MAT 128B: Project II: Using algebraic methods for optimization: backpropagation neural networks

Wednesday, March. 13th
To get full credit show all your work

Names:
[100 pts]

In this project we are going to implement a neural network to recognize hand written digits. As in the previous project you are expected to write a final report explaining how you organized your research group, the algorithms you used, the code you wrote, mathematical derivations (if needed), results and conclusions from your study.

Remember to describe how you are splitting the task with your team members and provide evidence you are using Github.

Data Set

- **i.- MINST data base** Read pages 179-180 in Greenbaum and Chartier. A neural network needs a *training data set* that you will use to tune up the parameters of the network and a *test set*. Details of the data base are given in Greenbaum and Chartier and the data set can be downloaded from the book's webpage.

- **ii.- Plot digits**. Implement a program that reads digits from the data base (see Fig. 4). Plot a couple of examples of the data base to convince yourself that the program is working. Compute the average digit as explained in problem 17a, plot them and compare them with those in page 180. These three examples should help you make sure that your program is working properly.

- **iii.- A neuron**. The implementation of a neuron is shown in Fig. 1. Each neuron consists of a set of weighted connections, and an internal activation function. Assume that a neuron has n input connections (from the data input or from other neurons). We will call the inputs $\{O_1, ..., O_n\}$ and the input weights $\{w_1, ..., w_n\}$. The value NET is calculated as explained in the figure and is the input of an activation function F (explained in Figure 2). Your next task is to implement a neuron where F is given on top o Fig. 1 and in Fig. 2.

  We will consider the activation function to be the sigmoidal (logistic) function shown in Fig. 2. Notice that its derivative has a nice expression in terms of the OUT value. (Verify that this expression is correct and include it in your report). Analyze this logistic function. What is going to be the output of the activation function for small vs large value of NET ? What other functions can you use and what would be the effect between input and output?

- **iv.- Multilayer Network** Figure 3 shows the structure of a network with one input layer, one output layer and one hidden layer. The last layer in the figure (labeled as TARGET) is not part of the network but instead contains the values of the training set against which we are comparing the output. The INPUT LAYER does not perform calculations, it only takes the values from the data (See Figure 4). Neurons in the HIDDEN and the OUTPUT layer contain NET and OUT (as indicated above). The values $w_{i,j}$ is the weight connecting neuron i in layer 1 with neuron j in layer 2. The

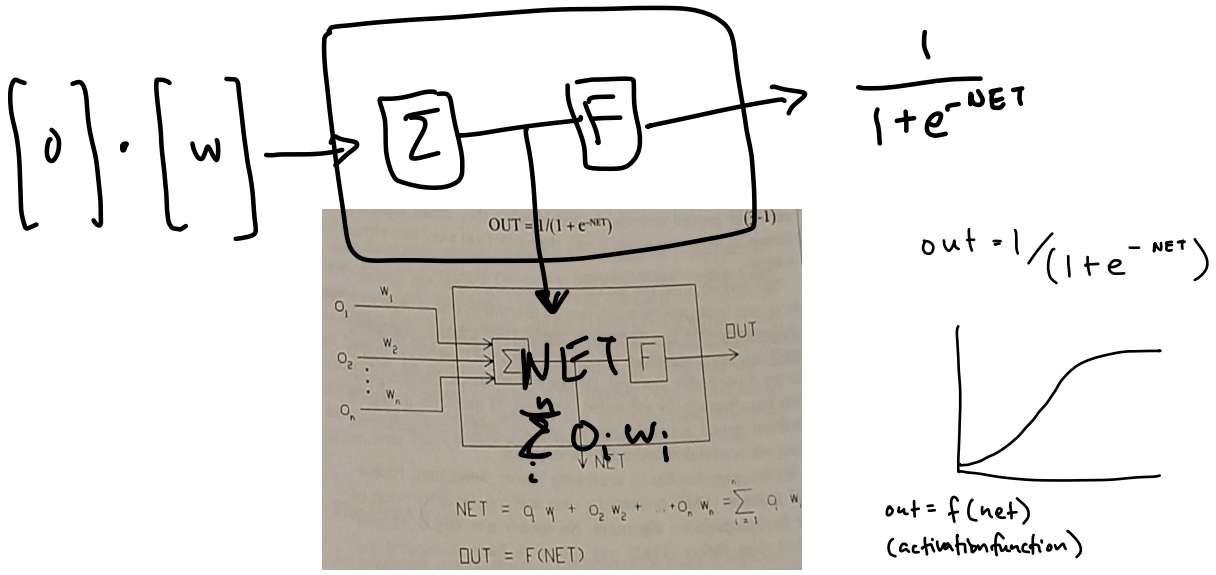


Figure 1: Structure of a neuron in the neural network. Given a set on inputs (to the neuron) and weights. The neuron first computes the value NET. The final output of the neuron will be given by the image of NET by the activation function F

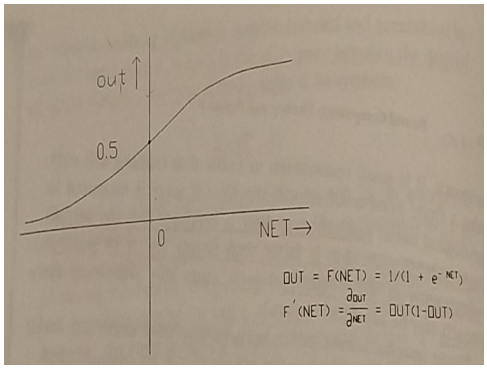


Figure 2: The activation function. The activation function can be any (bounded) differentiable function

output layer will produce OUTPUT that will be compared against the TARGET value. (Remember that we will be training the network, that means we know the input and output values). For instance, if we input the pixels for number five we will expect the network to output the number five. This will be compared with the target number five. The error can be encoded as 0 or 1 based on whether the network got it right or wrong. The network in Fig. 3 has only two layers (one hidden and one output) but a useful network will need more layers and more networks per layer. Implement a network with a variable number of hidden networks (and neurons per layer) in which each neuron has the structure explained above.

- **v.- Initializing the network** Initialize the network by assigning a random (small) number to each weight $w_{i,j}$.
- **vi.- Training the network** A network will learn by iteratively adapting the values of $w_{i,j}$. Each input is associated to an output. These are called *training pairs*. Follow these steps (i.e. general structure of your algorithm)
  - Select the next training pair (INPUT, OUTPUT) and apply the INPUT to the network
  - Calculate the output using the network

f(INPUT) = OUTPUT

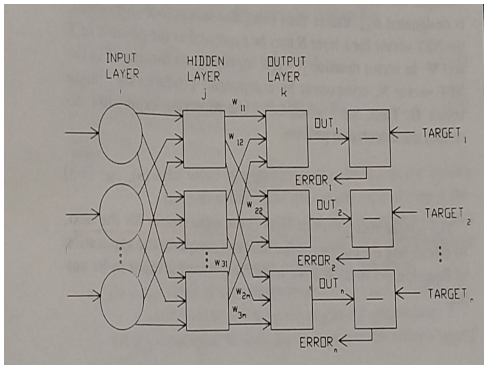


Figure 3: Multilayer network. Networks will have one INPUT, one OUTPUT layer and several HIDDEN layers. The number of layers and neurons per layer should be adjusted to the specific problem

- Calculate the error between the network's output and the desired output
- Adjust weights in a way that minimizes the error
- Repeat steps above for each training pair

Calculations are performed by layers, that is all calculations are performed in the hidden layer before any calculation is performed in the output layer. The same applies when several hidden layers are present. The first two steps above are called *forward pass* and the second two the *reverse pass*.

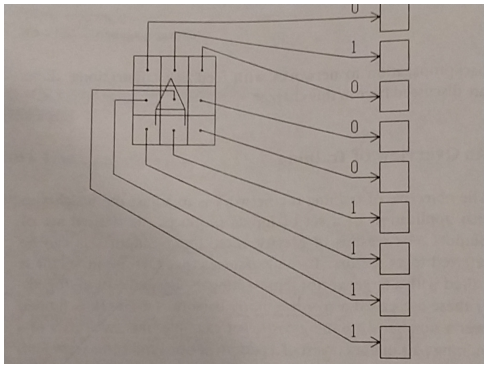


Figure 4: Structure of the INPUT layer for digit recognition

Forward pass: Notice that the weights in between layers of neurons can be represented by the matrix W and that if X is the input vector, then $NET = XW$ and the output vector is $O = F(NET)$. The output vector will be the input vector for the next layer.

Reverse pass. Adjusting the weights of the output layer: The ERROR signal is produced by comparing the OUTPUT with the TARGET value. We will consider the training process for a single weight from neuron p in the hidden layer j to neuron q in the output layer k (See Fig. 5). First we calculate the ERROR ($= TARGET - OUT$) for that output neuron k. This is multiplied by the derivative of the activation function for neuron k .

$$\delta_{q,k} = OUT_{q,k} * (1 - OUT_{q,k})(ERROR)$$

This value is further multiplied by the $OUT_{p,j}$ of the neuron p in hidden layer j and a training rate coefficient $\eta \in [0.01, 0.1]$. The result is added to the weight connecting the two neurons.

$$\Delta w_{pq,k} = \eta * \delta_{q,k} * OUT_{p,j};$$

Therefore the configuration of the output layer will change for the next training set as follows

$$w_{pq,k}(n+1) = w_{pq,k}(n) + \Delta w_{pq,k}$$

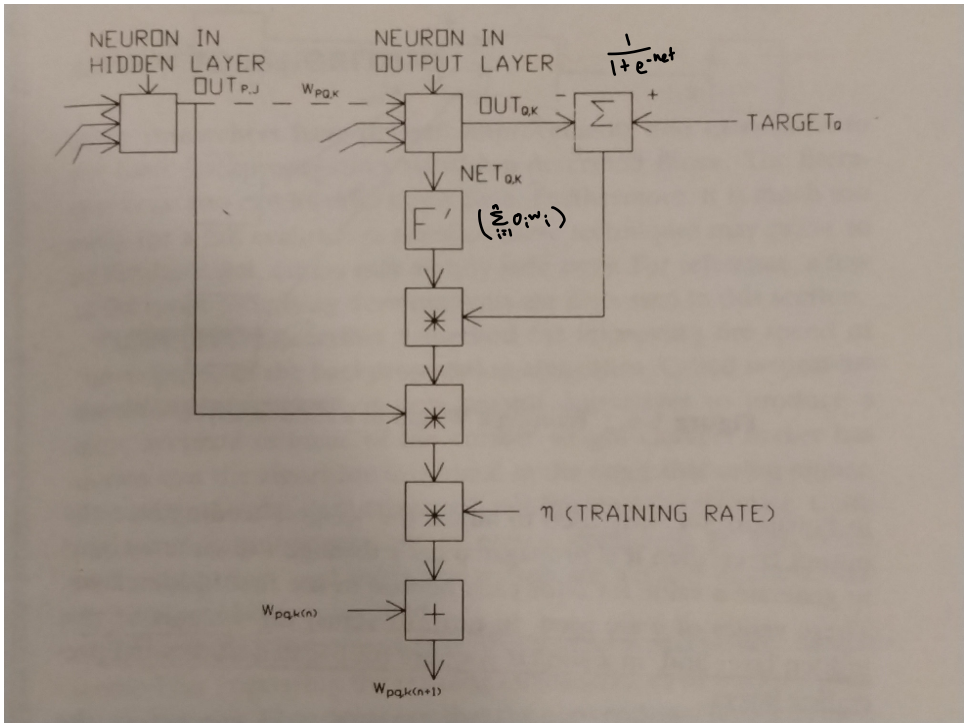


Figure 5: The training of each weight combines OUT values from the output and input neurons. The vertical line indicates the calculations that are needed to modify each weight

Reverse pass. Adjusting the weights of the hidden layers: Backpropagation will train the hidden layers by "propagating" the error back and adjusting the weights on its way. The algorithm uses the same two equations as above for $w_{pq,k}$ and $w_{pq,k}(n+1)$ however the value of δ needs to be computed differently.

The process is shown in Figure 6. Once δ has been calculated for the output layer it is used to compute the value of δ for each neuron.

$$\delta_{p,j} = OUT_{p,j}(1 - OUT_{p,j})(\sum_{k} \delta_{q,k} w_{pq,k})$$

It is important to realize that all the weights associated with each layer must be adjusted moving back from the output to the first layer.

- **vi.- Training the network** A network will learn by iteratively adapting the values of $w_{i,j}$. Each input is associated to an output. These are called *training pairs*. Follow these steps (i.e. general structure of your algorithm)
  - Select the next training pair (INPUT, OUTPUT) and apply the INPUT to the network
  - Calculate the output using the network
  - Calculate the error between the network's output and the desired output
  - Adjust weights in a way that minimizes the error
  - Repeat steps above for each training pair

Calculations are performed by layers, that is all calculations are performed in the hidden layer before any calculation is performed in the output layer. The same applies when several hidden layers are present. The first two steps above are called *forward pass* and the second two the *reverse pass*.

Forward pass: Notice that the weights in between layers of neurons can be represented by the matrix W and that if X is the input vector, then $NET = XW$ and the output vector is $O = F(NET)$. The output vector will be the input vector for the next layer.

Reverse pass. Adjusting the weights of the output layer: The ERROR signal is produced by comparing the OUTPUT with the TARGET value. We will consider the training process for a single weight from neuron p in the hidden layer j to neuron q in the output layer k (See Fig. 5). First we calculate the ERROR ($= TARGET - OUT$) for that output neuron k. This is multiplied by the derivative of the activation function for neuron k .

$$\delta_{q,k} = OUT_{q,k} * (1 - OUT_{q,k})(ERROR)$$

This value is further multiplied by the $OUT_{p,j}$ of the neuron p in hidden layer j and a training rate coefficient $\eta \in [0.01, 0.1]$. The result is added to the weight connecting the two neurons.

"η" "α"

$$\Delta w_{pq,k} = \eta * \delta_{q,k} * OUT_{p,j};$$

Therefore the configuration of the output layer will change for the next training set as follows

$$w_{pq,k}(n+1) = w_{pq,k}(n) + \Delta w_{pq,k}$$

Reverse pass. Adjusting the weights of the hidden layers: Backpropagation will train the hidden layers by "propagating" the error back and adjusting the weights on its way. The algorithm uses the same two equations as above for $w_{pq,k}$ and $w_{pq,k}(n+1)$ however the value of δ needs to be computed differently.

The process is shown in Figure 6. Once δ has been calculated for the output layer it is used to compute the value of δ for each neuron.

$$\delta_{p,j} = OUT_{p,j}(1 - OUT_{p,j})(\sum_k \delta_{q,k} w_{pq,k})$$ ← delta for adjusting weights of hidden layers

It is important to realize that all the weights associated with each layer must be adjusted moving back from the output to the first layer.

In matrix notation this is written as follows: If D_k is the set of the deltas at the output layer, W_k the set of weights at the output layer and D_j the vector of deltas for the hidden layer in the previous Fig. 3

$$D_j = D_k W_k^t \otimes [O_j \otimes (I - O_j)]$$

where $\otimes$ is defined to indicate the component-by-component multiplication of the two vectors. O_j is the output vector of layer j and I is the vector with all components equal to 1. Show that this last formula is correct.

$$\delta_2 = \delta_3 \cdot W_3^T \cdot (O_2 \cdot (I - O_2))$$

Calculating Error between final output & target & adjusting weight of last layer

Error = Target − Output (last layer)

δ = Output (1 − Output) (Error)

ΔW = δ × η × Output (last layer − 1)

new W_n = W_n + ΔW

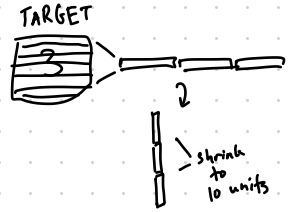
W = [784×100 | 100×100 | 100 × 10]

train0 matrix

5923 × 784 → 300 × 784

[0] (m) × m [W]

final out put



TARGET = mean (test #)

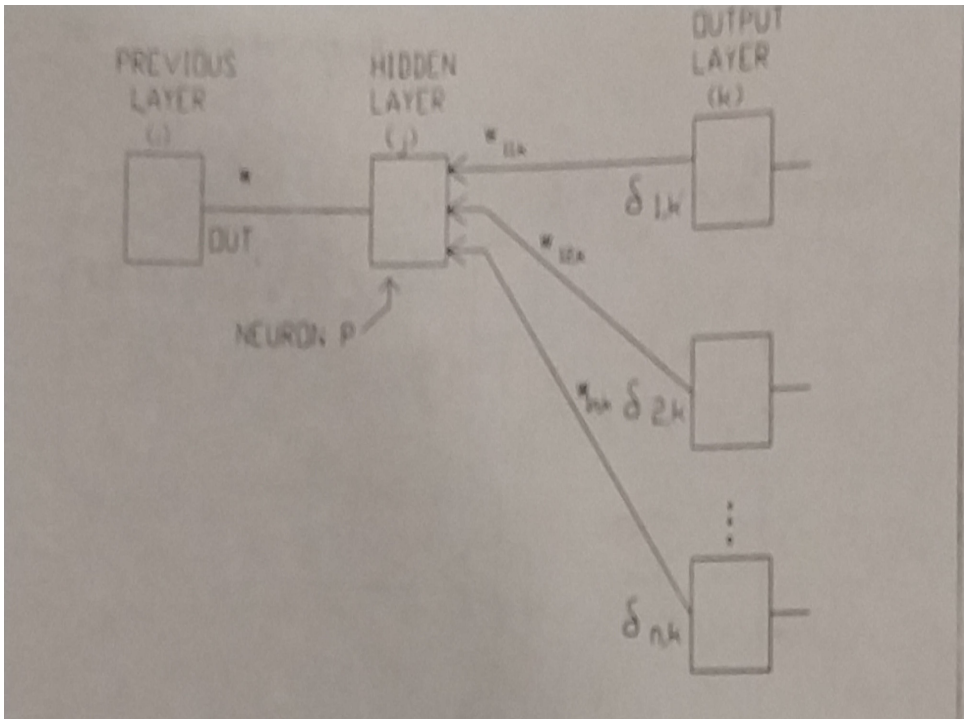


Figure 6: The training of each weight requires a new calculation of δ

In matrix notation this is written as follows: If D_k is the set of the deltas at the output layer, W_k the set of weights at the output layer and D_j the vector of deltas for the hidden layer in the previous Fig. 3

$$D_j = D_k W_k^t \otimes [O_j \otimes (I - O_j)]$$

where $\otimes$ is defined to indicate the component-by-component multiplication of the two vectors. O_j is the output vector of layer j and I is the vector with all components equal to 1. Show that this last formula is correct.

- **vii.- Dependence on parameters** The learning of the network (i.e. the minimization of the error) will depend on the number of layers, the number of neurons per layer, and for fixed values of these two parameters the network will also depend on the size of the training set. Set up a study in which you change the values of these parameters and report the error(s) you obtain (you will obtain an error for the training set and another for the test test–which should be very similar to each other provided the test and training set are similar enough).

References

– P.D. Wasserman. Neural Computing: Theory and Practice. Van Nostrand Reinhold

6. Training the network

- Select the next training pair (INPUT, OUTPUT) and apply the INPUT to the network
- Calculate the output using the network

} forward pass

- Calculate the error between the network's output and the desired output
- Adjust weights in a way that minimizes the error
- Repeat steps above for each training pair

} back pass

Required

Input: All train # matrices. *Dump training images & have it learn

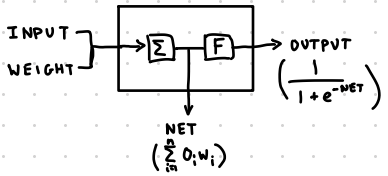
One test matrix.

Output: give recognized number.

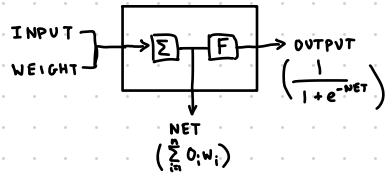
given  x100 to train. given  . Output # = 5.

What's a neuron?

1 neuron



2 neuron

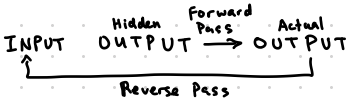


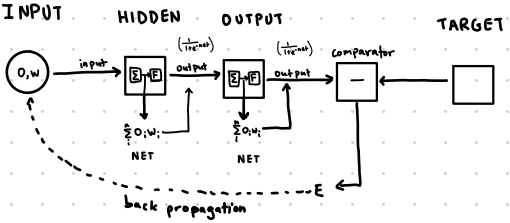
784 columns because a normalized 28x28 pixel image has 784 total pixels

The values in each matrix range from 0-256 since the image has been converted to an 8-bit greyscale image. 8-bit data has 256 values max.
(Ex: 0 = white ; 256 = black)

Backpropagation

Back Propagation increases or decreases weight to decrease error





$$E_n = \frac{1}{2}(\text{target}_n - \text{output})^2$$

$$E_{total} = \sum E = E_1 + \cdots E_n$$

Rate of change of error

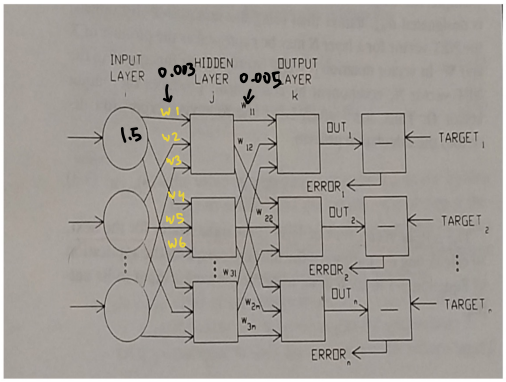
$$\frac{\delta E_{total}}{\delta W_n} = \frac{\delta E_{total}}{\delta \text{output } 1} \cdot \frac{\delta \text{output } 1}{\delta \text{net } 1} \cdot \frac{\delta \text{net } 1}{\delta W_n}$$

$$\frac{\delta E_{total}}{\delta \text{output } 1} = -(\text{target } 1 - \text{Output } 1)$$

$$\frac{\delta \text{output } 1}{\delta \text{net } 1} = \text{Output } 1 \cdot (1 - \text{Output } 1)$$

$$\downarrow \quad \frac{1}{1+e^{\text{net } 1}}$$

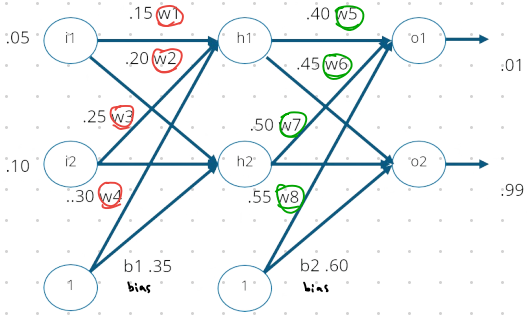
$$\frac{\delta \text{net } 1}{\delta W_n} = W_n \cdot (\text{Output } 1 + W_{n+1})_{\text{Hidden}}$$



Stochastic Gradient Algorithm

$$W_{n+1} = W_n - \eta \frac{\delta E_{total}}{\delta W_n}$$

η: training rate coeff $\in [0.01, 0.1]$



W = Each column vector of weighted matrix represents connection between 2 layers

input W output

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad \begin{bmatrix} w_1 & w_5 \\ w_2 & w_6 \\ w_3 & w_7 \\ w_4 & w_8 \end{bmatrix} \quad \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

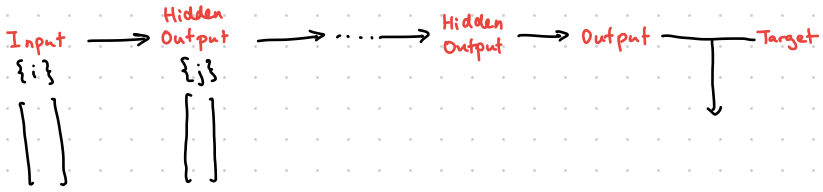
$a = 1w_1 + 2w_2 + 3w_3 + 4w_4$

$b = 2w_2 \ldots$

W = i x j values for 1 column

Output = $\sum$ input $\cdot$ W(:,1)

Code



Output: An minimum error showing the weights are trained to the data
[MIN_ERROR]

A percentage of how confident the answer is
[HOWSURE]

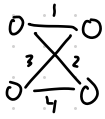
Ex: 86.7% sure this matrix of pixels is a "3"

Input: WEIGHT matrix
TARGET matrix
train # matrix

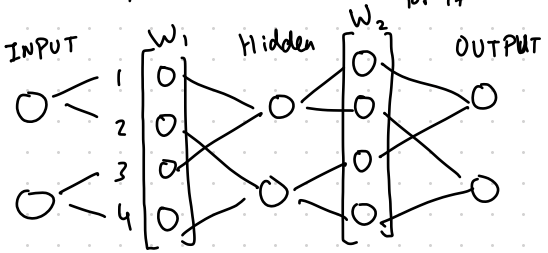
Appendix A: Precondition for faster convergence
Weight values rand $(-2.4/\text{inputs}, +2.4/\text{inputs})$
# inputs = 784 pixels
WEIGHT = rand (-0.003, +0.003)

2 columns
784 rows

WEIGHT
$((\text{inputrow})^2, \#\text{layers})$

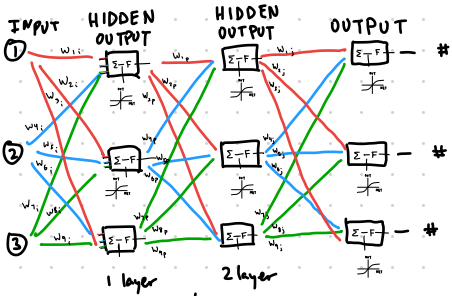


The issue is getting the input value to use the specific weight values saved for it



Is the output 1 value or a vector?

Hidden Output = vector
OUTPUT = 1 value



Hidden Output = vector

OUTPUT = 1 value

train 0
train 1
train 2
⋮
train 9

First step: normalize all input training matrice

6131 × 784

↓

[ ] normalized size

300 × 784 ←

you want to train

Let = 300 images so

Each train# set cut to 300 × 784 matrix

3-Layer NN (2 Hid. Layers)

Input = cell (1, 3)

Weight = cell (1, 3)

Weight{i} = rand([1010, 784]) between (-0.5, 0.5)

Output = cell (1, 3)

For : train3 1010 × 784
test3

Visualization : cell (1,3) = [1010×784 ; 1010×784 ; 1010×784]

Code

```
for 1: 1010
   O = Input{i}
   Output = cell ( 1, 3)
   for k = 1:3
      Net = O * Weight{j}
      O = 1 / (1 + e^-Net)
      Out{m} = O{i}
   end
```

Forward Pass

input = [ ] 784

Weight : [784×784] [ ] [ ]

3 layers each 784×784

[weight * input] gives $\sum_i O_i w_i$ for O_i

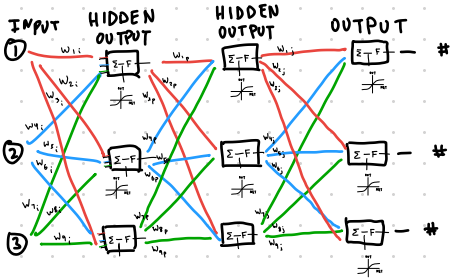
↓ net

[⋮] 784 ⇐ This is an array of net values for first layer

F(net) = Out for every value of first layer

Rinse & repeat for other layers

out = new input [⋮] 784



Answer Confidence

Check if output = target & how many are correct vs wrong. # correct

784 outputs

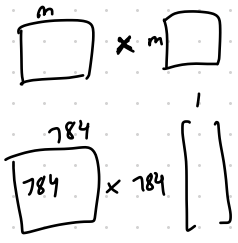
760 right
24 wrong

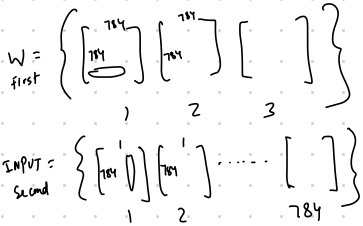
760 / 784

Create check if output is close enough to target & if it is correct, use correct total to get answer confidence

Troubleshooting

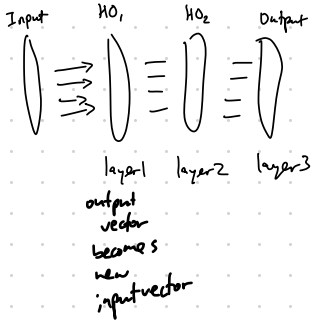
Make weight matrix same # columns as input matrix # rows



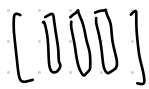


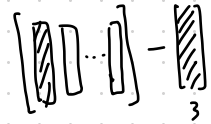

```
for 1:784
  0 = input first column
    for p=1:3
      for 1:784
        weight = Weight cell 1 firstrow
        Net = Wc1 1stRow * 0
        0 = F(Net)
end
```

```
hiddenoutput = cell(1,layers);
for L = 1 : layers
    weight = WEIGHT{L};
    for i = 1:784
        o = INPUT{i};
        for j = 1:784
            Net = weight(j,:) * o;
            o(i) = 1/(1+e^-NET)
        end
    end
    hiddenoutput{L} = o;
    OUTPUT = hiddenoutput{layers};
    for ii = 1:784
        ERROR = double(TARGET{ii}) - double(OUTPUT);
        δ = OUTPUT * (1 - OUTPUT) * ERROR;
        ΔW = η * δ * OUTPUT;
        WEIGHT{Layers} = WEIGHT{Layers} + ΔW;
        for w2 = (layers-1) : -1 : 1
            OUTPUT = hiddenoutput{w2}
            δ = OUTPUT * (1 - hiddenoutput{w2+1}) * OUTPUT
```



Gives hiddenoutput cell





Weights { 784×100 ; 100×100 ; 100×10 }

o { 1×784 columns, 1×784, ... ; 1×784 }

First iteration

o = input{800} = 1×784

W = weights{1} = 784×100 (100, 10)

NET = 1×100

o = F(NET) = 1×100

W = weight{2} = 100×100

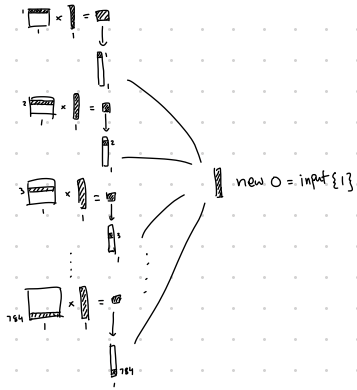
NET = 1×100

o = F(NET) = 1×100

W = weight{3} = 100×10

NET = 1×10

o = 1×10



TRAIN SET : [300×784, 300×784, ... , 300×784]

0 , 1 , ... 9

For each train sample put in,
it gives output, compares to target
& back propagates & changes weights

Then it moves onto next sample image.

* What number doesn't matter.
As long as input & target are for same image, it works.

- we want the total cost of network not just for 1 #.
- Test vectors run through network & weights aren't changed.
Those vectors become our TARGET DATASET.

```
for i = 1: length(INPUT)_784
    O = INPUT(i)
  NET = O * W1
  output = 1./(1+exp(-NET)
    O = output
      for j = 1: length(W1)_100
```

```
function output = actfunc ( NET)
output = 1./(1+exp(NET))
end
```