand(hidden_layer, features);              // hidden weights initialization
neural_out_wts = rand(k, hidden_layer);                 // output layer weights initialization
hidden_wts = hidden_wts./size(hidden_wts, 2);           // normalyzing the hidden wts
neural_out_wts = neural_out_wts./size(neural_out_wts, 2);       // normalyzing the output wts
n = zeros(neural_b_size);
targetValues=transpose(train_output_k_format);                  // format for training
cc=0;                                                   // counter for iterations
```

Training:

```
while(cc<neural_iterations)                    // loop for iterations
        // batch size stop limit
   neural_b_stop = min(train_size,neural_b_start+neural_b_size-1);
        // current training batch datapoints
   curr_train_design_mat = train_images(:,rand_train(neural_b_start:neural_b_stop));
        // current training output in k format
   curr_train_output_k_format = targetValues(:,rand_train(neural_b_start:neural_b_stop));
        // current train size
   curr_train_size = size(curr_train_design_mat,1);
        // current output size
   curr_train_output_k_format_size = size(curr_train_output_k_format,2);
```

// looping over the dataset in the batch, using sigmoid function here.
// we take each datapoint and multiply it with the hidden inp (initially random weights) vector. We apply the logistic formula on the hidden inp to get the hidden output. Similarly we do for the second later or the output layer. We perform the feed-forward and backward propagation. Later, we update the weights as we learnt them for a data-point.

```
   for j = 1: neural_b_size

      train_line = curr_train_design_mat(:, j);          // training datapoint
      hidden_inp = hidden_wts*train_line;                // hidden layer wts input
      hidden_out = 1./(1 + exp(-hidden_inp));            // hidden layer output
      second_inp = neural_out_wts*hidden_out;            // output layer input
      neural_out_line = 1./(1 + exp(-second_inp));       // output layer output
      train_out_line = curr_train_output_k_format(:, j); // actual output

     second_err = (1./(1 + exp(-second_inp)).*(1 - 1./(1 + exp(-second_inp)))).*(neural_out_line -
                  train_out_line);
// error for the output layer wts
     hidden_err=(1./(1+exp(-hidden_inp)).*(1-1./(1+exp(-hidden_inp)))).*
                  (neural_out_wts'*second_err);
// error for the hidden layer wts

// formula for back-propagation
      neural_out_wts = neural_out_wts - neural_neta.*second_err*hidden_out';
      hidden_wts = hidden_wts - neural_neta.*hidden_err*train_line';
   end;

   neural_b_start = neural_b_start+neural_b_size;          // batch size increasing

// After an iteration is done, we check the error rate for that iteration

   if(neural_b_start>train_size)                           // when one iteration gets over

      trained_val=zeros(1,test_size);
      test_input = test_images;                        // testing/validtion set output
```

// using the learnt weights, we use the testing design matrix, apply the weights for each later and find the output as found by the model. Now we check the valur against the actual output for the testing datapoint. We do so for each iteration.

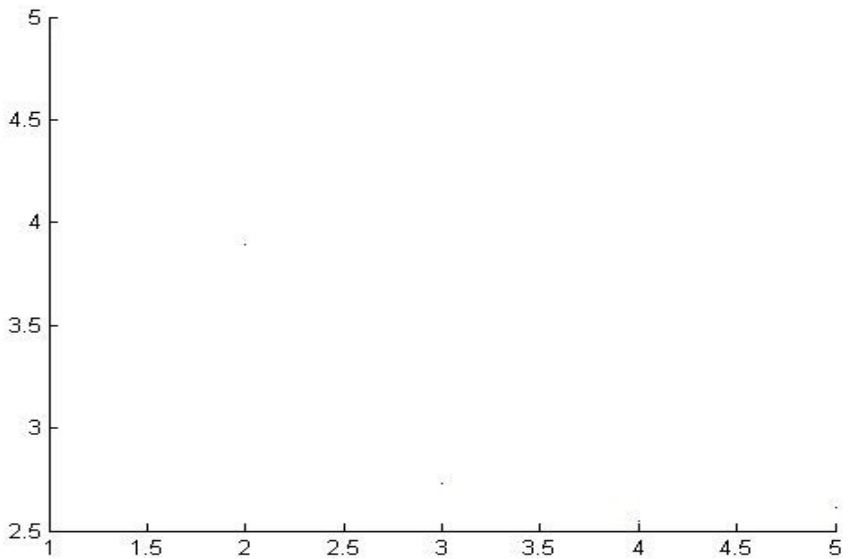
```
    for i=1:test_size
       test_inp_line = test_input(:, i);
       // finding output from input and wts (hidden and output layer)
       neural_out_line = 1./(1 + exp(-(neural_out_wts*(1./(1 + exp(-(hidden_wts*test_inp_line)))))));
       [~,trained_val(i)] = max(neural_out_line);
    end

    errorrate = 1 - sum(trained_val(:)==testlabels(:)) / size(test_design_mat,1);
    fprintf('Test error = %.1f%%, iteration = %d\n', errorrate*100, cc+1);

    neural_b_start = 1;                          // initialization of the batch
    cc = cc+1;                                   // incrementing the counter
    rand_train = randperm(train_size);           // randomizing the training datapoints
  end
end;
```

// variables for mat file

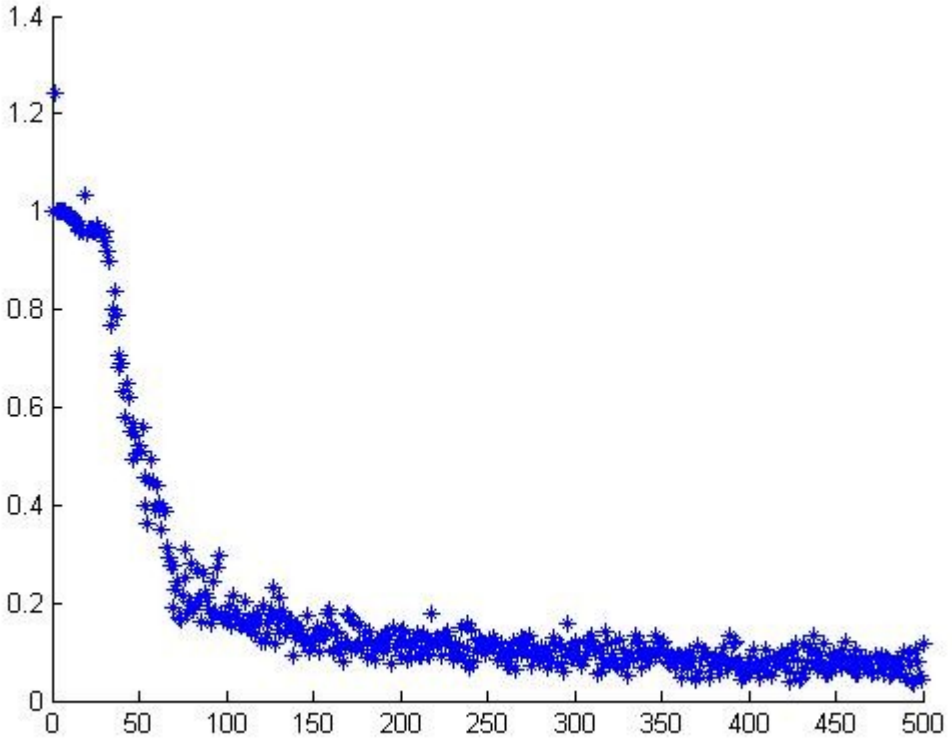
```
Wnn1=transpose(hidden_wts);
Wnn2=transpose(neural_out_wts);
bnn1=zeros(1,hidden_layer);
bnn2=zeros(1,k);
h='sigmoid';
```

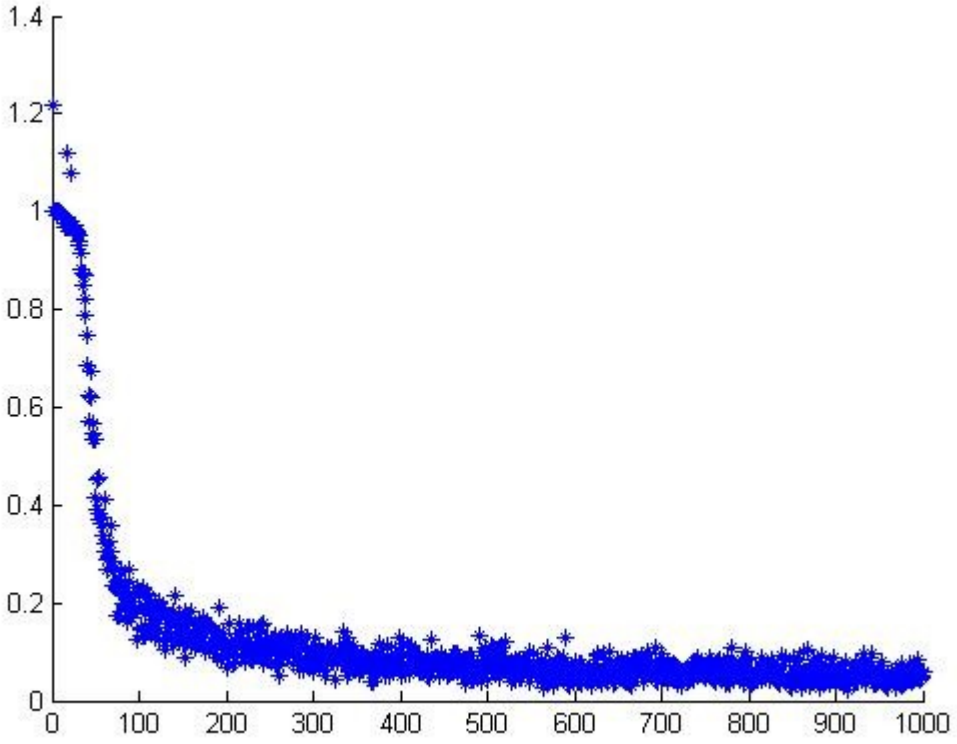
For me the tuned hyper-parameters values are:

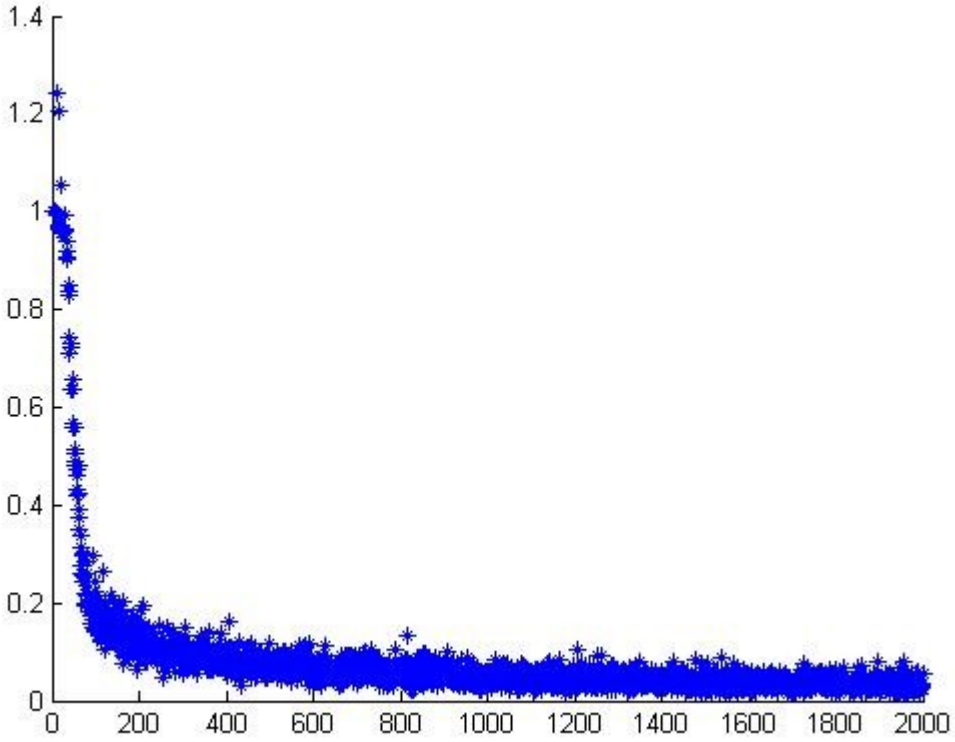
Batch size: 100
Iterations: 5 // we can increase the iterations but it would be very expensive computationally
Hidden Layer units: 500
Learning Rate: 0.5

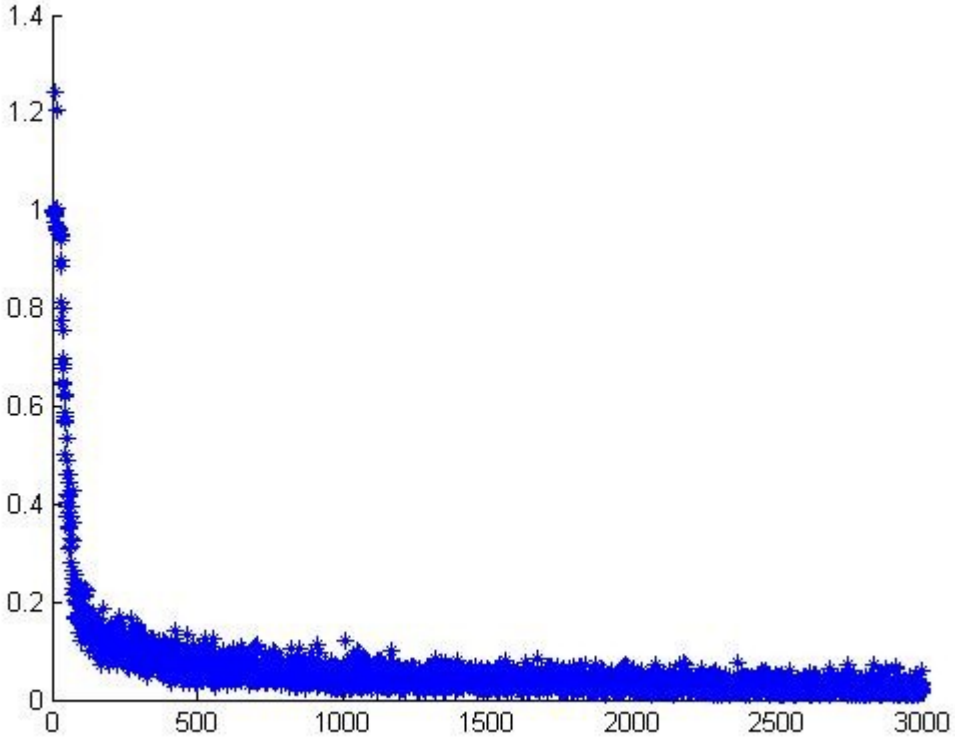


Figures while tuning:






1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
200
400
600
800
1000
1200
1400
1600
1800
2000


1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
500
1000
1500
2000
2500
3000

Convoluted Neural Network:

I used the deeplearning toolbox to generate the convoluted neural network. CNN with 3 hidden layers and 18 neurons for the dataset. The CNN gives an accuracy of 98.92% on the test set. It does back-propagation over the training datapoints and uses the sigmoid activation function.

We apply filter/kernel to an image which is convolved with the image. The operation performed between the filter mask and a single patch of the image, is the same as an MLP neuron–each of the corresponding components are multiplied together, then summed up. It's the same as taking the dot product between the image patch and the filter mask as vectors.

Filter/kernel correspond to weights of the layer. Weight of a particular neuron can be seen as a feature. The result of a convolution between a single filter mask and an image is referred to as a map. Each layer of a CNN will output multiple maps, one for each of the neurons. Each layer of the CNN applies its filters to the maps output by the previous layer.

Layer 1: Input is 28 by 28 pixel grayscale image.

Layer 2: Convolving a 5×5 filter with a 28×28 pixel image yields a 24×24 filtered image. This layer has 6 neurons, these six convolutions will generate 6 separate output maps, giving 24x24x6 matrix as the output of layer 2.

Layer 3: Pooling Layer, The pooling operation is averaging over 2×2 pixels. This has the effect of subsampling the output maps by a factor of 2 in both dimensions, we get a 12x12x6 matrix.

Layer 4: Convolution layer, kernel size of 5×5 and 12 neurons. For each of the 12 Layer 4 filters, there are 6 separate 5x5x1 kernels. To apply a single Layer 4 filter, we a perform 6 convolutions (one for each output map in Layer 3), and then sum up all of the resulting maps to make a single 8x8x1 output map for that filter. This is done for each of the 12 filters to create the 8x8x12 output of the Layer.

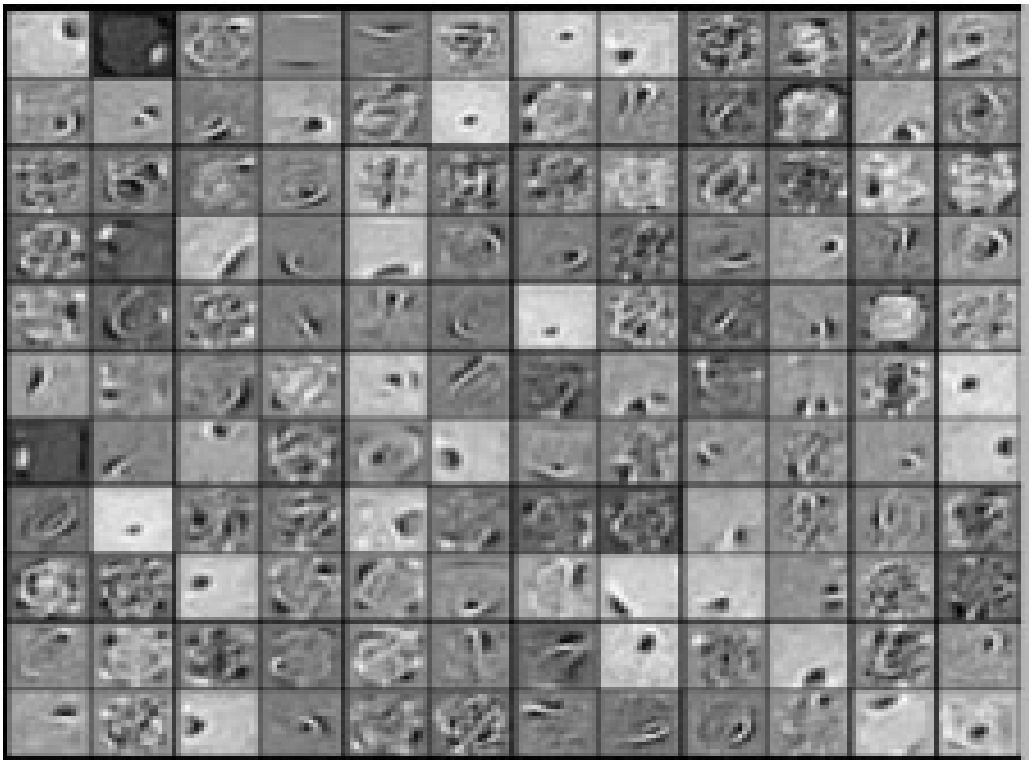
Layer 5: Perform pooling operation that's identical to the one in Layer 3. The resulting output maps are unwound into our final feature vector containing 192 values (4x4x12 = 192).

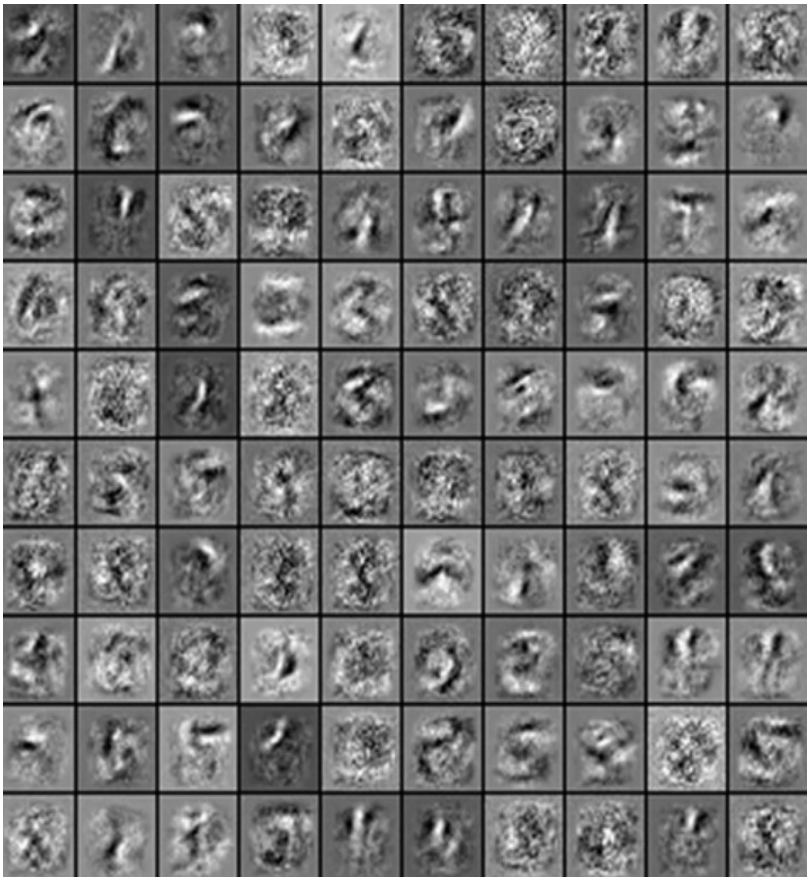
I just understood and ran the toolkit with the default epoch value of 100 and other hyper-parameters. Used the example code provided in the toolbox and other files to run the convoluted neural network:

```
load mnist_uint8;

train_x = double(reshape(train_x',28,28,60000))/255;
test_x = double(reshape(test_x',28,28,10000))/255;
train_y = double(train_y');
test_y = double(test_y');

rand('state',0)
```

```
cnn.layers = {
        struct('type', 'i') %input layer
        struct('type', 'c', 'outputmaps', 6, 'kernelsize', 5) %convolution layer
        struct('type', 's', 'scale', 2) %sub sampling layer
        struct('type', 'c', 'outputmaps', 12, 'kernelsize', 5) %convolution layer
        struct('type', 's', 'scale', 2) %subsampling layer
};
cnn = cnnsetup(cnn, train_x, train_y);
opts.alpha = 1;
opts.batchsize = 50;
opts.numepochs = 100;

cnn = cnntrain(cnn, train_x, train_y, opts);
[er, bad] = cnntest(cnn, test_x, test_y);

figure; plot(cnn.rL);
```

Visualization:





Conclusion:

Logistic regression gives us an error rate of around 7% without overfitting the model too much. The learning rate is to be adapted so that minimum error can be achieved. Batch size makes a difference in getting a smaller error rate. After some iterations the error reaches the minima. This is the least of the computationally expensive model among them for classification.

Single Layer Neural Network gives an error rate of around 2% on the test set. Tuning the number of neurons in the hidden layer is the key here. It is computationally expensive. Each iteration over the whole dataset takes a lot of time. Increasing iterations to more that 20 can give us error rate of around 1.5% but it is very expensive computationally. The programs runs for a few hours. I had a constant learning rate of 0.5 and batch size of 100 for this model.

Convoluted Neural Network gives the best performance and is easy to configure, but can be hard on computation if a GPU is not available. A single epoch/iteration on the data-points takes around 5 minutes without a GPU. This is the case when we have very limited number of layers and neurons in them. To build a complex CNN a GPU is a requirement. With epoch limit of 100, the error rate achieved was 1.08%. We can do even better than this if we use a large number of neurons in the hidden layers but would be very expensive computationally. With a GPU error rate of below 1% is achievable.

THE END