Deep Neural Network

Philip Hoddinott

December 17, 2018

Contents

List of Figures

Abstract

The purpose of this report is to develop a neural net that can identify handwritten digits in the MNIST database at near human levels of accuracy. The neural net will be developed without the assistance of libraries such as Python's tensor flow or MATLAB's Deep Learning.

The author would like to express his gratitude to Professor Hicken for the suggestion of this project. The author would also like to thank Theodore Ross and Varun Rao for their assistance with artificial neural networks.

1 Introduction

The first computational models for neural networks were thought up in the 1940s, however it would take 50 years for computers to achieve the processing power to implement the first neural networks. Today neural networks can be used for a variety of tasks. One of these tasks is image recognition of numbers.

1.1 The MNIST database

The Modified National Institute of Standards and Technology database or MNIST database [5] is a database of handwritten numbers used to train image processing systems. It contains 60,000 training images and 10,000 testing images. The database is comprised of images that are made up of a grid of 28x28 pixels. Some of these are seen in figure 1.

A number of attempts have been made to get the lowest possible error rate on this dataset. As of August 2018 the the lowest achieved so far is a error rate of 0.21% or an accuracy of 99.79%. For comparison human can accurately recognize digits at a rate of 98% - 98.5% [6].

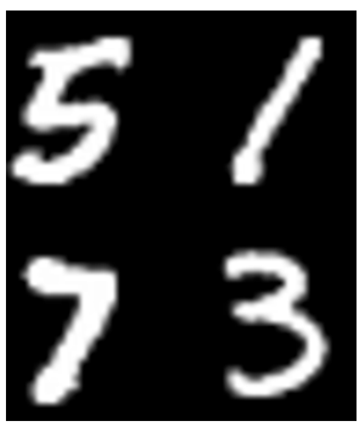


Figure 1: Sample numbers from MNIST [1].

1.2 Artificial neural network

An artificial neural network (referred to as a neural network in this paper) is a computation system that mimics the biological neural networks found in animal brains. A neural network is not an algorithm, but a general framework to solve problems. Artificial neural networks are based of layers of interconnected neurons that transmit signals to each other. Neural networks may be trained for tasks, such as the number recognition in this report.

The neural net implemented in this project had an input vector of 784×1 and an output vector of 10×1. Different configurations were tried, with one hidden layer of 250×1 producing the best results. A visualization of an example neural net is seen in figure 2.

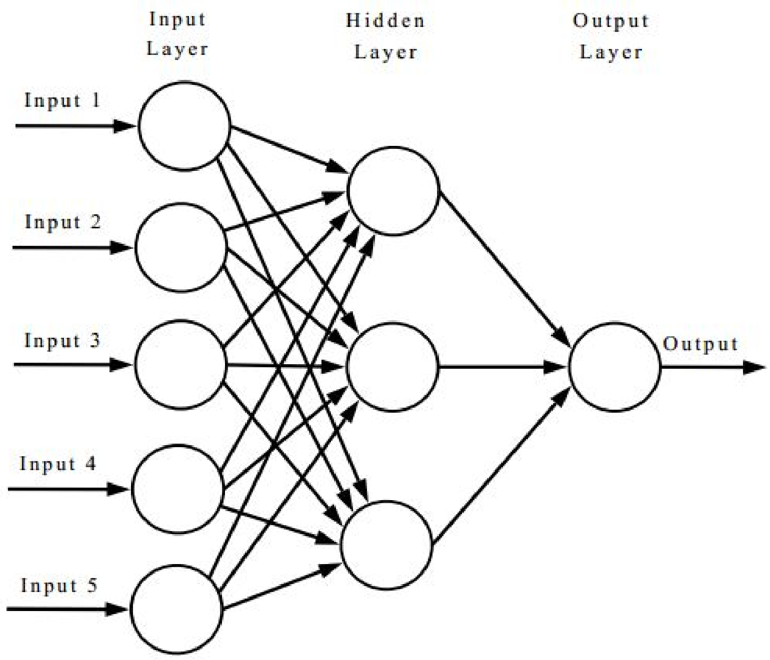


Figure 2: Visualization of a neural network with one hidden layer [2].

1.3 Neural Network Walkthrough

The training of a neural network involves four main steps:

1. Initialize weights and biases.
2. Forward propagation
3. Compute the loss
4. Backward propagation

1.3.1 Parameter Initialization

The first step in training a neural net is to initialized the bias vectors and weight matrices. They are initialized with random numbers between 0 and 1, then multiplied by a small scalar around the order of 10^{-2} so that the units are not on the region where the derivative of the activation function are close to zero. The initial parameters should be different values (to keep the gradients from being the same).

There are various forms of initialization such as Xavier initialization or He-et-al Initialization, but a discussion on methods of initialization outside the scope of this paper. In this paper we will stick with random parameter initialization.

1.3.2 Forward Propagation

The next step is the forward propagation. The network takes the inputs from a previous layer, computes their transformation, and applies an activation function. Mathematically the forward propagation at level "i" is represented by equation 1.

$$\begin{aligned} z_i &= A_{i-1} * W_i + b \\ A_i &= \phi(z_i) \end{aligned} \tag{1}$$

Where z is the input vector, A is the layer, W is the weights going into the layer, b the bias, and ϕ the activation function. This process then repeats for the next layer until it reaches the end of the neural net.

1.3.3 Compute loss

The loss is simply the difference between the output and the actual value. In this neural net it is computed by equation 2.

$$\text{loss} = A_{i=\text{end}} - y \tag{2}$$

This loss is used to begin the next step: backward propagation.

1.3.4 Backward propagation

After going forward through the neural net in the forward propagation step and computing the loss the final step is backwards propagation. Backwards propagation is the updating of the weight parameters via the derivative of the error function with respect to the weights of the neural net. For the output layer this is seen in equation 3 and equation 4 for all other layers.

$$dW_{i=\text{end}} = \phi'(z_{i=\text{end}}) * (A_{i=\text{end}} - y) \tag{3}$$

$$dW_i = \phi'(z_i) * \left(W^T_{(i+1)} * dW_{(i+1)}\right) \tag{4}$$

Once these derivatives have been computed, the weights are updated by equation 5

$$W_i = W_i = \alpha * dW_i * A^T_{(i-1)} \tag{5}$$

Where for the first layer $z^T_{(i-1)}$ will be the input vector and for all the following layers it will be the vector from the previous layer.

At this point the neural net has completed a full run through. The next input vector is selected and the forward and backward propagation are run again. A visualization of forward and backward propagation is in figure 3.

1.4 Gradient Decent

Also known as steepest decent, gradient decent is a first order optimization algorithm. It is used to find the minimum of a function. Equation 6 shows gradient decent implemented in a neural net.

$$\Delta W(t) = -\alpha \frac{\partial E}{\partial W(t)} \tag{6}$$

Where α is the learning rate, and $\partial E/\partial W(t)$ is the error derivative with respect to the weight. As these derivatives must be computed for each node the more nodes there are in a neural net the longer it will take to train.

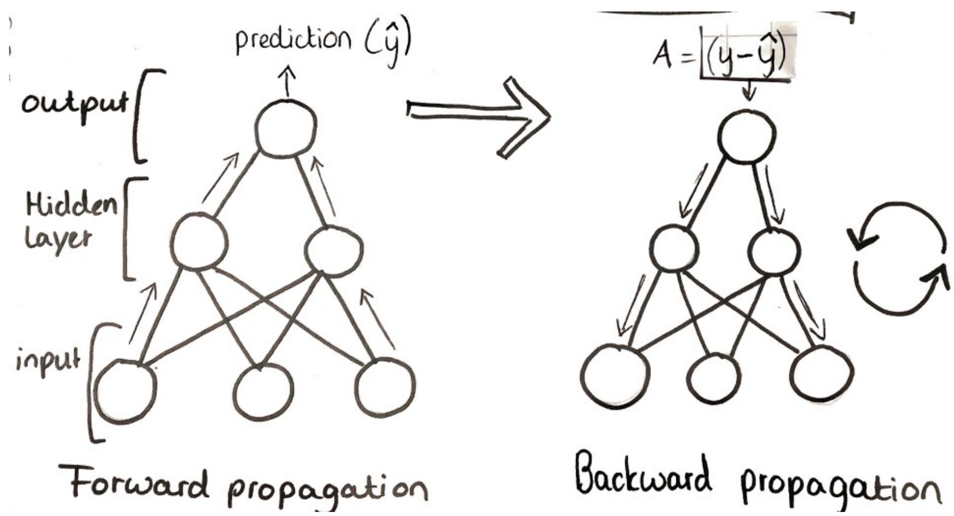


Figure 3: A visualization of forward and backward propagation [3].

1.5 Activation Function

The activation function was previously mentioned as a function used to convert the input signal to the output signal. Activation functions introduce non-linear properties to the neural net's functions, allowing the neural net to represent complex functions [3].

The two most common activation functions used in neural nets for the gradient decent are sigmoid and hyperbolic tangent (Tanh). The formula for Tanh is seen in equation 7, and the formula for it's derivative is seen in equation 8

$$\phi_{\text{Tanh}}(z) = \frac{1 - e^{-2z}}{1 + e^{-2z}} \tag{7}$$

$$\phi'_{\text{Tanh}}(z) = \frac{4}{(e^{-z} + e^{z})^2} \tag{8}$$

The formula for the sigmoid function is seen in equation 9, the formula for it's derivative is seen in equation 10.

$$\phi_{\text{Sigmoid}}(z) = \frac{1}{1 + e^{-z}} \tag{9}$$

$$\phi'_{\text{Sigmoid}}(z) = \frac{e^{-z}}{(e^{-z} + 1)^2} \tag{10}$$

The sigmoid and Tanh function are visualized in figure 4.

Both functions have relatively simple mathematical formulas and are differentiable. In this paper the sigmoid function is used over the Tanh function as it had better results. Sigmoid and Tanh are not the only activation functions. Other functions that should be noted are the Rectified Linear Unit (ReLU) and the Leaky Rectified Linear Unit function. These functions have their own separate pros and cons, but a proper discussion of them is outside the scope of this paper.

1.6 Pitfalls

The most important thing to steer clear of is over training. Over training occurs when the neural net trains too much to the training data. While it will have a high accuracy for the training data, it's performance for the test data will decay, as it has become too well attuned to the training data.

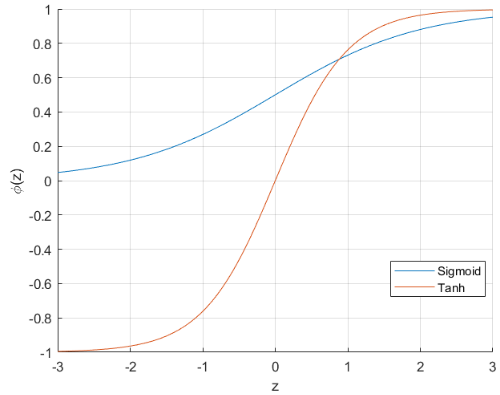


Figure 4: Visualization of sigmoid and Tanh function

ID	architecture (number of neurons in each layer)	test error for best validation [%]	best test error [%]	simulation time [h]	weights [milions]
1	1000, 500, 10	**0.49**	0.44	23.4	1.34
2	1500, 1000, 500, 10	**0.46**	0.40	44.2	3.26
3	2000, 1500, 1000, 500, 10	**0.41**	0.39	66.7	6.69
4	2500, 2000, 1500, 1000, 500, 10	**0.35**	0.32	114.5	12.11

Figure 5: Run times for various neural network architectures [4].

The other problem is the time it takes to train. A three layer neural net can be trained to 97% accuracy within 10 minutes, however it will not improve far beyond that. Larger nets will take longer to train, but will take far longer to train.

2 Implementation

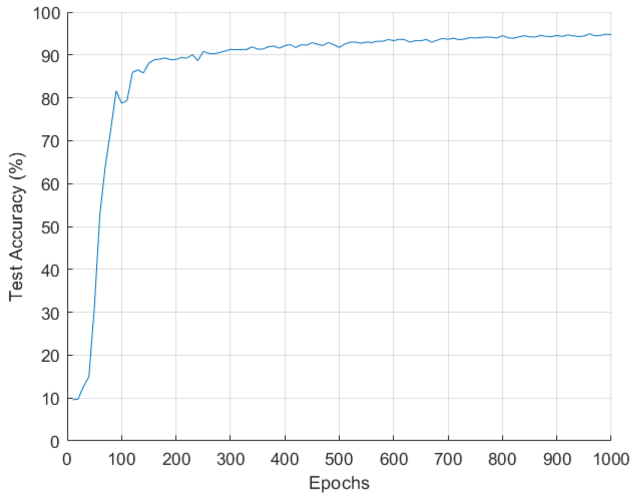
2.1 Object Oriented Programming in MATLAB

This neural net had to be made without the use of any built in libraries [7] and the code had to be modular [8]. To create code for neural network subject to these constraints the author decided to create their own neural net class in MATLAB.

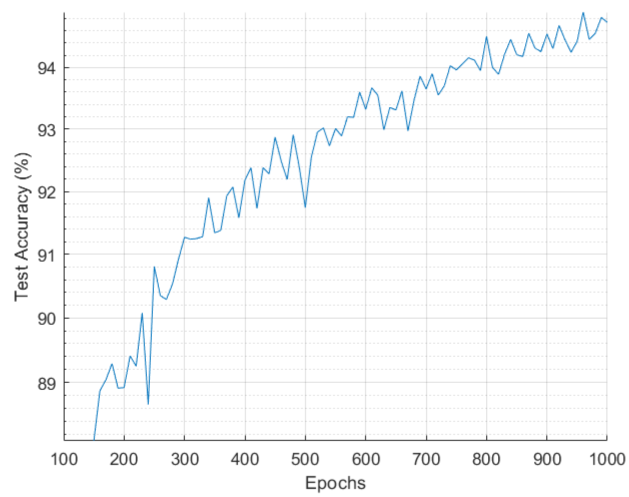
The MATLAB class philipNeuralNet.m was written for this neural net project. It has the parameters learningRate, enableBias, actFunSwitch, and Level. The learningRate parameter is obviously the learning rate. The Level parameter has four parameters attached to it: W (weight), dW (weight derivative), z (input), and A (the vector for the layer). By having this class we have avoided hard coding the propagation of the neural net and it is possible to test different neural net architectures on this code.

The MATLAB class also has an activation function and a derivative of the activation function. The actFunSwitch variable allows either the Sigmoid or the Tanh function to be selected. Additionally the enableBias variable allows for biases to be used or not used in the code's execution.

Finally it has a outputVector function that is simply an implementation of the neural net. It takes the input, runs it through the net, and returns the net's output.



(a) The results for the first 1000 epochs.



(b) The results for the first epochs scaled logarithmically

Figure 6: The results from the simple network for the first 1000 epochs.

2.2 MATLAB Code

The MALAB code first initializes a neural net from given parameters. It obtains the MNIST data from a function [9]. It then uses the handleTrainNet function to train the net. This function implements batch training, using the forward and backward propagation functions. It then computes and displays the training error and the testing accuracy after a specified number of runs via the testAcc.m function. Once it has done this it plots the accuracy of the neural net via the plot accuracy function, generating the plots seen in the results section.

3 Results

3.1 Simple Neural Net

The best results were found for the simplest neural net examined; a one hidden layer with 250 nodes a learning rate of 0.1, and no biases. For this simple a test accuracy of 98% was achieved. The first 1000 epochs of this net are visualized in figure 6. To get the 98.37% accuracy it took approximately 12 hours of running the code and over three million neural net evaluations.

3.2 Comparison of different hidden layer sizes

From Shure [10], the optimal size for the hidden layer in a three layer neural network for the MNIST is 250 nodes. Comparing the results for hidden layers show in figure 7, different sizes of hidden layer do not have a large effect on the accuracy of these results. However what is different is the time it takes to run each net. The more nodes in a net, the longer it takes for the net to train, as there are more operations to perform. Thus if a neural net with 250 nodes will have the same accuracy as a net with 800 nodes, the first net is preferable, as it will be trained faster.

3.3 Multiple hidden layers

The best accuracy occurred when both layers had a size of 250×1. For network with two hidden layers of 250 each the best test accuracy was 96.49%. The accuracy from the first 2000 epochs is seen in figure

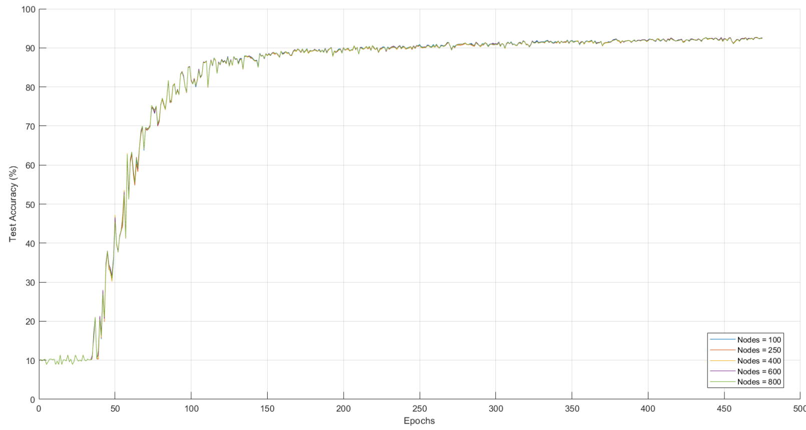
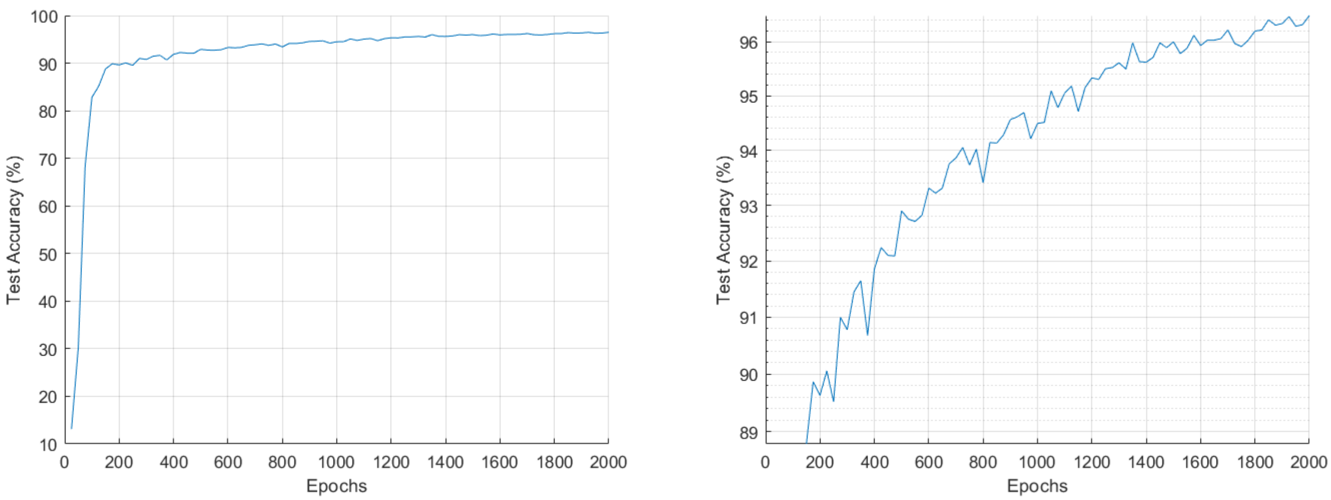


Figure 7: Net accuracy for different layer sizes.



(a) The results for the first 2000 epochs.

(b) The results for the first epochs scaled logarithmically

Figure 8: The results from the two layer network for the first 2000 epochs.

8.

The accuracy of two hidden layers was not as good as the accuracy of one hidden layer. No other configurations achieved an accuracy as high as this one.

4 Conclusion

The simple neural net achieved an accuracy of 98.37%. This is on par with human recognition, and is about as accurate as a simple neural net can achieve. Higher accuracy rates are achieved via the use of constitutional neural networks, such as the LeNet-5 [**?**]. Due to the long run time of large multi layered neural networks they were not studied, but could provide a more accurate identification with out convolution.

What is interesting is that the best results occurred with out biases and with one hidden layer. It was expected that adding more complexion to the neural net would increase the accuracy, however this was not the case. As neural nets are very much a trial and error process, it is possible that these more complex nets will achieve a better accuracy with more fiddling.

References

[1] Deep learning for classification on the mnist dataset. https://www.mathworks.com/examples/computer-vision/community/36153-deep-learning-for-classification-on-the-mnist-dataset, 2018.

[2] Diagram of an artificial neural network. https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network, September 2013.

[3] Daphne Cornelisse. https://medium.freecodecamp.org/building-a-3-layer-neural-network-from-s fbclid=IwAR1jjh1IsEVdvN0pIeMygzfY2ZG3K-Zata3z_jqlfLtQZKK0X6QEbNZABzw, February 2018.

[4] Luca Maria Gambardella Jurgen Schmidhuber Dan Claudiu Ciresan, Ueli Meier. Deep big simple neural nets excel on handwritten digit recognition. https://arxiv.org/pdf/1003.0358.pdf, March 2010.

[5] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[6] John Denker Patrice Simard, Yann Le Cun. Efficient pattern recognition using a new transformation distance. https://papers.nips.cc/paper/656-efficient-pattern-recognition-using-a-new-transformation-distance.pdf.

[7] Jason Hicken. Mane 6963 independent study description, December 2018.

[8] Jason Hicken. Rubric for mane 6963 independent study, December 2018.

[9] Using the mnist dataset. http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset, may 2011.

[10] Loren Shure. Artificial neural networks for beginners. https://blogs.mathworks.com/loren/2015/08/04/artificial-neural-networks-for-beginners/, August 2015.

Appendix 1 - MATLAB code

NN_Master.m

```
%% Philip Hoddinott NN
% Neural Net for MNIST numbers
%% Setup
clear all; close all;
%% load values
% These functions come from
% http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset
inputValues = loadMNISTImages('train-images.idx3-ubyte');
labels = loadMNISTLabels('train-labels.idx1-ubyte');

inputValuesTest = loadMNISTImages('t10k-images.idx3-ubyte');
labelsTest = loadMNISTLabels('t10k-labels.idx1-ubyte');

% change labels
targetValues = 0.*ones(10, size(labels, 1));
```

```
for n = 1: size(labels, 1)
    targetValues(labels(n) + 1, n) = 1;
end
% traing paramters
sizeArr=[250;10];
learningRate=.1;
batchSize = 100;
epochs = 500; numEpoch=1;
% net switches
enableBias=0; %  0 for off, 1 for on
actFunSwitch =0;  % 0 for sigmoid, 1 for Tanh
% create net
pNet=philipNeuralNet(inputValues,sizeArr,learningRate,enableBias,actFunSwitch);
% train net
[pNet,pNetBest,errorM,testAccM]=handleTrainNet(pNet, inputValues, targetValues, ...
    epochs, batchSize, inputValuesTest, labelsTest,numEpoch,sizeArr);
% plot results
plotAcc

function [pNet,pNetBest,errorM,testAccM] = handleTrainNet(pNet,  inputValues, ...
    targetValues, epochs, batchSize, inputValuesTest, labelsTest,numEpoch,sizeArr)
    % handleTrainNet function to train net via batch traning
    % Input
    % pNet : net
    % epochs : number of epochs
    % numEpoch : epoch multiplier
    % batchSize : size of batch
    % inputVector : MNIST Input vector for training
    % targetVector : MNIST Labels for traning validation
    % sizeArr : net architecture
    % inputValuesTest : MNIST Input vector for testing
    % labelsTest : MNIST Labels for testing validation
    %
    % Output
    % pNet : net
    % pNetBest : pNet with best accuracy
    % errorM : matrix of traning error
    % testAccM : matrix of test accuracy
    trainingSetSize = size(inputValues, 2); % get traning set size
    errorM=[]; testAccM=[]; % init values

    n = zeros(batchSize); % init values
    errorBest=100;  accBest=-1;  % init values

    figure; hold on; % init figure
    ylabel('Training Error');     xlabel('Epochs');
    tic
    for t = 1: numEpoch*epochs
        for k = 1: batchSize
            % Select input vector to train on.
            n(k) = floor(rand(1)*trainingSetSize + 1);
            % get inputs and targets
            inputVector = inputValues(:, n(k));
            targetVector = targetValues(:, n(k));
            % forward propogation
            pNet = forwardProp(pNet,sizeArr,inputVector);
            % backwards Propagation
            pNet=backprop(pNet,sizeArr,inputVector,targetVector);
        end % end foor loop
```

```

        % Calculate the error for plotting.
        error = 0;
        for k = 1: batchSize
            inputVector = inputValues(:, n(k));
            targetVector = targetValues(:, n(k));
            outputVector = pNet.netOutput(inputVector,sizeArr);
            error=error+norm(outputVector- targetVector, 2);

        end
        error = error/batchSize; errorM=[errorM,error];
        plot(t,error,'*')

        if error<errorBest
            pNetBest=pNet; errorBest=error; % get best error
        end

        if mod(t,25)==0 %
            [numCorrect, numErrors] = testAcc(pNet, inputValuesTest, ...
               labelsTest,sizeArr);
            acc=100*(numCorrect) / (numCorrect+numErrors); % get accuracy
            if acc>accBest
                accBest=acc; % get best accuracy
            end
            testAccM=[testAccM,acc];
            fprintf('Epoch = %d,error = %.4f, best acc = %.4f\n',t, error,accBest)
            grid on;
            toc % time to run
        end
        drawnow % draw error
    end

end

function pNet = forwardProp(pNet,sizeArr,inputVector)
    % forwardProp function to perform forward propgation
    %
    % Input
    % pNet : net
    % inputVector : MNIST Input vector for training
    % targetVector : MNIST Labels for traning validation
    % sizeArr : net architecture
    %
    % Output
    % pNet : net
    for i=1:length(sizeArr)
        if i==1 % 1st layer from input
            pNet.Level(i).z=pNet.Level(i).W*inputVector;
        else % all other layers
            pNet.Level(i).z=pNet.Level(i).W* pNet.Level(i-1).A;
        end
        pNet.Level(i).A=pNet.actFunc(pNet.Level(i).z+pNet.Level(i).b); % handle bias
    end
end

function pNet=backprop(pNet,sizeArr,inputVector,targetVector)
    % backprop function to perform backpropgation
    %
    % Input
```

```
    % pNet : net
    % inputVector : MNIST Input vector for training
    % targetVector : MNIST Labels for traning validation
    % sizeArr : net architecture
    %
    % Output
    % pNet : net
    learningRate=pNet.learningRate; % get leraning rate
    iArr=linspace(length(sizeArr),1,length(sizeArr)); % array to go backwards
    for i=iArr
        if i==length(sizeArr) % derivative from cost at output
            pNet.Level(i).dW=pNet.dactFunc( pNet.Level(i).z).* ...
                (pNet.Level(i).A-targetVector);
        else % derivative for other hidden layers
            pNet.Level(i).dW = pNet.dactFunc( ...
                pNet.Level(i).z).*(pNet.Level(i+1).W'* pNet.Level(i+1).dW);
        end
        dz=pNet.dactFunc(pNet.Level(i).z); % vector deriv
        pNet.Level(i).db=(1/length(dz))* sum(dz,2); % bias deriv
    end

    for i=iArr
        if i≠1 % adjust weight for all hidden layers not at input
            pNet.Level(i).W= ...
                pNet.Level(i).W-learningRate*pNet.Level(i).dW*pNet.Level(i-1).A';
        else % adjust hidden layer weight at input
            pNet.Level(i).W= ...
                pNet.Level(i).W-learningRate*pNet.Level(i).dW*inputVector';
        end
        if pNet.enableBias==1 % switch for enable bias
            pNet.Level(i).b=pNet.Level(i).b-learningRate*pNet.Level(i).db;
        end
    end
end
```

philipNeuralNet.m

```
classdef philipNeuralNet
    %philipNeuralNet Class created for neural net
    %   Detailed explanation goes here

    properties
        learningRate; % learning rate
        Level; % level, which has z, W, dW, b, db, and A
        enableBias; % switch to toggle bias on /off
        actFunSwitch; % switch for Tanh or Sigmoid
    end

    methods
        function obj = ...
            philipNeuralNet(inputValues,sizeArr,learningRate,enableBias,actFunSwitch)
            % philipNeuralNet function to initzlied neural net
            % Input
            % inputValues : input values for net
            % sizeArr : net architecture
            % learningRate : net learningRate
```

```
            % enableBias : switch to toggle bias on /off
            % actFunSwitch : switch for Tanh or Sigmoid
            %
            % Output
            % obj : pNet
            inputDim = size(inputValues, 1); % get dim

            for i =1:length(sizeArr)
                if i==1
                    obj.Level(i).W=rand(sizeArr(i),inputDim);
                    obj.Level(i).W=( obj.Level(i).W)./size( obj.Level(i).W,2);
                else
                    obj.Level(i).W=rand(sizeArr(i),sizeArr(i-1));
                    obj.Level(i).W=( obj.Level(i).W)./size( obj.Level(i).W,2);
                end
                obj.Level(i).z=learningRate*rand(sizeArr(i),1);
                obj.Level(i).A=learningRate*rand(sizeArr(i),1);
                obj.Level(i).dW=learningRate*rand(sizeArr(i),1);
                obj.Level(i).b=learningRate*zeros(sizeArr(i),1);
                obj.Level(i).db=learningRate*obj.Level(i).b;

            end
            obj.learningRate=learningRate;
            obj.enableBias=enableBias;
            obj.actFunSwitch=actFunSwitch;
        end
         function funcVal = actFunc(obj,x)
            %actFunc activation function depending on the activation
            %funciton switch, this performs sigmoid or TanH
            % Input
            % x : vector to perfom function on
            % obj : pNet
            %
            % Output
            % funcVal: result of function(z)
            if obj.actFunSwitch==0 % for sigmoid
                funcVal = 1./(1 + exp(-x));
            elseif obj.actFunSwitch==1 % for tanh
                funcVal=tanh(x);
            end
         end

        function funcD = dactFunc(obj,x)
            %dactFunc activation func derivative
            %   depending on the activation funciton switch, this performs
            %   derivative of sigmoid or Tanh
            %
            % Input
            % x : vector to perfom function on
            % obj : pNet
            %
            % Output
            % funcD: result of function(z)
            if obj.actFunSwitch==0
                funcD = obj.actFunc(x).*(1 - obj.actFunc(x));
            elseif obj.actFunSwitch==1
                funcD=(1-tanh(x).^2);
            end
        end
```

```

        function outputVector = netOutput(pNet, inputVector,sizeArr)
            % netOutput get the output of the neural net given an input and
            %
            % INPUT:
            % pNet : net
            % inputVector : input to net
            % sizeArr : net architecture
            %
            % OUTPUT:
            % outputVector : output of net
            for i=1:length(sizeArr)
                if i==1
                    pNet.Level(i).z=pNet.Level(i).W*inputVector;
                else
                    pNet.Level(i).z=pNet.Level(i).W*pNet.Level(i-1).A;
                end
                pNet.Level(i).A=pNet.actFunc(pNet.Level(i).z);
            end
            outputVector=pNet.Level(i).A;
        end

    end
end
```

testAcc.m

```
function [numCorrect, numErrors] = testAcc(pNet, inputValues, labels, sizeArr)
    % testAcc test the accuracy of a net using mnist validation set
    %
    % INPUT:
    % pNet : net
    % inputValues : MNIST Input values for training
    % labels : MNIST Labels for validation
    % sizeArr : net architecture
    %
    % OUTPUT:
    % numCorrect : number of correctly classified numbers.
    % numErrors : number of classification errors.
    %
    testSetSize = size(inputValues, 2);
    numErrors = 0;   numCorrect = 0;

    for n = 1: testSetSize
        inputVector = inputValues(:, n);
        outputVector = pNet.netOutput(inputVector,sizeArr);
        max = 0; class = 1;

        for i = 1: size(outputVector, 1)
            if outputVector(i) > max
                max = outputVector(i);
                class = i;
            end
        end

        if class == labels(n) + 1
```

```
            numCorrect = numCorrect + 1;
        else
            numErrors = numErrors + 1;
        end
    end
end
```