# MNIST Handwriting Recognition

**Chongyi Xu**
**University of Washington**
**AMATH 482/582 Winter Quarter 2018**
chongyix@uw.edu

**Abstract**

For this project, we will use MNIST handwritten digit dataset to build neural networks, which might help us as a detector of digits. MNIST database has 60000 images of handwriting digits with labels as training data and another 10000 images also with labels as testing data. A single layer and a 2 layer will be programmed to classify each digit.

## 1. Introduction and Overview

Neural networks (NNs)are computing systems vaguely inspired by the biological neural networks that constitute animal brains. Such systems "learn" (i.e. progressively improve performance on) tasks by considering examples, generally without task-specific programming. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any a priori knowledge about cats, e.g., that they have fur, tails, whiskers and cat-like faces. Instead, they evolve their own set of relevant characteristics from the learning material that they process.

## 2. Theoretical Background

Consider we have an input spike $\mathbf{x}$, we would like to know if input object is either a dog or a cat. Neural network says that, with a series of mapping function $f$, we would find a mapped perceptron $\mathbf{y}$ such that we could make our decision based on the output value of the perceptron, for instance, $1 = dog$ and $-1 = cat$. During the training process, we would like to get such $f$ for our network, from the data matrix. For example, we have 5 dogs(A, B, C, D, E) and 5 cats(a, b, c, d, e), the training process is as simple as telling the machine that A E are dogs and a e are cats.

$$[1\;1\;1\;1\;1\;\;-1\;\;-1\;\;-1\;\;-1\;\;-1] = f([A\;B\;C\;D\;E\;a\;b\;c\;d\;e])$$

And the $f$ is what we are looking for. $f$ will differ with different layers. In this project, I will focus on a single-layer network and a 10-layer networks as an example of multi-layer networks.

### 2.1. *One Layer Network*

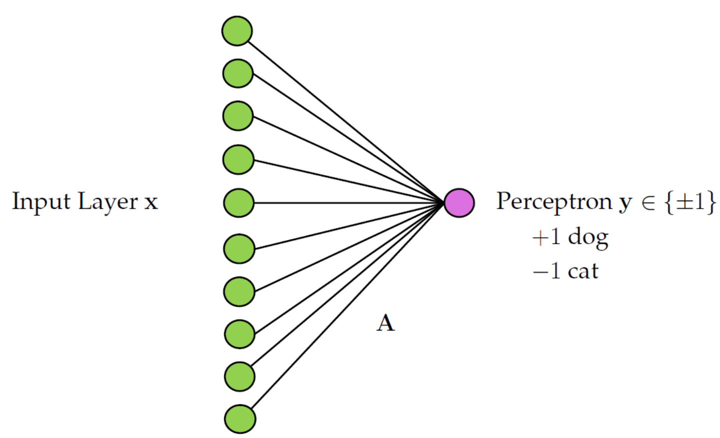


Figure 1: Single Layer Network

With one layer, as shown in the Figure 1, it will classify the object after one single linear map and give the decision based on the output of the perceptron.

$$\mathbf{Y} = A\mathbf{X}$$

And the layer $A$ could be simply calculated by

$$A = \mathbf{Y}\mathbf{X}^{+}$$

This gives a least-square regression for the mapping matrix $A$.

*2.2. Multiple Layer Networks*

As for some more complicated mapping models, like what we as humans always do, it is needed to have a higher-order layer for making decision. Figure 2 is an example of two-layer net architecture. It can be seen that

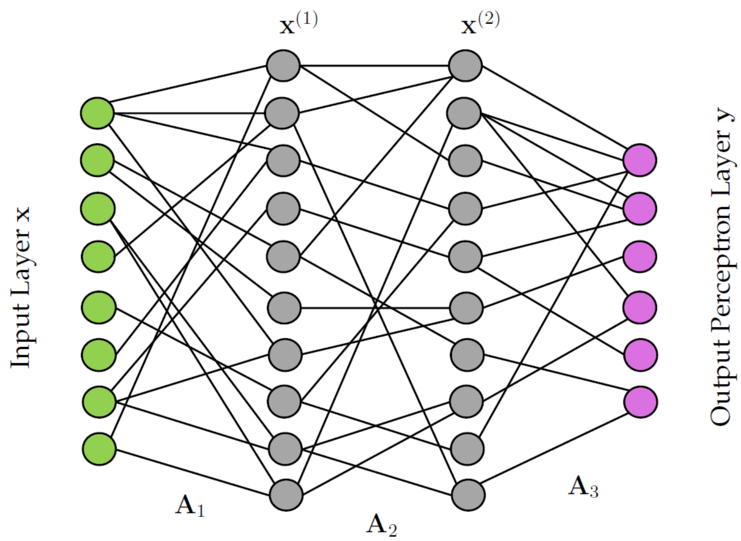


Figure 2: Two Layer Network

the mapping process now becomes much more complicated rather than a single layer. Figure 3 shows how mapping goes to chaos as the number of layers get increased. And constructing such network is also known as deep learning process since the machine now is trying to learn by itself how to make decision. And the training for N layer neural network to the data matrix X will be

$$\begin{aligned}
\mathbf{X^{(1)}} &= f_1(A_1, \mathbf{X}) \\
\mathbf{X^{(2)}} &= f_2(A_2, \mathbf{X^{(1)}}) \\
&\vdots \\
\mathbf{X^{(N-1)}} &= f_N(A_{N-1}, \mathbf{X^{(N-2)}}) \\
\mathbf{X^{(N)}} &= f_N(A_N, \mathbf{X^{(N-1)}}) \\
\mathbf{Y} &= f_N(A_n, f_{N-1}(A_{N-1}, \dots f_1(A_1, \mathbf{X})))
\end{aligned}$$

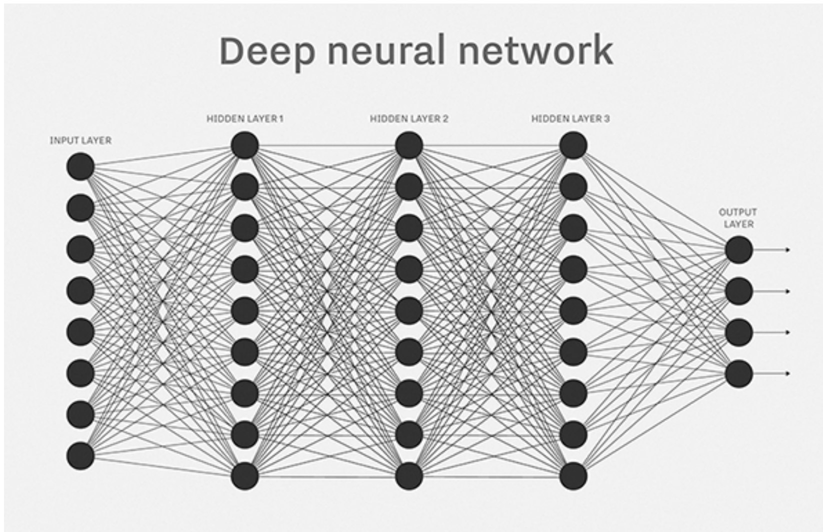


Figure 3: Three Layer Network

The mapping and remapping will repeat for N times until we get to the final layer, the output layer. And the **Y** value will be the perceptron of the whole network.

## 3. Algorithm Implementation and Development

### *3.1. Single Layer*

For the single layer, I modified synsis's work to build a weight algorithm. The general algorithm is to find a weight $A_j$ such that for every pattern (in this project, $j \in [0, 9]$) the outer product of the pixel vector and the weight will generate the perceptron to detect the digit image.

$$Y_{ij} = A_{ij} * X_{ij}, where\ i = 1,\ 2,\ \ldots N,\ j = 1,\ 2,\ \ldots 10$$

And for every training image, the training process will result in either 1 for correct detection or -1 for wrong detection. The algorithm itself will have a learning rate at $\alpha = 0.1$ to know that if the weight is good or not.

### 3.2. *Two Layers*

For the two-layer network, I basically used the same idea as I did in the single-layer network, but this time train the model use two weights, $w_{between}$ and $w_{output}$.

## 4. Computational Results

### 4.1. *Relearning Times*

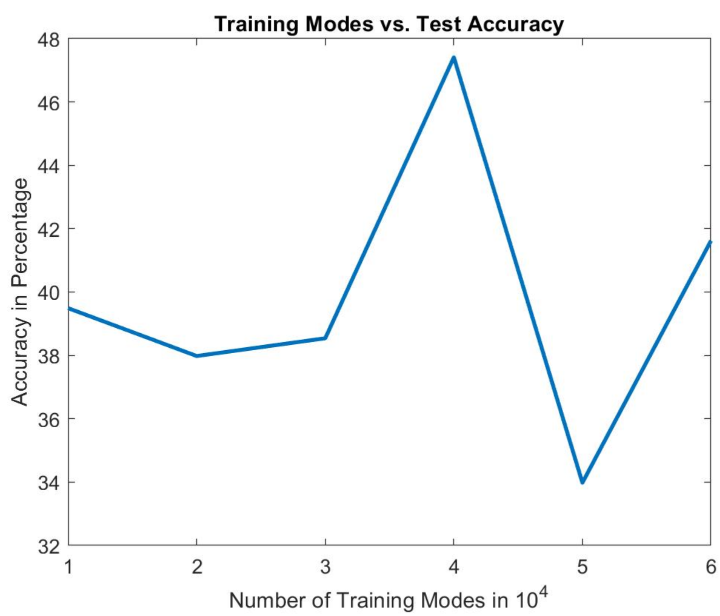


Figure 4: Accuracy vs. Relearning times

Figure 4 shows how the accuracy changes as the relearning times increases.

### 4.2. *Number of Epochs*

Figure 6 shows how the error rate changes as the number of epochs increases.

As the result, the accuracy of one single layer is about $39.84 \pm 0.2\%$ and the accuracy of two layers is about 92%

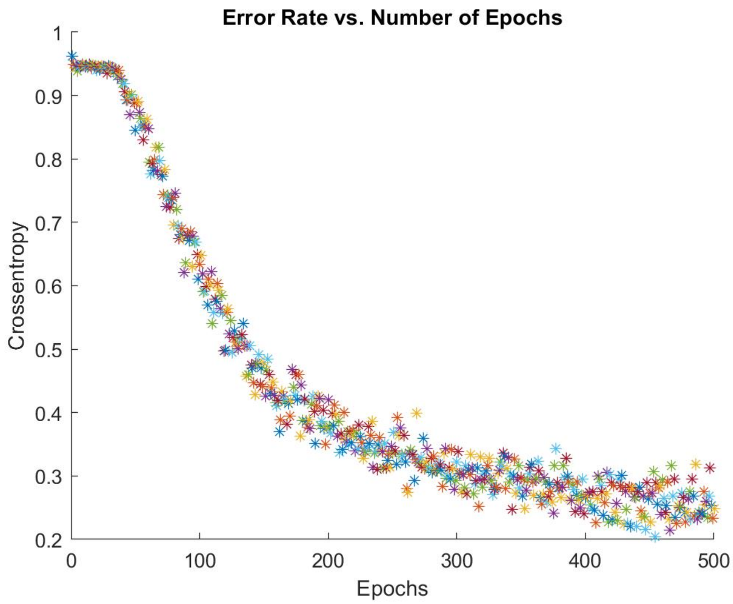


Figure 5: Performance of the network

```
Classification errors: 827
Correctly classified: 9173
Accuracy: 9.173000e-01
```

Figure 6: Result Output of Two-Layer Network

## 5. Summary and Conclusion

As a conclusion, one single layer network does not perform well enough on detecting digits. A 40% accuracy is not considered to be reliable. Meanwhile, a two-layer network has much more better performance on the accuracy.

```
clc; clear all; close all;
%% Import test data
test_img = loadMNISTImages('t10k-images.idx3-ubyte');
test_lab = loadMNISTLabels('t10k-labels.idx1-ubyte');
test_lab = test_lab';
%% A using weight bias algorithm
tic
acc = zeros(6, 1);
for n = 1:6
    w =find_weights(5, n);
    % Test
    pattern=10;element=size(test_img, 2);pixel=size(test_img, 1);
    e=ones(pattern,element);
    for j=1:pattern
        for i=1:element
            y=sign(test_img(:,i)'*w(j,:)');
            if mod(j,10)==test_lab(i)
                d=1;
            else
                d=-1;
            end
            e(j,i)=d-y;
        end
    end
    sum=0;
    for j=1:pattern
        for i=1:element
            if e(j,i)~=0
                sum=sum+1;
            end
        end
    end
    % Find accuracy
    acc(n) = sum/element;
end
toc
%%
plot(1:6, acc * 100, 'LineWidth', 2);
title('Training Modes vs. Test Accuracy');
xlabel('Number of Training Modes in 10^4');
ylabel('Accuracy in Percentage');

clc; clear all; close all;
%%
% Load MNIST.
train_data = loadMNISTImages('train-images.idx3-ubyte');
train_labs = loadMNISTLabels('train-labels.idx1-ubyte');

targetValues = 0.*ones(10, size(train_labs, 1));
for n = 1: size(train_labs, 1)
    targetValues(train_labs(n) + 1, n) = 1;
end

units = 10;

alpha = 0.1;

activation = @logisticSigmoid;
dactivation = @dLogisticSigmoid;

batchSize = 100;
epochs = 500;

[w_between, w_output, error] = train(activation, dactivation, ...
    units, train_data, targetValues,epochs, batchSize, alpha);
```

```

test_data = loadMNISTImages('t10k-images.idx3-ubyte');
test_labs = loadMNISTLabels('t10k-labels.idx1-ubyte');

[correct, err] = validate(activation, w_between, w_output, test_data, test_labs);

fprintf('Classification errors: %d\n', err);
fprintf('Correctly classified: %d\n', correct);
fprintf('Accuracy: %d\n', correct/(err+correct));

function [w]=find_weights(times, modes)
data = loadMNISTImages('train-images.idx3-ubyte');
labels = loadMNISTLabels('train-labels.idx1-ubyte');
data = data(:, 1:modes*10000);
pattern=10;element=size(data, 2);pixel=size(data, 1);
alpha=0.1;% Learing rate
w=zeros(pattern,pixel);%weights
e=ones(pattern,element);%error
d=0;

for count=1:times
    for j=1:pattern
        for i=1:element
            y=sign(data(:,i)'*w(j,:)');
            if mod(j,10)==labels(i)
                d=1;
            else
                d=-1;
            end
            e(j,i)=d-y;
            w(j,:) = w(j,:)+alpha.*e(j,i).*data(:,i)';
        end
    end
end

end
function [w_bet, w_out, error] = train(act, deact, units, data, labs, epochs, batchSize, alpha)
    trainingSetSize = size(data, 2);
    datasize = size(data, 1);
    pattern = size(labs, 1);
    w_bet = rand(units, datasize);
    w_out = rand(pattern, units);

    w_bet = w_bet./size(w_bet, 2);
    w_out = w_out./size(w_out, 2);

    n = zeros(batchSize);

    figure; hold on;

    for t = 1: epochs
        for k = 1: batchSize
            % Select which input vector to train on.
            n(k) = floor(rand(1)*trainingSetSize + 1);

            % Propagate the input vector through the network.
            inputVector = data(:, n(k));
            hiddenActualInput = w_bet*inputVector;
            hiddenOutputVector = act(hiddenActualInput);
            outputActualInput = w_out*hiddenOutputVector;
            outputVector = act(outputActualInput);

            targetVector = labs(:, n(k));

            % Backpropagate the errors.
```

```
            outputDelta = deact(outputActualInput).*(outputVector - targetVector);
            hiddenDelta = deact(hiddenActualInput).*(w_out'*outputDelta);

            w_out = w_out - alpha.*outputDelta*hiddenOutputVector';
            w_bet = w_bet - alpha.*hiddenDelta*inputVector';
        end

        % Calculate the error for plotting.
        error = 0;
        for k = 1: batchSize
            inputVector = data(:, n(k));
            targetVector = labs(:, n(k));

            error = error + norm(act(w_out*...
                act(w_bet*inputVector)) - targetVector, 2);
        end
        error = error/batchSize;

        plot(t, error,'*');
        xlabel('Epochs'), ylabel('Crossentropy');
        title('Error Rate vs. Number of Epochs');
    end
end
function [correct, err] = validate(act, w_bet, w_out, data, labels)

    testSetSize = size(data, 2);
    err = 0;
    correct = 0;

    for n = 1: testSetSize
        inputVector = data(:, n);
        outputVector = evaluate(act, w_bet, w_out, inputVector);

        class = decisionRule(outputVector);
        if class == labels(n) + 1
            correct = correct + 1;
        else
            err = err + 1;
        end
    end
end

function class = decisionRule(output)
    max = 0;
    class = 1;
    for i = 1: size(output, 1)
        if output(i) > max
            max = output(i);
            class = i;
        end
    end
end

function output = evaluate(act, w_bet, w_out, data)
    output = act(w_out*act(w_bet*data));
end
function y = logisticSigmoid(x)
    y = 1./(1 + exp(-x));
end
function y = dLogisticSigmoid(x)
    y = logisticSigmoid(x).*(1 - logisticSigmoid(x));
end
```