Neural Network: Multi-Layer Perceptron and Back Propagation

2019 Spring, CS902 Thinking and Approach of Programming

Kaiwen Tan, School of Mechanical Engineering, Shanghai Jiao Tong University

Introduction

Although majoring in mechanical engineering, I like coding very much. This semester, thanks to CS902, I have a chance to approach to the cutting-edge technology - neural network - and even build one by myself. And during the course, I met two talented guys, Bill Chen from China and Lorenz Kleissner from Austria. We achieve our ideal goals through fun, international teamwork. And later we become good friends. Every teammate is indispensable so I'm going to use "we" instead of "I" in this report.

To make our first acquaintance with neural network, we read some articles on the internet (Dr. Andrew Ng's AI lectures, for instance). After we know about the basic ideas of multi-layer perceptron (MLP) and its back propagation (BP), we try to build our own network with the help of the template file. Then we convert the data file to suit the interface of the MLP and train it for thousands of time. Finally, we test the MLP using the data provided, and compare the output with the correct answer.

Main Ideas

About Multi-Layer Perceptron (MLP) & Back Propagation (BP)

Understanding the fundamental ideas of MLP is the key to the project. Take Fig. 1, for a basic example. This MLP only has three layers: an input layer, an output layer, and a hidden one. Firstly, we initialize w1~w8 and b1~b2 with not particularly exact values as shown in Fig. 2. Our goal is to train the MLP with an input of (0.05, 0.10) and output of (0.01, 0.99). In other words, after many times of training, when we input (0.05, 0.10), our MLP can automatically get us (0.01, 0.99). Our activation function is the sigmoid function:

$$y = \frac{1}{1 + e^{-x}} \tag{1}$$

When we input i1 and i2 values like (0.05, 0.10), net values of h1 and h2 can be

$$net_{h1} = w_1 * i1 + w_2 * i2 + b_1 * 1 \tag{2}$$
$$net_{h2} = w_3 * i1 + w_4 * i2 + b_1 * 1 \tag{3}$$

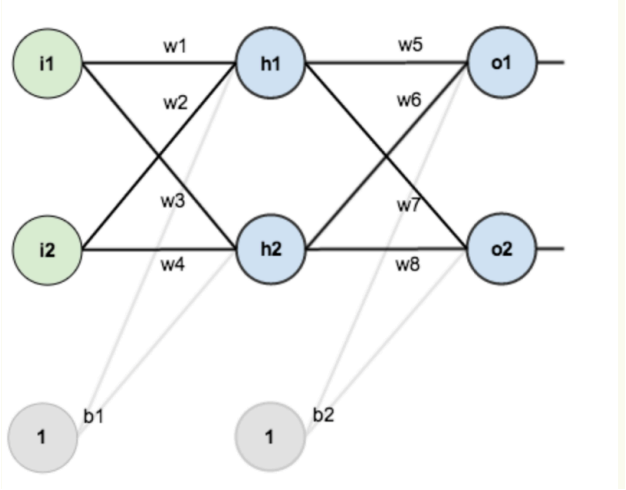

Fig. 1 A three-layer MLP

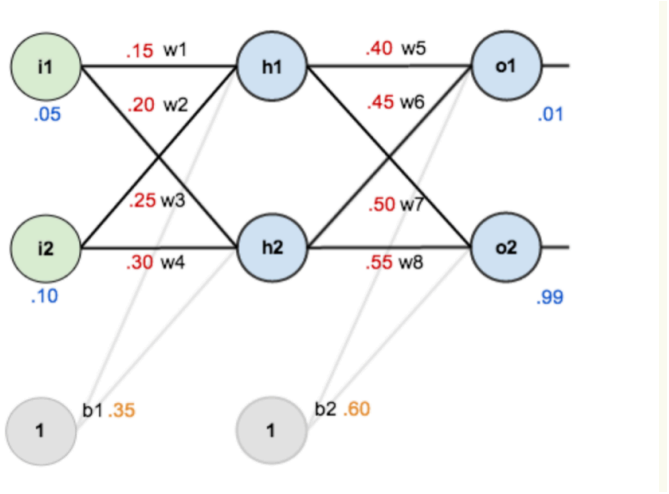

Fig. 2 A three-layer MLP with parameters

We convert those net values to real h1, h2 values using the sigmoid function. And we get

$$h1 = \frac{1}{1 + e^{net_{h1}}} \tag{4}$$

$$h2 = \frac{1}{1 + e^{net_{h2}}} \tag{5}$$

With the same method, we can get o1 and o2 values, which are generated by h1 and h2. Now let's see how to implement the back propagation. Its main idea is to use partial differential method to get the calculus results, i.e., total error partial over w1, w2, and other weight parameters. We take w1, as an instance. First we get the overall error of the (o1, o2) output:

$$E_{total} = \sum \frac{1}{2}(target - output)^2 \tag{6}$$

$$E_{total} = E_{o1} + E_{o2} \tag{7}$$

The total error is made up of o1's error and o2's error. And o1's (or o2's) error is closely related to, (or we can say generated by) w5, w6, w7, w8, and b2. So we get

$$\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w5} \tag{8}$$

The key of back propagation is to use this value to update w5. This process is a little bit like the Newton's method: every time we do it, the error shrinks a little. If the differential value is positive, then decrease w5 a little, and if negative, increase it a bit. So after nearly infinite times of training, w5's error will finally be close to zero. And that is our goal.

What I describe above is only about the weights between the last hidden layer and the output layer. If fact, we can apply the same method to w1, w2… because they are decided by the layers and weights after them. After we deal with the latter layers, we push back and update the former layer, then the former layer of the former layer…, until we reach the input layer. Then all of the weights like w1, w2, b1, b2, are prepared to get the exactly desired answer from the test input. This process is called the back propagation.

About the Project & Our Solution

After we understand the basic of MLP & BP, let's have a look at the problem we're dealing with.

A digital picture is made up of many pixels. So, if we are provided with a matrix of numbers, with each number indicating the greyscale/brightness value of that pixel, we can get the information we want: what the picture represents. Take a pixel picture of number 0, as an example:

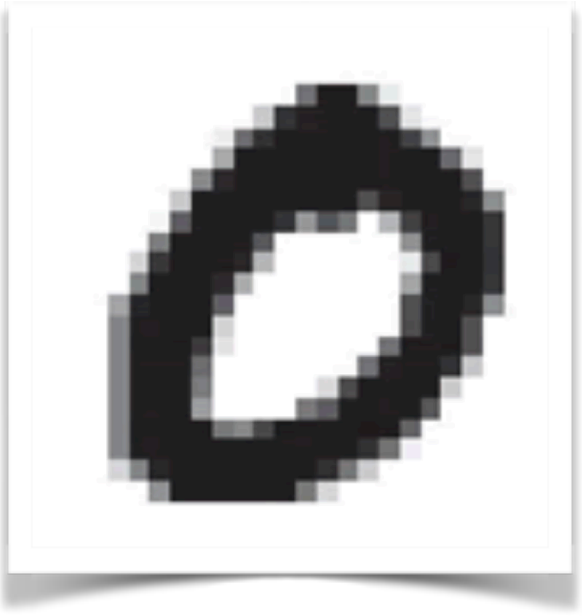

Fig. 3 A pixel picture of a handwritten number 0

Some pixels are blacker while others are whiter. In fact, every pixel has its own greyscale. And the greyscale values can be a list of numbers. This is the information provided by train_mnist.csv and test_mnist.csv. The first number of each row is the result number. For example, as Tab. 1 shows, 0 is the first one in the row, and there are 39 other numbers after it.

Tab 1. A Sample of Data Values

0	1.29E+03	-5.38E+02	3.37E+02	2.39E+02	1.13E+02	-5.04E+02	…

So technically we can train the MLP with those 39 numbers for many times. And after we finish training the MLP, we can input the test data, which represent number 0 as well, to get our MLP's result. If the number of training times is enough, the MLP will output a 0.

Understanding the main ideas of our project, we divide our project to two parts:
1) Implement MLP & BP.
2) Convert data from csv to suit the API of MLP & BP.

About the first part, you can refer to neural-network-final-for-windows.rkt. In this file, *Part 1 - Neuron Network Functions* contains *new-neuron* and *train* procedures. We create a new neuron to satisfy the need of back propagation. *Part 2 - Layer Functions* is all about layers, which is made up of *new-layer*, *link-layers*, *set-layer*, *reset-layer*, *train-layer*, *run-layer*, *sum-layer* and *train-layers*. Because the number of hidden layers is not decided, we should deal with multi layers. For example, users may create a MLP of '(39 50 50 10) which contains one input layer (of 39-D input), two hidden layers (each with 50 parameters), and one output layer (of 10-D output). While when he creates a '(39 60 10), there is only one hidden layer. *Part 3 - Mlp Functions* (containing *new-mlp*, *link-mlp*, *reset-mlp*, *run-mlp*, *train-mlp*) is about the MLP object the user operates. If the user wants to create a new MLP, firstly he should call *new-mlp* with an input list for example, '(39 50 10). Then he can train the MLP and finally test it. Last, in *Part 4 - Calculation Functions*, there are procedures of *sigmoid*, *rand-weight*, *rand-theta*, *push!*, *sum-weight* and *round-output*. They are called in other parts of the code.

For the second part, I'd like to emphasize how we manage to use the knowledge we learnt in our code. The csv files provided contain the data we need (in string format, sadly), so first we convert the csv files, which can be considered a list of lists of strings, to datasets (a list of lists of numbers). Then we map the dataset. For each list of numbers, the car is the result number (the meaningful information, like the 0 shown in Fig. 3), and the cdr is the input (39-D) we train the MLP with. However, to train the MLP, we can't feed it with a single digit result. We need conversion. For example, after we have our list like

```
(define lst '(0 3.13 4.21 ... 1.21))
```

We should't call

```
(train-mlp my-mlp (cdr lst) (car lst))
```

Instead, we should transfer the digit 0 to its binary vector version, '(1 0 0 0 0 0 0 0 0 0). So I wrote a procedure called *digit-transfer*

```
; Transfer a decimal digit number to a 10-element binary list (e.g. 0 -> '(1 0 0
1 0 0 0 0 0 0))
(define (digit-transfer n)
  (define (digit-iter r i)
    (define flag
      (if (= i n) 1 0))
    (if (> i 9) '() (cons flag (digit-iter r (+ i 1)))))
  (digit-iter '() 0))
```

So we should call the following to train the MLP:

```
(train-mlp my-mlp (cdr lst) (digit-transfer (car lst)))
```

As you can see, to implement *digit-transfer*, I use an iteration. I think it's great to use what we learnt from class.

When I test the code on my Mac, there's a bug (it won't happen on a Windows PC). In the csv table, the last number of each row will be converted to a #f, implying a conversion error. For example, during the transition, '("4" "3.13" "4.21" ... "1.21") will become ("4" "3.13" "4.21" ... #f), so we lose the last dimension. Instead of changing the conversion method, I choose to add a new column of zeros to the source csv table. So now the last zero will become #f, I just need to drop the last element of '("4" "3.13" "4.21" ... "1.21" #f). The implementation of *drop-last* procedure also uses the recursion idea of LISP:

```
; Drop the last element of a non-empty list
(define (drop-last lst)
  (if (eq? (cdr (cdr lst)) '())
      (list (car lst))
      (cons (car lst) (drop-last (cdr lst)))) )
```

As you can see in my code, the testing is made up of *Part 5 - Testing with XOR Function* and *Part 6 - Testing with Simple-MNIST Dataset*.

Results

The ideal answer should be

```
; For XOR test
'(0)
'(1)
'(1)
'(0)

; For MNIST dataset
'(7 2 1 0 4 1 4 9 5 9)
```

```
; Or its binary version
'((0 0 0 0 0 0 0 1 0 0)
  (0 0 1 0 0 0 0 0 0 0)
  (0 1 0 0 0 0 0 0 0 0)
  (1 0 0 0 0 0 0 0 0 0)
  (0 0 0 0 1 0 0 0 0 0)
  (0 1 0 0 0 0 0 0 0 0)
  (0 0 0 0 1 0 0 0 0 0)
  (0 0 0 0 0 0 0 0 0 1)
  (0 0 0 0 0 1 0 0 0 0)
  (0 0 0 0 0 0 0 0 0 1))
```

When we apply an MLP of '(2 8 1) to XOR with 100,000 times of training, the result always looks very nice. However, when training a '(39 40 40 10) MLP with MNIST dataset for hundreds of times, the result is not very satisfactory. The MLPs in Fig. 4, 5 and 6, are trained for 100, 500, and 1,000 times. In the code, I convert the binary version to a digital one. For example, if the output of the MLP is '(1 0 0 0 0 0 0 0 0 0), I call a procedure named *digit-reverse* to convert it to a number 0. The procedure always finds the first 1 in the list and returns its sequence number. If no 1 is found, it returns an n. (We use an app called DrRacket. Thanks to this app, we can write LISP code very easily.)

We think if we train it for more times, with more hidden layers, the result could be much better. However, it may cost us a huge amount of time.

```
; Train mlp for many times
(do ((i 1 (+ i 1)))
        ((> i 100) #t)
  (train-data dataset) (write i))


; Test mlp
; If you want to see the binary version (a 10 * 10 matrix), uncomment the following line
; (test-data  dataset-test)
(map digit-reverse (test-data  dataset-test))
```

```
Welcome to DrRacket, version 7.2 [3m].
Language: racket, with debugging; memory limit: 4000 MB.
#t
'(0.015558806392653602)
'(0.9848265721593406)
'(0.9847585868259088)
'(0.016374616623746074)
'(0)
'(1)
'(1)
'(0)
12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849574757677787980818283848586878889909192939495969798991 00#t
'(7 n 1 0 4 1 n 1 n n)
>
```

Fig. 4 Result of 100 times of training

```
; Train mlp for many times
(do ((i 1 (+ i 1)))
        ((> i 500) #t)
  (train-data dataset) (write i))


; Test mlp
; If you want to see the binary version (a 10 * 10 matrix), uncomment the following line
; (test-data  dataset-test)
(map digit-reverse (test-data  dataset-test))
```

```
'(0.9797759480866045)
'(0.9801136147781347)
'(0.01688380162085219)
'(0)
'(1)
'(1)
'(0)
12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849
74757677787980818283848586878889909192939495969798991001011021031041051061071081091101111
28129130131132133134135136137138139140141142143144145146147148149150151152153154155156157
17417517617717817918018118218318418518618718818919019119219319419519619719819920020120220
92202212222232242252262272282292302312322332342352362372382392402412422432442452462472482
65266267268269270271272273274275276277278279280281282283284285286287288289290291292293294
31131231331431531631731831932032132232332432532632732832933033133233333433533633733833934
63573583593603613623633643653663673683693703713723733743753763773783793803813823833843853
02403404405406407408409410411412413414415416417418419420421422423424425426427428429430431
44844945045145245345445545645745845946046146246346446546646746846947047147247347447547647
34944954964974984995005015025035045055065075085095105115125135145155165175185195205215225
39540541542543544545546547548549550551552553554555556557558559560561562563564565566567568
58558658758858959059159259359459559659759859960060160260360460560660760860961061161261361
06316326336346356366376386396406416426436446456466476486496506516526536546556566576586596
76677678679680681682683684685686687688689690691692693694695696697698699700701702703704705
72272372472572672772872973073173273373473573673773873974074174274374474574674774874975075
77687697707717727737747757767777787797807817827837847857867877887897907917927937947957967
13814815816817818819820821822823824825826827828829830831832833834835836837838839840841842
85986086186286386486586686786886987087187287387487587687787887988088188288388488588688788
49059069079089099109119129139149159169179189199209219229239249259269279289299309319329339
50951952953954955956957958959960961962963964965966967968969970971972973974975976977978979
9969979989991000#t
'(7 n 1 n 4 1 n n n n)
> |
```

Fig. 5 Result of 500 times of training

```
; Train mlp for many times
(do ((i 1 (+ i 1)))
        ((> i 1000) #t)
  (train-data dataset) (write i))


; Test mlp
; If you want to see the binary version (a 10 * 10 matrix), uncomment the following line
; (test-data  dataset-test)
(map digit-reverse (test-data  dataset-test))
```

```
92202212222232242252262272282292302312322332342352362372382392402412422432442452462472482
65266267268269270271272273274275276277278279280281282283284285286287288289290291292293942
31131231331431531631731831932032132232332432532632732832933033133233333433533633733839340
63573583593603613623633643653663673683693703713723733743753763773783793803813823833843853
02403404405406407408409410411412413414415416417418419420421422423424425426427428429430431
44844945045145245345445545645745845946046146246346446546646746846947047147247347447547647
34944954964974984995005015025035045055065075085095105115125135145155165175185195205215225
39540541542543544545546547548549550551552553554555556557558559560561562563564565566567568
58558658758858959059159259359459559659759859960060160260360460560660760860961061161261361
06316326336346356366376386396406416426436446456466476486496506516526536546556566576586596
76677678679680681682683684685686687688689690691692693694695696697698699700701702703704705
72272372472572672772872973073173273373473573673773873974074174274374474574674774874975075
77687697707717727737747757767777787797807817827837847857867877887897907917927937947957967
13814815816817818819820821822823824825826827828829830831832833834835836837838839840841842
85986086186286386486586686786886987087187287387487587687787887988088188288388488588688788
49059069079089099109119129139149159169179189199209219229239249259269279289299309319329339
50951952953954955956957958959960961962963964965966967968969970971972973974975976977978979
9969979989991000#t
'(7 n 1 n 4 1 n n n n)
> |
```

Fig. 6 Result of 1,000 times of training

Acknowledgements

I've already taken CS902 class (in C++) two years ago when I was a freshman. I take this course again because I like coding and I want to go further to some deeper areas.

So I did, indeed. In fact, this class is far beneficial to me than I expected. I think I have earned the following things:

1) Knowledge of LISP. This language is super natural and powerful, though tricky sometimes.
2) Deeper understanding of recursive and iterative functions.
3) First approach to neural network.
4) Great teamwork experience. Also friendships with friends from other majors and from different countries.
5) Practice of English. I remember Prof. Yuan asked me which country am I from. China, I said. Maybe he thought my spoken English too good. That made me laugh a lot.

I think taking this course, CS902 in LISP, is really a pleasant and unforgettable experience. And that's why I'm spending so much effort doing every homework including the final project and its report. I want to learn more because I like it. I want to be better because I like it. At the end of this report essay, I want to show my deepest gratitude to Prof. Yuan and T.A., Mr. Wang. Mr. Wang is very talented and kind-hearted. He does everything he can to help me with my problems. I haven't known any T.A. so accountable and easygoing. And thanks to Prof. Yuan, I can have this opportunity to learn so much and earn so much. So I would like to say thank you both from the bottom of my heart.

Name: Kaiwen Tan
Student ID: 516021910353
Class: F1602107
Date: June 16th, 2019

Appendix: Source Code

```
#lang racket


;;; The code is divided into
;;; Part 1 - Neuron Network Functions
;;; Part 2 - Layer Functions
;;; Part 3 - Mlp Functions
;;; Part 4 - Calculation Functions
;;; Part 5 - Testing with XOR Function
;;; Part 6 - Testing with Simple-MNIST Dataset



;;; Part 1 - Neuron Network Functions

;; Create a new neuron
(define (new-neuron)
  (let ((theta (rand-theta))
        (backward '())
        (cache #f)
        (trained #f)
        (train-sum 0))
    (lambda ([method 'activate] [arg '()])
      (cond
       ((eq? method 'backward)
        (push! (list arg (rand-weight)) backward))
       ((eq? method 'set)
        (set! cache arg))
       ((eq? method 'reset)
        (set! cache #f)
        (set! trained #f)
        (set! train-sum 0))
       ((eq? method 'sum)
        (set! train-sum (+ train-sum arg)))
       ((eq? method 'list)
        (map (lambda (el) (cadr el)) backward))
       ((eq? method 'train)
        (if (not trained)
            (set! backward (train backward (* cache (- 1 cache) train-sum)))
            (set! trained #t)))
       ((eq? method 'activate)
        (if cache
            cache
            (begin
              (set! cache (sigmoid (- (sum-weight backward) theta)))
              cache)))))))

;; Train neurons
(define (train lst err)
  (if (empty? lst)
      '()
      (let ((n (caar lst))
            (w (cadar lst)))
        (n 'sum (* err w))
        (cons (list n (+ w (* (n) err)))
              (train (cdr lst) err)))))



;;; Part 2 - Layer Functions

;; Create a new neuron layer
(define (new-layer n)
```

```
  (if (= n 0) '()
    (cons (new-neuron) (new-layer (- n 1)))))

;; Link two layers together DDD
(define (link-layers left right)
  (if (or (empty? left) (empty? right))
    '()
    (begin
      (map (lambda (e) ((car right) 'backward e)) left)
      (link-layers left (cdr right)))))

;; Set a layer of neurons (activated/inactivated)
(define (set-layer layer in)
  (if (empty? layer) '()
      (begin
        ((car layer) 'set (car in))
        (set-layer (cdr layer) (cdr in)))))

;; Reset a single layer (inactivate it)
(define (reset-layer layer)
  (if (empty? layer)
      '()
      (begin
        ((car layer) 'reset)
        (reset-layer (cdr layer)))))

;; Run 'train on each neuron in layer
(define (train-layer layer)
  (if (empty? layer)
    '()
    (begin
      ((car layer) 'train)
      (train-layer (cdr layer)))))

;; Used in 'run-mlp & 'train-mlp
(define (run-layer layer)
  (if (empty? layer) '()
      (cons ((car layer)) (run-layer (cdr layer)))))

;; Used in 'train-mlp procedure
(define (sum-layer layer out desired a)
  (if (empty? layer)
      '()
      (begin
        ((car layer) 'sum (* a (- (car desired) (car out))))
        (cons (car out)
              (sum-layer (cdr layer)
                         (cdr out)
                         (cdr desired)
                         a)))))

;; Train each layer in reversed mlp
(define (train-layers rev-mlp)
  (if (empty? rev-mlp)
      '()
      (begin
        (train-layer (car rev-mlp))
        (train-layers (cdr rev-mlp)))))



;;; Part 3 - Mlp Functions

;; Create new mlp
(define (new-mlp spec)
  (let ((mlp (map new-layer spec)))
```

```
    (link-mlp mlp)
    mlp))

;; Link up layers in an unlinked mlp
(define (link-mlp mlp)
  (if (= (length mlp) 1) '()
      (begin
        (link-layers (car mlp) (cadr mlp))
        (link-mlp (cdr mlp)))))

;; Reset each neuron in mlp (inactivate it)
(define (reset-mlp mlp)
  (if (empty? mlp)
      '()
      (begin
        (reset-layer (car mlp))
        (reset-mlp (cdr mlp)))))

;; Receive the output of mlp
(define (run-mlp mlp in)
  (set-layer (car mlp) in)
  (let ((out (run-layer (last mlp))))
    (reset-mlp mlp)
    out))

;; Train mlp
(define (train-mlp mlp in desired [a 1])
  (set-layer (car mlp) in)
  (let ((out (run-layer (last mlp))))
    (sum-layer (last mlp) out desired a)
    (train-layers (reverse mlp))
    (reset-mlp mlp)
    out))



;;; Part 4 - Calculation Functions

;; Sigmoid function
(define (sigmoid x)
  (/ (+ 1.0 (exp (- x)))))

;; Return a new random weight in (-1.2, 1.2)
(define (rand-weight)
  (- (* (random) 1.2) 0.6))

;; Return a new random threshold in (-1.2, 1.2)
(define (rand-theta)
  (- (* (random) 1.2) 0.6))

;; Define push! syntax
(define-syntax push!
  (syntax-rules ()
    ((push item place)
     (set! place (cons item place)))))

;; Sum weight
(define (sum-weight 1st)
  (if (empty? 1st) 0
      (+ (* ((caar 1st)) (cadar 1st))
         (sum-weight (cdr 1st)))))

;; Round to binary 1's and 0's
(define (round-output out)
  (map (compose inexact->exact round) out))
```

```
;;; Part 5 - Testing with XOR Function

;; Construct mlp
(define mlp (new-mlp '(2 8 1)))

;; Train mlp. Print #t if it succeeds.
(do ((i 1 (+ i 1)))
    ((> i 10000) #t)
  (train-mlp mlp '(0 0) '(0))
  (train-mlp mlp '(0 1) '(1))
  (train-mlp mlp '(1 0) '(1))
  (train-mlp mlp '(1 1) '(0)))

;; Test mlp
(run-mlp mlp '(0 0))
(run-mlp mlp '(0 1))
(run-mlp mlp '(1 0))
(run-mlp mlp '(1 1))

;; Round the testing mlp result, and it should print the following
;'(0)
;'(1)
;'(1)
;'(0)
(round-output (run-mlp mlp '(0 0)))
(round-output (run-mlp mlp '(0 1)))
(round-output (run-mlp mlp '(1 0)))
(round-output (run-mlp mlp '(1 1)))




;;; Part 6 - Testing with Simple-MNIST Dataset

;; Toolbox Functions

; Transfer a decimal digit number to a 10-element binary list (e.g. 4 -> '(0 0 0
1 0 0 0 0 0 0))
(define (digit-transfer n)
  (define (digit-iter r i)
    (define flag
      (if (= i n) 1 0))
    (if (> i 9)
     '()
     (cons flag (digit-iter r (+ i 1)))))
  (digit-iter '() 0))

; Transfer a list of ones and zeros to a digit by returning the sequence of the
first 1 (e.g. '(0 0 0 1 0 0 0 0 0 0) -> 4)
(define (digit-reverse lst)
  (define (digit-iter a-lst i)
    (if (eq? (cdr a-lst) '())
        'n
        (if (= (car a-lst) 1)
            i
            (digit-iter (cdr a-lst) (+ i 1)))))
  (digit-iter lst 0))


;; Train mlp with data in train_mnist.csv file

(require "csv-reader.rkt")
```

```
(define file-path "train_mnist.csv")
(define file-path-test "test_mnist.csv")

(define input-file (open-input-file file-path #:mode 'text ))
(define input-file-test (open-input-file file-path-test #:mode 'text ))

; Make a csv-reader
(define csv-reader (make-csv-reader #:doublequote #f
                                    #:quoting #f))

; Return a list of array lists (consisting of string data) from given input-file
(define dataset (for/list ([m (in-producer (csv-reader input-file))]
                      #:break (eof-object? m))
            m))
(define dataset-test (for/list ([m (in-producer (csv-reader input-file-test))]
                      #:break (eof-object? m))
            m))

; Create a new mlp
(define my-mlp (new-mlp '(39 40 40 10)))

; Train the mlp with dataset
(define (train-data dataset)
  ; Map dataset with train-list
  (define (train-list lst)
    ; Train the mlp one time with one list like '(5 3.13 4.21 ... 1.21)
    (train-mlp my-mlp (cdr lst) (digit-transfer (car lst))))
  (map train-list dataset))

; Test the mlp with given data
(define (test-data dataset)
  ; Map dataset with test-list
  (define (test-list lst)
    ; E.g., 3.18E+02 -> 318
    (set! lst (map (lambda(e) (string->number e 10 'read 'decimal-as-inexact))
lst))
    ; Test the mlp
    (round-output (run-mlp my-mlp (cdr lst))))
  (map test-list dataset))

;; Implement training & testing

; Set dataset to an numerical version, e.g.,
; from '("5" "3.13" "4.21" ... "1.21")
; to   '( 5   3.13   4.21  ...  1.21 )
(set! dataset
  (map (lambda(lst)
         ; E.g., 3.18E+02 -> 318
         (set! lst (map (lambda(e) (string->number e 10 'read 'decimal-as-
inexact)) lst))
         lst) dataset))

; Train mlp for many times
(do ((i 1 (+ i 1)))
        ((> i 100) #t)
  (train-data dataset) (write i))


; Test mlp with dataset
; If you want to see the binary version (a 10 * 10 matrix), uncomment the
following line
;(test-data  dataset-test)

; If you want to see the decimal version (recommended), uncomment the following
line
(map digit-reverse (test-data  dataset-test))
```