

# CS 4222/5222 – Artificial Intelligence

## Lab 5: Live Handwritten Digit Recognition 20 points

### Goal

To create a feedforward backpropagation multilayer neural network for learning how to classify real handwritten digits in real time. I have provided you with a graphical user interface frontend that will use your neural network to

1. try to recognize digits pre-loaded from a data file
2. try to recognize digits that you draw on the screen.

The file `lab5.py` contains the code for loading the training and testing data (discussed below), as well as the GUI. It interfaces with a library called `NeuralNet`, which you will need to write. Skeleton code is in the `NeuralNet.py` file.

### Data

The data set consists of real handwritten digits automatically scanned from zip codes on envelopes by the US postal service. The data has been deslanted and normalized, so that each data point is a 256 element vector representing a 16x16 grayscale pixel array. This dataset (and many others) is hosted at <https://web.stanford.edu/~hastie/ElemStatLearn/data.html> as a companion to the book *The Elements of Statistical Learning* by Hastie, Tibshirani and Friedman.

The training and testing data are provided for you in the Canvas resource for this assignment in the files `train.dat` and `test.dat`.

### Overview

Your implementation will closely follow the pseudocode for BACK-PROP-LEARNING (see Figure 18.24 below) to train a three layer (only one hidden layer) feedforward net. However, we'll make our life slightly easier by using some linear algebra tricks (instead of painstakingly looping over all the example points).

Each data point  $(\mathbf{x}, \mathbf{y})$  is described as follows. Each example  $\mathbf{x} = (x_1, x_2, \dots, x_{256})$  is a 256-element vector with grayscale picture values from a particular scanned image of a digit. The corresponding  $\mathbf{y} = (y_0, y_1, \dots, y_9)$  is a 0/1 vector with a 1 in the position  $y_i$  if  $\mathbf{x}$  is an image of digit  $i$ .

```


function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
           network, a multilayer network with  $L$  layers, weights  $w_{i,j}$ , activation function  $g$ 
  local variables:  $\Delta$ , a vector of errors, indexed by network node

  repeat
    for each weight  $w_{i,j}$  in network do
       $w_{i,j} \leftarrow$  a small random number
    for each example (x, y) in examples do
      /* Propagate the inputs forward to compute the outputs */
      for each node  $i$  in the input layer do
         $a_i \leftarrow x_i$ 
      for  $\ell = 2$  to  $L$  do
        for each node  $j$  in layer  $\ell$  do
           $in_j \leftarrow \sum_i w_{i,j} a_i$ 
           $a_j \leftarrow g(in_j)$ 
      /* Propagate deltas backward from output layer to input layer */
      for each node  $j$  in the output layer do
         $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$ 
      for  $\ell = L - 1$  to 1 do
        for each node  $i$  in layer  $\ell$  do
           $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$ 
      /* Update every weight in network using deltas */
      for each weight  $w_{i,j}$  in network do
         $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ 
  until some stopping criterion is satisfied
  return network

```

**Figure 18.24** The back-propagation algorithm for learning in multilayer networks.



For example, the image  is represented by the data point  $(\mathbf{x}, \mathbf{y})$  where  $\mathbf{x}$  is its 256 grayscale pixel values (normalized to lie in  $[-1, 1]$ ) and

$$\mathbf{y} = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0),$$

where  $y_3$  specifies the example belongs to the class of “3” digits.

We will store all of the  $N$  training points  $(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \dots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})$  in a big  $N \times 256$  matrix  $\mathbf{X}$  where

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_{256}^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_{256}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_{256}^{(N)} \end{bmatrix}$$

and an  $N \times 10$  matrix  $\mathbf{Y}$  where

$$\mathbf{Y} = \begin{bmatrix} y_0^{(1)} & y_1^{(1)} & \dots & y_9^{(1)} \\ y_0^{(2)} & y_1^{(2)} & \dots & y_9^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ y_0^{(N)} & y_1^{(N)} & \dots & y_9^{(N)} \end{bmatrix}$$

To test the neural net, the program will also load testing points in similar matrices  $\mathbf{X}_{\text{test}}$  and  $\mathbf{Y}_{\text{test}}$ . The program will read in the data and enter a command loop where you can direct it to learn how to recognize digits (specifying the number of epochs and size of the hidden layer).

## Procedure

Examine the pseudocode for BACK-PROP-LEARNING posted in the figure above. The authors choose to loop over each example, each layer, and each node to compute  $in_j$  as  $\sum_i w_{i,j} a_i$ . We can do this in one quick matrix operation using  $\mathbf{X}$  as defined above. Suppose we designate  $m$  neurons in the hidden layer. Then we can store the weights between the input layer and the hidden layer in a  $256 \times m$  weight matrix  $\mathbf{W}_1$  and compute the matrix

$$\mathbf{in} = \mathbf{XW}_1.$$

This produces an  $N \times m$  matrix  $\mathbf{in}$  where  $in_{i,j}$  is the input to the  $j$ -th hidden node for the  $i$ -th example.

We can then compute the activation for each hidden node and each example as the  $N \times m$  matrix  $\mathbf{a}$  defined as

$$\mathbf{a} = g(\mathbf{in})$$

by applying the activation function  $g$  applied element-wise to the  $\mathbf{in}$  matrix.

This can also be done for the output layer by taking an  $m \times 10$  weight matrix  $\mathbf{W}_2$  and computing

$$\mathbf{in}' = \mathbf{aW}_2, \quad \mathbf{a}' = g(\mathbf{in}').$$

The output layer now contains the neural excitations in the  $N \times 10$  matrix  $\mathbf{a}'$  where  $a'_{i,j}$  contains a value between 0 and 1 that represents the prediction that the  $i$ -th example is the digit  $j$ .

Now we just need to adjust the weights by measuring how far off we are from the training example targets  $\mathbf{Y}$  and performing backpropagation.

**Backpropagation.** The observed error at the output layer is just

$$\mathbf{Y} - \mathbf{a}'$$

$$\Delta_{out} = (\mathbf{Y} - \mathbf{a}') \odot g'(\mathbf{a}')$$

where  $\odot$  denotes component-wise multiplication. The  $\Delta$  values for the hidden layer can be computed similarly. The weight updates can be written as

$$\begin{aligned}\mathbf{W}_1 &\leftarrow \mathbf{W}_1 + \alpha \mathbf{a} \Delta_{hidden} \\ \mathbf{W}_2 &\leftarrow \mathbf{W}_2 + \alpha \mathbf{a}' \Delta_{out}\end{aligned}$$

**Bias units.** Recall that we would like each layer to have a bias unit that represents the input of “always one”. This helps to generalize the neural net by allowing it to have greater degrees of freedom for its underlying hypothesis function. To do this, we can bind a one column to the input matrix

$$\mathbf{a} = [\mathbf{1} \mid \mathbf{X}] = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_{256}^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_{256}^{(2)} \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & x_1^{(N)} & x_2^{(N)} & \dots & x_{256}^{(N)} \end{bmatrix}.$$

This gives us an  $N \times 257$  input node activation matrix, where the zeroth input is always 1. Similarly, we can define the hidden weight matrix  $\mathbf{W}_1$  to be  $257 \times m + 1$  and always set the first column to 1s.

## Implementing the neural net

The file `NeuralNet.py` contains the interface to a `NeuralNet` class for which you will provide the implementation. The two routines within this class that you’ll need to implement are `NeuralNet.train` and `NeuralNet.predict`.

The first function, `NeuralNet.train` is the BACK-PROP-LEARNING routine, listed in Algorithm 1. This is essentially identical to Figure 18.24 in the book, except we use the aforementioned matrix approach. The user interface (discussed below) will load the training data and pass it to `NeuralNet.train` in a data matrix  $\mathbf{X}$ , along with a target matrix  $\mathbf{Y}$  (the actual classes of the digits), the size of the hidden layer, and the number of epochs to train.

The second function, `NeuralNet.predict` simply implements the function

$$g(g(\mathbf{a}_0 \mathbf{W}_1) \cdot \mathbf{W}_2).$$

where  $\mathbf{a}_0$  is the input activation layer (input to the hidden layer),  $\mathbf{a}_0 \mathbf{W}_1$  is the output of the hidden layer, and  $g$  is the sigmoid activation function.  $\mathbf{W}_1$  and  $\mathbf{W}_2$  are the weight matrices between input and hidden, and hidden and output. The pseudocode is provided in Algorithm 2.

Matrices are quite easy to work with using the `numpy` package. The `NeuralNet.train` and `NeuralNet.predict` routines expect their matrix inputs ( $\mathbf{X}$  and  $\mathbf{Y}$ ) as `numpy` arrays. To access

---

**Algorithm 1:** Backpropagation learning

---

```
1 def NeuralNet.train( $\mathbf{X}, \mathbf{Y}, \text{hiddenLayerSize}, \text{epochs}$ ):
2    $n_i \leftarrow$  number of columns of  $\mathbf{X} + 1$ ; // size of input layer (inputs plus bias)
3    $n_h \leftarrow \text{hiddenLayerSize} + 1$ ; // size of hidden layer (hidden nodes plus bias)
4    $n_o \leftarrow 10$ ; // size of output layer
5    $\mathbf{W}_1 \leftarrow n_i \times n_h$  matrix with random entries in  $[-1, 1]$ ;
6    $\mathbf{W}_2 \leftarrow n_h \times n_o$  matrix with random entries in  $[-1, 1]$ ;
7    $\alpha \leftarrow 0.001$ ; // Learning rate
8   for  $\text{epoch}$  in  $\{1, 2, \dots, \text{epochs}\}$  do
9      $\mathbf{a}_0 \leftarrow [\mathbf{1} \mid \mathbf{X}]$ ; // activation of input layer
10     $\mathbf{in}_0 \leftarrow \mathbf{a}_0 \mathbf{W}_1$ ; // input to hidden layer
11     $\mathbf{a}_1 \leftarrow g(\mathbf{in}_0)$ ; // activation of hidden layer
12     $\mathbf{a}_1[:, 0] \leftarrow 1$ ; // set bias unit for hidden layer
13     $\mathbf{in}_1 \leftarrow \mathbf{a}_1 \mathbf{W}_2$ ; // input to output layer
14     $\mathbf{a}_2 \leftarrow g(\mathbf{in}_1)$ ; // activation of output layer
15     $\mathbf{err}_{\text{out}} \leftarrow \mathbf{Y} - \mathbf{a}_2$ ; // observed error on output
16     $\Delta_{\text{out}} \leftarrow \mathbf{err}_{\text{out}} \odot g'(\mathbf{a}_2)$ ; // direction of target
17    Record MSE (provided);
18     $\mathbf{err}_{\text{hidden}} \leftarrow \Delta_{\text{out}} \mathbf{W}_2^T$ ; // contribution of hidden nodes to error
19     $\Delta_{\text{hidden}} \leftarrow \mathbf{err}_{\text{hidden}} \odot g'(\mathbf{a}_1)$ ; // direction of target for hidden layer
20     $\mathbf{W}_1 \leftarrow \mathbf{W}_1 + \alpha \cdot \mathbf{a}_0^T \Delta_{\text{hidden}}$ ; // hidden layer weight update
21     $\mathbf{W}_2 \leftarrow \mathbf{W}_2 + \alpha \cdot \mathbf{a}_1^T \Delta_{\text{out}}$ ; // output layer weight update
```

---

---

**Algorithm 2:** Classification

---

```
1 def NeuralNet.predict( $\mathbf{X}$ ):
2    $\mathbf{a}_0 \leftarrow [\mathbf{1} \mid \mathbf{X}]$ ; // activation of input layer
3   return sigmoid(sigmoid( $\mathbf{a}_0 \mathbf{W}_1$ )  $\mathbf{W}_2$ ); // compute activation of output layer
```

---

the dimensions of a **numpy** array, you can use the **shape** attribute which retrieves a tuple containing the number of rows and columns.

```
from numpy import *
X = array([[0,1,2],[3,4,5],[6,7,8]])
print(f"X has {X.shape[0]} rows and {X.shape[1]} columns")
```

You can take the transpose of a matrix with **X.T**

```
print(X.T)

[[0, 3, 6],
 [1, 4, 7],
 [2, 5, 8]]
```

To bind a column of ones to a matrix, you can use **hstack**. Be sure to transform the array of ones into a column vector using transpose, as follows:

```
hstack((array([[1]*X.shape[0]]).T,X))

array([[1, 0, 1, 2],
       [1, 3, 4, 5],
       [1, 6, 7, 8]])
```

You can access a **numpy** array in clever ways. For example, to set the first column of a matrix to ones (hint, hint), you can use

```
a1[:,0] = 1
```

Matrix multiplication is straightforward using the **numpy.dot** procedure. To compute

$$\mathbf{XY} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1/2 & 1/2 \\ 3/10 & 3/10 \\ 1/10 & 1/10 \end{bmatrix}$$

in python, we can use:

```
X = array([[0,1,2],[3,4,5],[6,7,8]])
Y = array([[0.5,0.5],[0.3,0.3],[0.1,0.1]])
P = dot(X,Y)
```

To do element-wise matrix multiplication  $\mathbf{X} \odot \mathbf{Z}$ , just use the standard **\*** operation

```
X*Z
```

The activation function  $g$  and its derivative are provided for you as well:

```
def sigmoid(x):
    return 1 / (1+exp(-x))

def dsigmoid(x):
    return x * (1 - x)
```

## Graphical User Interface

The interface is written in `lab5.py` and uses the `tkinter` library that we have been using for graphics. When you run `lab5.py`, it will

1. automatically load the training and test data (make sure that `train.dat` and `test.dat` are in the same directory)
2. import your `NeuralNet` library and access your code through the provided interfaces.

After the application starts up, you specify the number of epochs and the hidden layer size through the text entry fields. When debugging, I suggest low numbers for these to keep it fast (but inaccurate). When you're ready to really put your neural net to the test, I suggest **1000 epochs** and **hidden layer size 10**. After you select these parameters, you press the **Train** button and wait for the neural net to train on the loaded training data.

You can then draw in the large white input box with the left mouse button. You can clear the input box by right-clicking in it. The pixels of this box are hard-wired to the input neurons of your network. As you draw, a histogram will appear at the bottom that represents the level of each output neuron. Each level represents how confident your neural net is that the digit is the corresponding number. **Hint: predictions are better if you utilize as much of the drawing area as you can to draw your digit.** Remember that the network was trained on images of all the same size, so you should try to recreate this when you're playing around with the network.

You may also test out any of the 2007 provided test data by entering a number between 0 and 2006 in the "load test instance" text box and hitting enter. This will load a test instance and place it on your neural net inputs for prediction (it also displays the test instance in the input box).

At any time after training, you may observe the mean-squared error profile during training by clicking the **MSE** button. This draws a plot of MSE as a function of epochs: the closer to zero, the lower the training error.

## Turn In

1. Please demonstrate your program.
2. Please submit your source code file `NeuralNet.py` on Canvas.
3. Please document your code!