Backpropagation

Norwick Lee, Kathleen Guinee

Due: February 24, 2020

Abstract

Using backpropagation, we trained two neural networks, a simple one to recognize three custom numbers, and a more thoroughly designed one to recognize handwritten single digit numbers using data collected from the MNIST database.

1 Introduction

In the previous assignment, Hebbian Learning, we created and trained single layer neural networks. Now that we understand the simplified math of training neural network layers, we have graduated to training multiple layer neural networks.

In part one, we revisited the previous problem of recognizing low-resolution pixel numbers. However we used a three layer log sigmoid transfer function neural network in place of an autoassociative or pseudoinverse single layer network. The familiarity of the problem allowed us to practice implementing backpropogation with known constraints.

For part two, we expanded our network schema to recognize handwritten numbers. We used the image data from the MNIST database of handwritten digits (`http://yann.lecun.com/exdb/mnist/`). This included 60,0000 training images and 10,000 testing images. Each image was stored as a 28 x 28 vector of numbers.

This assignment gave us experience designing a neural network to process real world data, providing us with practical experience in how changing aspects of a neural network affects performance. It also demonstrated how neural networks can be used to accomplish image recognition.

2 Methods

To solve these problems, we used the back propagation algorithm as described in our textbook and as seen in Figure 1.

We used a three-layer model. The first layer was the input layer, the second, a hidden layer, and the last was the output layer. This can be seen in Figure 2.

For input data, we took vectors of length mn where n was the width and m was the height of a rectangular pixel image. Each entry in the vector was a number. For part one, the vector represented a 5x6 image, and each input was either -1 to indicate white space or 1 to indicate filled space. For part two, the vector represented a 28x28 image, and each input was a real number ranging between 0 to 1, with larger values corresponding to darker and lower values corresponding to lighter color in the pictures.

Our code followed the backpropogation algorithm and an example can be seen below.

Forward Propagation

$$\mathbf{a}^0 = \mathbf{p},$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1,$$

$$\mathbf{a} = \mathbf{a}^M.$$

Backward Propagation

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a}),$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \text{ for } m = M-1, \ldots, 2, 1,$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \ldots & 0 \\ 0 & \dot{f}^m(n_2^m) & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & & \dot{f}^m(n_{S^m}^m) \end{bmatrix},$$

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}.$$

Weight Update (Approximate Steepest Descent)

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T,$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m.$$

Figure 1: Back propagation algorithm as described in Neural Network Design. The steps are in order. First, you propagate forward, then you use the error to compute the sensitivities. You propagate backwards to find all of the sensitivities. You then update the weights and biases according to these sensitivities

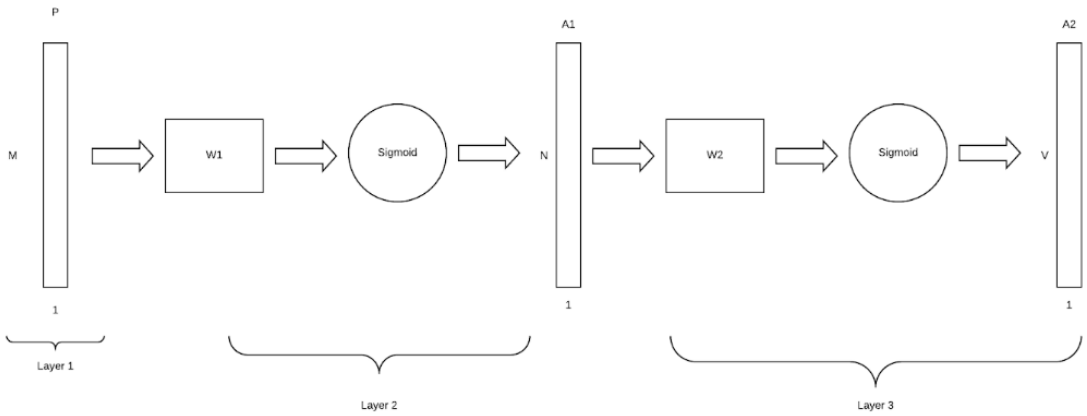

Figure 2: Three Layer Neural Network Layout. M is the input vector length. W1 and W2 are updated according to the sensitivities.

```
numNeuronInputsW1 = length(P(:,1)); %num of inputs for each input vector
numNeuronOutputsW1 = 20; %hidden layer number of neurons
numNueronOutputsW2 = length(T(:,1));

%randomize the starting weights to values close to zero
w1 = 2.*rand(numNeuronInputsW1,numNeuronOutputsW1).' - 1;
w2 = 2.*rand(numNeuronOutputsW1, numNueronOutputsW2).' - 1;
b1 = 2.*rand(numNeuronOutputsW1,1) - 1;
b2 = 2.*rand(numNueronOutputsW2,1) - 1;

input_num = length(P(1,:)); %number of input vectors
for i = 1:input_num %i is column num (or respective input vector)
    [n1, n2, out, a1] = propogateForward(P(:,i), w1, w2, b1, b2);
    s2 = computeOutputSensitivities(out, T(:,i), n2);
    s1 = backpropogateSensitivities(w2, s2, n1);
    [w1, b1] = updateWeightAndBiases(w1, b1, s1, P(:,i),r); %update first weight
    [w2, b2] = updateWeightAndBiases(w2, b2, s2, a1,r); %update second weight
end
```

For part 1, we initially set the number of neurons in the hidden layer to 6. By picking a relatively small number, this allowed us to focus on experimenting with the other aspects of architecture design such as the number of epochs. We tested the mean square error over 100 epochs, 1,000 epochs, and 10,000 epochs.

For part 2, we varied the number of neurons in the hidden layer in order to find the optimal number of neurons. Due to the sheer amount of training data, we opted to only iterate through 10 epochs and not check if increasing the epoch exponentially would improve training results. For the future project, we may research forcing matlab to use GPU in order to improve our ability to handle higher epoch iterations.

We additionally tested our network in part 2 against several different learning rates in order to identify the optimal rate. We initially used a wide scope of learning rates, and then we tested a smaller scope of learning rates around the best performing values in the previous test.

3 Results

We found that larger of epochs led to better performance in part 1. In part 2, we found that using learning rates below 0.1 and above 0.04 and less than 40 nodes in the hidden layer led to the best performance.

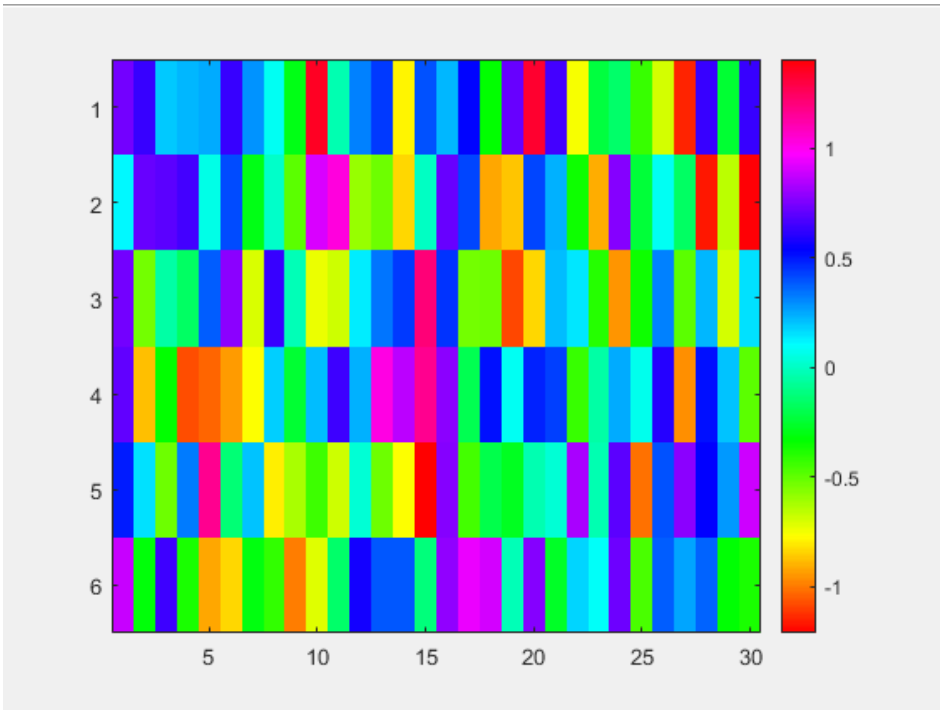

Figure 3: Hidden Layer Weights (W1) for Part 1

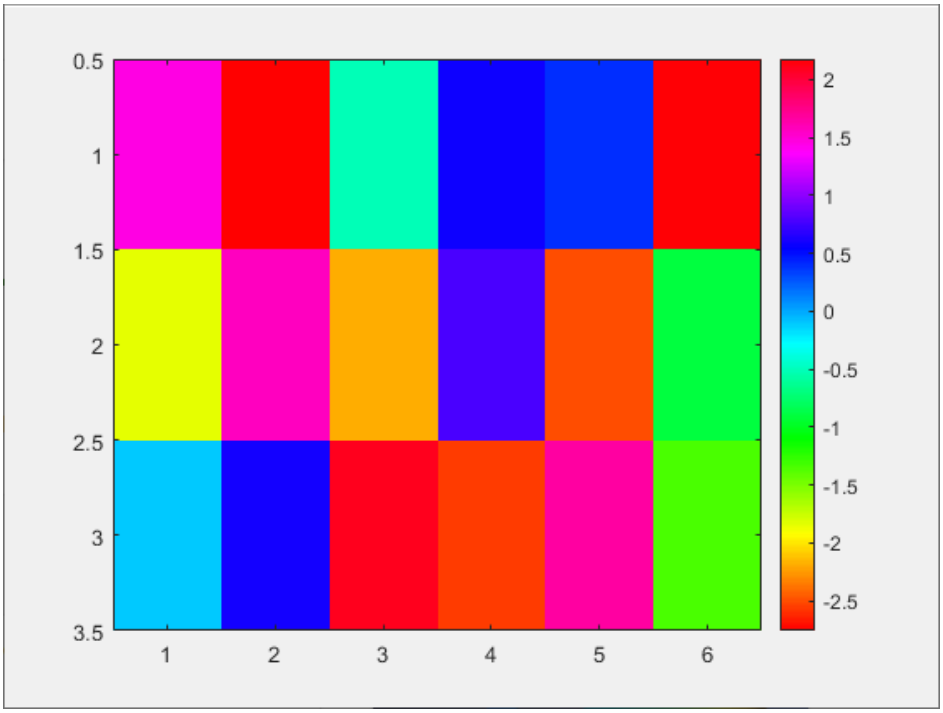

Figure 4: Output Layer Weights (W2) for Part 1

In the test described by Figure 5, we varied the number of epochs used in training the network. The results showed that increased epoch amount reduced the amount of error– especially for increased noise in data. For 8 noisy pixels, the mean squared error was almost half for 10,000 epochs compared to 100 epochs.

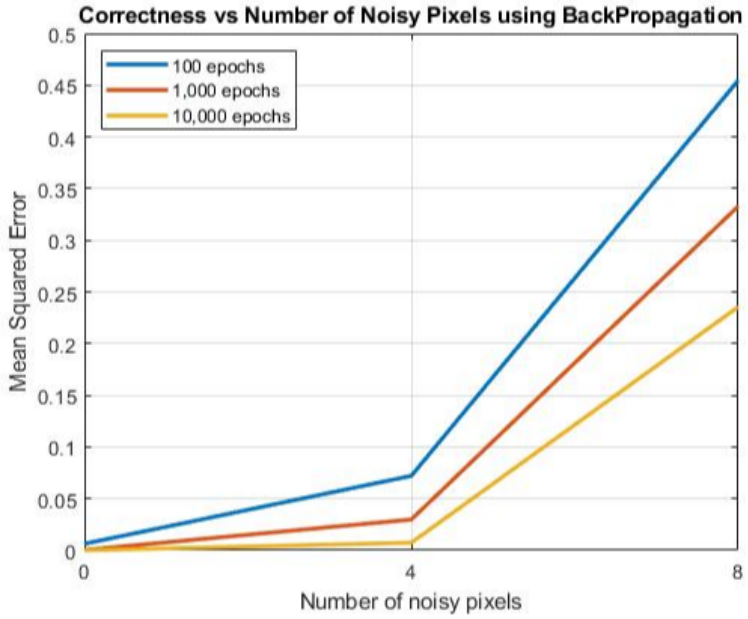

Figure 5: Mean Squared Error vs Number of Noisy Pixels for Part 1

We next experimented with the number of neurons in the hidden layer. We used the second training set (handwritten numbers identifier) to train the weights and then varied the number of neurons in the hidden layer. We chose numbers of neurons that were somewhere between the input vector length and the output vector length. The results, as shown in Figure 6, indicate that the optimal number of neurons in the hidden layer is close to 20. The higher number of neurons resulted in increasing amounts of error.

We began testing the learning rate to learn how it impacts the error by first casting a large net. We tested a range from 0.1 to 10 and found that the lowest error rate happened with a learning rate of 0.1 We then targeted the learning rate by focusing on much smaller values. The results from this indicate that an optimal learning rate occurs at around 0.08 .

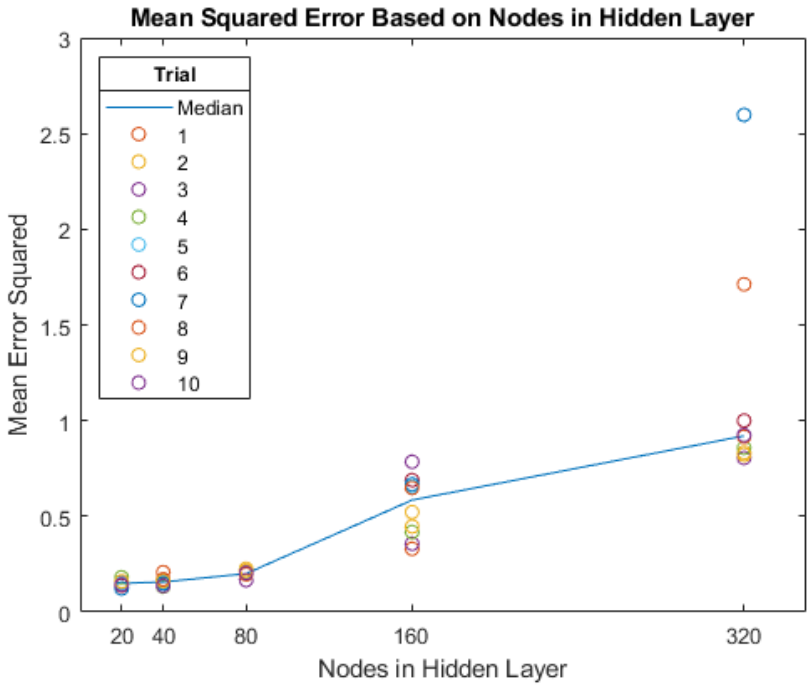

Figure 6: Correctness vs Number of Nodes in Hidden Layer for Part 2

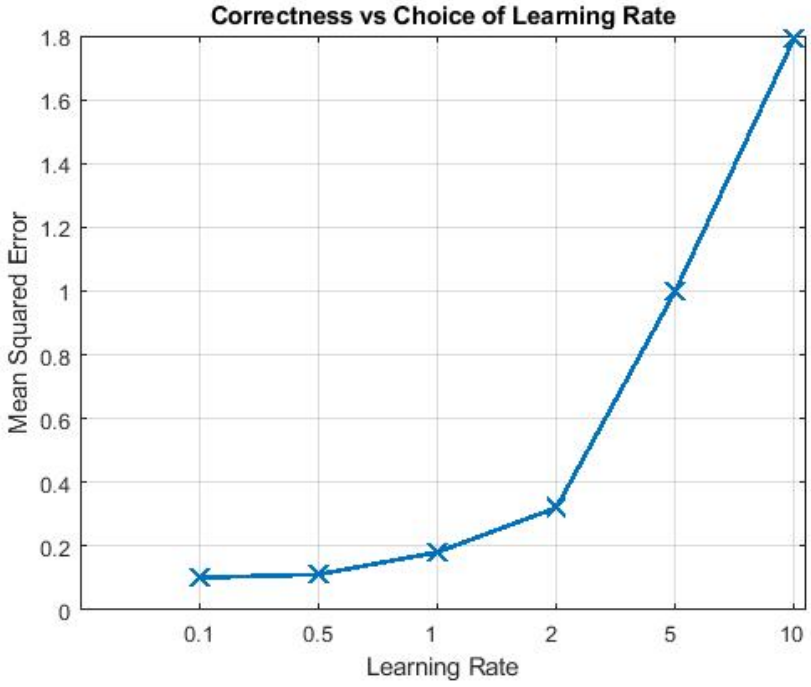

Figure 7: Wide: Mean squared error vs learning rate as tested for part 2

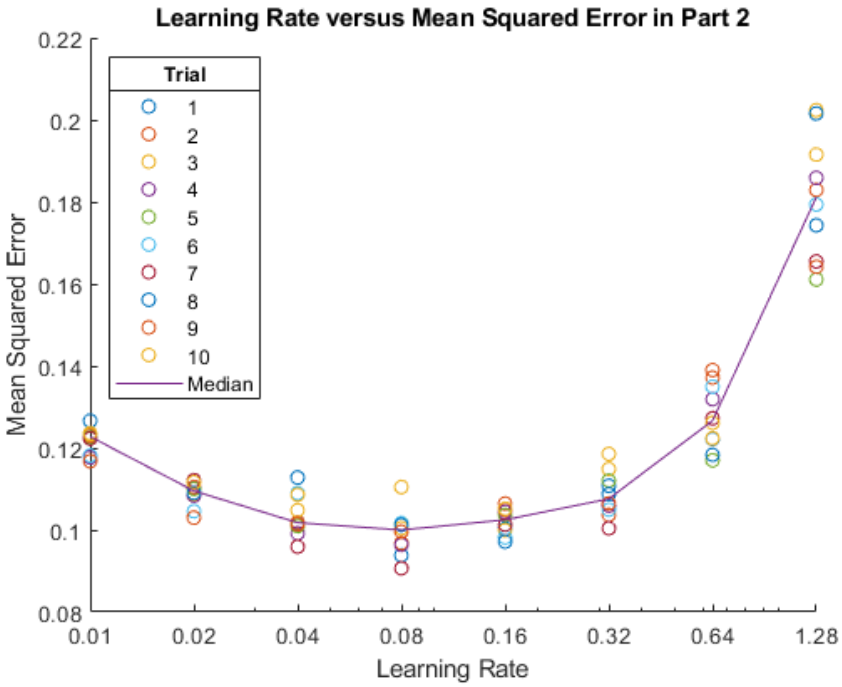

Figure 8: Targeted: Mean squared error vs learning rate as tested for part 2

4 Conclusion

4.1 Need for Processing Power

At 10,000 epochs, there were improvements in error reduction compared to 1,000. However, running the part 1 neural network for 10,000 epochs took several minutes to complete, forcing us to not probe further exponentially. With the part 2, we did not even try running multiple epochs due to having 60,000 samples in the training set. However, our trials for part 2 still took several hours to complete. In order to truly find optimum settings and global minimums for neural networks, we need to access to more computing power. In order to satisfactorily complete our final project, we will need to look into code optimization, using the linux lab, using GPU, or keeping a computer on reserve just for matlab.

4.2 Less Nodes Go Further

Despite having over 724 inputs in part 2, the lowest amount of nodes performed best. Due to the algorithm only needing to differentiate between 10 patterns, higher node layers may have simulated overly complicated functions leading to over-fitting.

4.3 Set Constant Learning Rates Low but Not Too Low

As discussed in class and mentioned in reading, small learning rates performed better than large learning rates. The larger learning rates likely continuously overshot the global optimum, wasting training iterations on backtracking and potentially caused a feedback loop to continuously worse weights. However, decreasing learning rate from the optimum found learning rate, 0.08, also resulted in continuously worse performance. Too small learning rates likely either became trapped in local optima too soon or failed to travel as close to the global optimum in the constant amount of iterations as larger learning rates did.

4.4 Data Formatting Concerns

MNIST formatted data through an algorithm that centered images around the high mass areas and cut away non centered pixels. Our final project will also utilize MNIST, meaning this concern will not apply. However, when creating neural networks designed to be responsive to real world inputs with less controlled parameters (for example, scale and image resolution), we will may need to spend significant time designing data pre-processing algoirthms.

5 Appendix

Code:

```
function [w1, w2, b1, b2, out] = backprop(epoch_num, P, T,r)

    numNeuronInputsW1 = length(P(:,1)); %num of inputs for each input vector
    numNeuronOutputsW1 = 20; %hidden layer neurons
    numNueronOutputsW2 = length(T(:,1));

    %randomize the starting weights to values close to zero
    w1 = 2.*rand(numNeuronInputsW1,numNeuronOutputsW1).' - 1;
    w2 = 2.*rand(numNeuronOutputsW1, numNueronOutputsW2).' - 1;
    b1 = 2.*rand(numNeuronOutputsW1,1) - 1;
    b2 = 2.*rand(numNueronOutputsW2,1) - 1;

    input_num = length(P(1,:)); %number of input vectors


    for j = 1:epoch_num
      for i = 1:input_num %i is column num (or respective input vector)
        [n1, n2, out, a1] = propogateForward(P(:,i), w1, w2, b1, b2);

        s2 = computeOutputSensitivities(out, T(:,i), n2);
        s1 = backpropogateSensitivities(w2, s2, n1);
        [w1, b1] = updateWeightAndBiases(w1, b1, s1, P(:,i),r);
        [w2, b2] = updateWeightAndBiases(w2, b2, s2, a1,r);
      end
    end


function s2 = computeOutputSensitivities(a, t, n2)
    % Sm = -2(t - a) * F'M(nm)
    %m is the layer number
    F = zeros(length(n2),length(n2));
    for i = 1:length(n2)
        F(i, i) = lsderiv(n2(i));
    end
    %F is 3x3, n is 3 x 1
s2 = -2 * F * (t - a);

function s_prev = backpropogateSensitivities(W, s, n)
%Sm = F'm* (nm) * (W2)' * S2'
F = zeros(length(n), length(n));
    for i = 1:length(n)
        F(i, i) = lsderiv(n(i));
```

```
    end
    s_prev = F * (W' *s);

function [W_m, b_m] = updateWeightAndBiases(W, b, s, a,r)
%Wm(k+1) = Wm(k) - r(sm* aM+1)'
%bm(k+1) = bm(k) - r(sm)
  W_m = W  - r*(s * a');
  b_m = b -r*s;

function lsd = lsderiv(n)
lsd = logsig(n)*(1 - logsig(n));
```