Assignment One: Neural Networks

Stephen Arnold
Department of Computer Science
George Mason University

David Freelan
Department of Computer Science
George Mason University

March 24, 2015

1 Introduction

An artificial neural network (ANN, or NN) is a statistical learning algorithm based (loosely) on the characteristics of the human brain. The NN accepts a fixed number of inputs and calculates a fixed number of numeric outputs. A typical feature of a NN is a layer of unobservable, or hidden, units. These units are called nodes or neurons. Input nodes are connected to hidden or output nodes by weight values which modify the value of each subsequent node.

This project report details our experiments in changing various Neural Network parameters and the effects those modifications had upon the ability of the Neural Network to converge to a solution. Our neural network assignment went a little beyond the default assignment. We really wanted to know: what really happens when you change alpha, and when you change the number of neurons? As a result, we ended up re-discovering a method of training, the use of a validation set.

1.1 Motives

One of the topics discussed in class was overfitting and under fitting. We want to visualize this as its happening on a real dataset. We have two variables we're interested in: number of neurons, and the value of alpha. Alpha is the learning rate of the neural net, and the neuron width is the number of neurons in the hidden layer of the network. Both of these settings should have some effect on the fitting of the dataset.

2 Procedure

Our Neural Network made use of the simple-generalization method for classification of the data sets. To train and test our classifier, the provided dataset was divided in two: half of the data for training, and half for testing. A single network was iterated through 10,000 times, and 100 networks were realized for statistical analysis. After training the neural network, the testing data was applied and the realized errors were accumulated over an entire run. All tests were done on the supplied Voting Records training set using the the Simple Generalization method discussed in the homework.

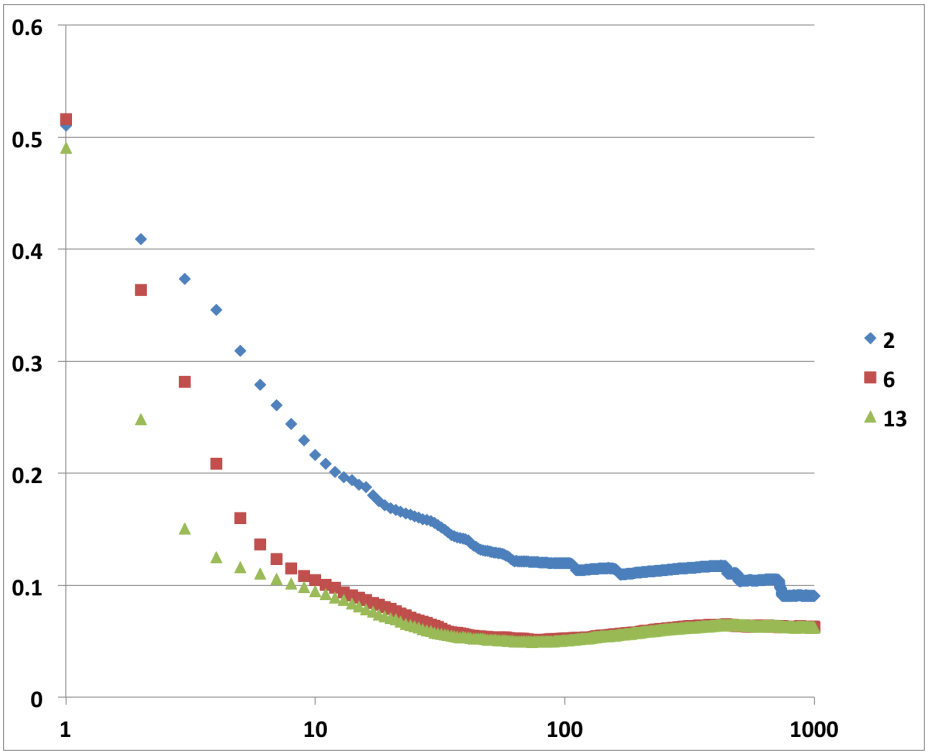

Figure 1: *This figure shows results of Nobel prize winning importance.*

3 Results

The accumulated error for three different hidden node widths may be seen in Figure 1. These three neural networks - with 2, 6 and 13 neurons - were run using a constant alpha = 0.1.

4 Conclusion

An interesting observation from the results of these experiments was the behavior of overfitting. There appears to be a period where each graph achieves a minimal error (usually within the first 1000 iterations). Following this initial minimum, the observed error becomes progressively worse. Given this insight, it is believed that the use of a validation set could verify a successful training had been achieved. Training simply train until your validation set starts to overfit, and immediately stop training. However, while we tested with a wide range of alphas and hidden neurons, we only tested it with one problem set. In the future to get a better idea of when to stop training, we would try a larger subset of problems and see if the same result happens.

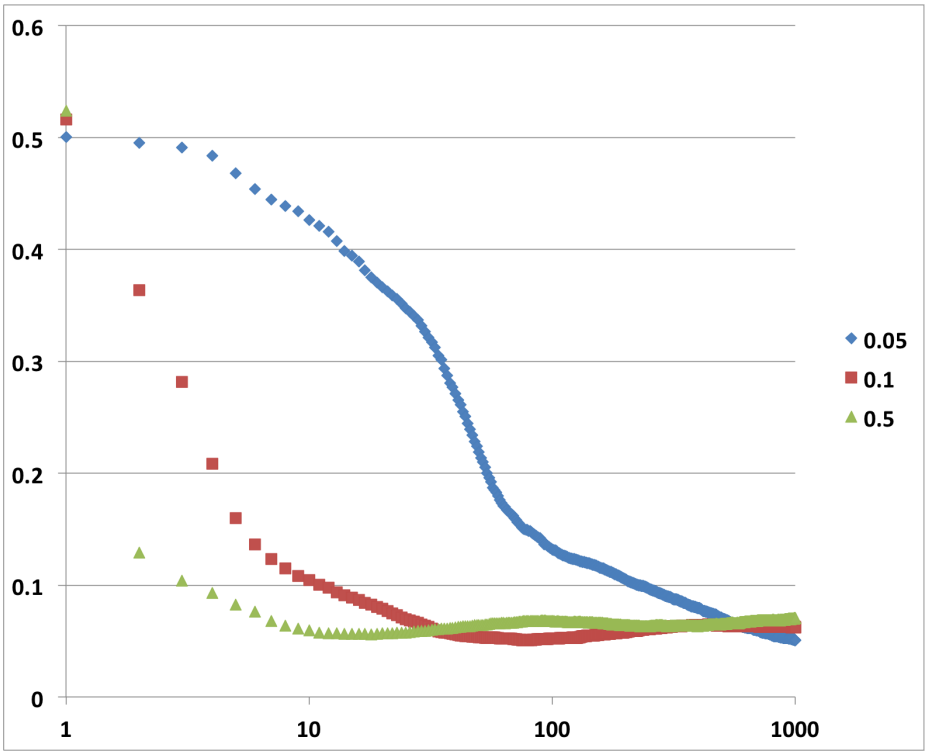

Figure 2: *This figure shows results of more Nobel prize winning importance.*

```
;;; Function Dictionary
;;
;;  *DEBUG*                    *VERIFY*                   *A-GOOD-MINIMUM-ERROR*
;;  DPRINT                     SHUFFLE                    THROW-ERROR
;;  VERIFY-EQUAL               VERIFY-MULTIPLICABLE       MAP-M
;;  TRANSPOSE                  MAKE-MATRIX                MAKE-RANDOM-MATRIX
;;  E                          PRINT-MATRIX               MULTIPLY2
;;  MULTIPLY                   ADD                        E-MULTIPLY
;;  SUBTRACT                   SCALAR-ADD                 SCALAR-MULTIPLY
;;  SUBTRACT-FROM-SCALAR       SQ                         SIGMOID
;;  NET-ERROR                  FORWARD-PROPAGATE          BACK-PROPAGATE
;;  OPTIONALLY-PRINT           INIT-NEURAL-LAYERS
    ↪ EXTRACT-INPUT-AND-OUTPUT-SIZES
;;  NET-BUILD                  SIMPLE-GENERALIZATION      FULL-DATA-TRAINING
;;  K-FOLD-VALIDATION          SCALE-LIST                 SCALE-DATUM
;;  CONVERT-DATUM              AVERAGE                    TEST-CASES
;;
;;
;; dprint (some-variable &optional (additional-message '()))
;;        "Debug Print - useful for allowing error/status messages
;;        to be printed while debug=t."
;;
```

```
;; shuffle (lis)
;;   "Shuffles a list.  Non-destructive.  O(length lis), so pretty
   ↪ efficient.
;;   Returns the shuffled version of the list."
;;
;; average (lis)
;;   "Computes the average over a list of numbers.  Returns 0 if the
   ↪ list
;;   length is 0."
;;
;; verify-multiplicable (matrix1 matrix2)
;;
;;
(setf *random-state* (make-random-state t))
(defparameter *debug* t)
(defparameter *verify* t)
(defparameter *a-good-minimum-error* 1.0e-9)

(defun dprint (some-variable &optional (additional-message '()))
        "Debug Print - useful for allowing error/status messages
to be printed while debug=t."
        (if *debug*
                (progn
                        (if additional-message (print
                           ↪ additional-message) nil)
                        (print some-variable))
                some-variable))

(defun shuffle (lis)
  "Shuffles a list.  Non-destructive.  O(length lis), so
pretty efficient.  Returns the shuffled version of the list."
  (let ((vec (apply #'vector lis)) bag (len (length lis)))
    (dotimes (x len)
      (let ((i (random (- len x))))
        (rotatef (svref vec i) (svref vec (- len x 1)))
        (push (svref vec (- len x 1)) bag)))
    bag))   ;; 65 s-expressions, by the way

;;; hmmm, openmcl keeps signalling an error of a different kind
;;; when I throw an error -- a bug in openmcl?  dunno...
(defun throw-error (str)
  (error (make-condition 'simple-error :format-control str)))

(defun verify-equal (funcname &rest matrices)
  ;; we presume they're rectangular -- else we're REALLY in trouble!
  (when *verify*
    (unless (and
             (apply #'= (mapcar #'length matrices))
```

```
            (apply #'= (mapcar #'length (mapcar #'first matrices))))
      (throw-error (format t "In ~s, matrix dimensions not equal: ~s"
                           funcname
                           (mapcar #'(lambda (mat) (list (length mat) '
                              ↪ by (length (first mat))))
                                   matrices))))))

(defun verify-multiplicable (matrix1 matrix2)
  ;; we presume they're rectangular -- else we're REALLY in trouble!
  (when *verify*
    (if (/= (length (first matrix1)) (length matrix2))
        (throw-error (format t "In multiply, matrix dimensions not
           ↪ valid: ~s"
                             (list (list (length matrix1) 'by (length (
                                ↪ first matrix1)))
                                   (list (length matrix2) 'by (length (
                                      ↪ first matrix2))))))))))


;; Basic Operations

(defun map-m (function &rest matrices)
  "Maps function over elements in matrices, returning a new matrix"
  (apply #'verify-equal 'map-m  matrices)
  (apply #'mapcar #'(lambda (&rest vectors)        ;; for each matrix...
                      (apply #'mapcar #'(lambda (&rest elts)      ;; for
                         ↪  each vector...
                                          (apply function elts))
                                      vectors))
         matrices))    ;; pretty :-)

(defun transpose (matrix)
  "Transposes a matrix"
  (apply #'mapcar #'list matrix))  ;; cool, no?

(defun make-matrix (i j func)
  "Builds a matrix with i rows and j columns,
    with each element initialized by calling (func)"
  (map-m func (make-list i :initial-element (make-list j :
     ↪ initial-element nil))))

(defun make-random-matrix (i j val)
  "Builds a matrix with i rows and j columns,
    with each element initialized to a random
    floating-point number between -val and val"
  (make-matrix i j #'(lambda (x)
                       (declare (ignore x))  ;; quiets warnings about x
                          ↪  not being used
```

```
                    (- (random (* 2.0 val)) val))))

(defun e (matrix i j)
  "Returns the element at row i and column j in matrix"
  ;; 1-based, not zero-based.  This is because it's traditional
  ;; for the top-left element in a matrix to be element (1,1),
  ;; NOT (0,0).  Sorry about that.  :-)
  (elt (elt matrix (1- i)) (1- j)))

(defun print-matrix (matrix)
  "Prints a matrix in a pleasing form, then returns matrix"
  (mapcar #'(lambda (vector) (format t "~%~{~8,4,,F~}" vector)) matrix)
    ↪ matrix)

;;; Matrix Multiplication

(defun multiply2 (matrix1 matrix2)
  "Multiplies matrix1 by matrix2
    -- don't use this, use multiply instead"
  (verify-multiplicable matrix1 matrix2)
  (let ((tmatrix2 (transpose matrix2)))
    (mapcar #'(lambda (vector1)
                (mapcar #'(lambda (vector2)
                            (apply #'+ (mapcar #'* vector1 vector2)))
                          ↪ tmatrix2))
            matrix1)))  ;; pretty :-)

(defun multiply (matrix1 matrix2 &rest matrices)
  "Multiplies matrices together"
  (reduce #'multiply2 (cons matrix1 (cons matrix2 matrices))))

;;; Element-by-element operations

(defun add (matrix1 matrix2 &rest matrices)
  "Adds matrices together, returning a new matrix"
  (apply #'verify-equal 'add matrix1 matrix2 matrices)
  (apply #'map-m #'+ matrix1 matrix2 matrices))

(defun e-multiply (matrix1 matrix2 &rest matrices)
  "Multiplies corresponding elements in matrices together,
        returning a new matrix"
  (apply #'verify-equal 'e-multiply matrix1 matrix2 matrices)
  (apply #'map-m #'* matrix1 matrix2 matrices))

(defun subtract (matrix1 matrix2 &rest matrices)
  "Subtracts matrices from the first matrix, returning a new matrix."
  (let ((all (cons matrix1 (cons matrix2 matrices))))
    (apply #'verify-equal 'subtract all)
```

```
    (apply #'map-m #'- all)))

(defun scalar-add (scalar matrix)
  "Adds scalar to each element in matrix, returning a new matrix"
  (map-m #'(lambda (elt) (+ scalar elt)) matrix))

(defun scalar-multiply (scalar matrix)
  "Multiplies each element in matrix by scalar, returning a new matrix"
  (map-m #'(lambda (elt) (* scalar elt)) matrix))

;;; This function could
;;; be done trivially with (scalar-add scalar (scalar-multiply -1
   ↪ matrix))
(defun subtract-from-scalar (scalar matrix)
  "Subtracts each element in the matrix from scalar, returning a new
     ↪ matrix"
  (map-m #'(lambda (elt) (- scalar elt)) matrix))

;;; Useful functions - spa
(defun sq (n) (* n n))


;;error = .5(tr[c-o] * (c-o))

;;; Functions you need to implement

;; IMPLEMENT THIS FUNCTION

(defun sigmoid (value)
        "Sigmoid function: 1/(1+e^value)"
        (/ 1 (+ 1 (exp (* -1 value)))))

;; output and correct-output are both column-vectors

;; IMPLEMENT THIS FUNCTION
;;  "Returns (as a scalar value) the error between the output and
   ↪ correct vectors"

(defun net-error (output correct-output)
        "Error Metric - sum of squared differences.
ERROR = (1/2)(SIGMA(correct-output - output)^2)"
        ;;(setf *debug* t)
        (let ((error (mapcar #'- (dprint correct-output "here is
           ↪ correct output") (dprint output "here is some output"))))
              ;;(print "hey i made it here")
              (* 1/2 (first (first (multiply (dprint (list error) "
                 ↪ err transpose") (dprint (transpose (list error))
                 ↪ "error")) )))
```

```
        ))

        ;;(mapcar
        ;;(* 1/2 (apply #'+ (mapcar 'sq (mapcar '- correct-output
           ↪ output)))))

;; a single datum is of the form
;; (--input-column-vector--  -- output-column-vector--)
;;
;; Notice that this is different from the raw datum provided in the
   ↪ problems below.
;; You can convert the raw datum to this column-vector form using
   ↪ CONVERT-datum

;; IMPLEMENT THIS FUNCTION
;; "Returns as a vector the output of the OUTPUT units when presented
;;the datum as input."
(defun forward-propagate (input layers)
        (dprint "forward propagate:")

        (dprint input "input:")
        (dprint layers "layers:")
        ;; this is recursive purely for the sake of being 'lispy'

        (if layers
                ;;do the multiplication of the first layer, keep popin
                   ↪ recursively until....
                (list input (forward-propagate   (map-m #'sigmoid (
                   ↪ multiply (pop layers) input)) layers))
                ;;...there are no more layers left. just return input
                   ↪ given to us, it was the multiply
                input))

;; IMPLEMENT THIS FUNCTION
;;"Back-propagates a datum through the V and W matrices,
;;returning a list consisting of new, modified V and W matrices."
  ;; Consider using let*
  ;; let* is like let, except that it lets you initialize local
  ;; variables in the context of earlier local variables in the
  ;; same let* statement.
(defun back-propagate (layer-outputs layers desired-output alpha)
  (dprint "BACK-prop desired-output:")
  (dprint desired-output)
        (if layers
                ;;
                (let ((o (second (second layer-outputs)))
                                (h (first (second layer-outputs)))
                                (i (first layer-outputs))
```

```
                                    (c desired-output)
                                    (V (first layers)) ;; V & W reversed
                                      ↪ !!! - spa
                                    (W (second layers))) ;; V & W
                                      ↪ reversed !!! - spa
                            (dprint o "o")
                            (dprint h "h")
                            (dprint i "i")
                            (dprint c "c")
                            (dprint W "W before")
                            (dprint V "V before")

                            (setf odelta (e-multiply (e-multiply (subtract
                              ↪ c o) o) (subtract-from-scalar 1 o)))
                            (dprint odelta "odelta")
                            (setf hdelta (e-multiply (e-multiply h (
                              ↪ subtract-from-scalar 1 h)) (multiply (
                              ↪ transpose W) odelta)))
                            (dprint hdelta "hdelta")
                            (setf W (add W (scalar-multiply alpha (multiply
                              ↪  odelta (transpose h)))))
                            (dprint W "W after ")
                            (setf V (add V (scalar-multiply alpha (multiply
                              ↪  hdelta (transpose i)))))
                            (dprint V "V after ")
                            (setf layers (list V W)))))  ;; V & W reversed
                              ↪ !!! - spa


;; "If option is t, then prints x, else doesn't print it.
;;In any case, returns x"
  ;;; perhaps this might be a useful function for you
(defun optionally-print (x option)

  (if option (print x) x))


;; datum is of the form:
;; (
;;  (--input-column-vector--  --output-column-vector--)
;;  (--input-column-vector--  --output-column-vector--)
;;  ...
;; )
;;
;;
;; Notice that this is different from the raw datum provided in the
  ↪ problems below.
;; You can convert the raw datum to this column-vector form using
```

```
    ↪ CONVERT-datum


;;; DAVID's helpers for net-build ;;;
(defun init-neural-layers (num-neurons input-size num-layers
    ↪ output-size initial-bounds)

        (let (layers '())
                ;; this line is really long and annoying, i have to do
                    ↪ it 3 times, there has to be a way to avoid this?
                ;;besides being long, this lines initializes the first
                    ↪ layer of nn, leading form input to hidden layers
                (dprint "info stuff:")
                (dprint num-neurons)
                (dprint input-size)
                (dprint initial-bounds)
                (dprint "layer dimensions:")
                (setf layers (append layers (list (make-random-matrix
                    ↪   (dprint num-neurons)  (dprint input-size)
                    ↪ initial-bounds))))
                (dprint ":")
                ;;does the same thing as above, but between each hidden
                    ↪  layer (!!not used for base assignment)
                (dotimes (i (- num-layers 1))
                                (setf layers (append layers (list (
                                    ↪ make-random-matrix  (dprint
                                    ↪ num-neurons)  (dprint num-neurons
                                    ↪ ) initial-bounds)))))
                (dprint ":")
                ;;creates the matrix hidden layer to output
                (setf layers (append layers (list  (make-random-matrix
                    ↪ (dprint output-size) (dprint num-neurons)
                    ↪ initial-bounds))))))


  ;;returns a list of two elements, representing input and output sizes
      ↪ . Example: nand returns (2 1)
(defun extract-input-and-output-sizes (datum)
 ;; this is parsed based on how the datum is formatted in sean's
     ↪ datumsets (AFTER convert-datum)
 (dprint "input, outputs is returning")
 (dprint (second (first datum)))
 (dprint (list (length (first  (first datum)))  (length  (second (first
     ↪  datum))))))
;;; IMPLEMENT THIS FUNCTION

;;"Builds a neural network with num-hidden-units and the appropriate
    ↪ number
;;of input and output units based on the datum.  Each element should be
```

```
   ↪  a random
;;value between -(INITIAL-BOUNDS) and +(INITIAL-BOUNDS).
;;
;;Then performs the following loop MAX-ITERATIONS times, or until the
   ↪ error condition
;;is met (see below):
;;
;;   1. For each datum element in a randomized version of the datum,
   ↪ perform
;;      backpropagation.
;;   2. Every modulo iterations,
;;         For every datum element in the datum, perform forward
   ↪ propagation and
;;         A.  If print-all-errors is true, then print the error for
   ↪ each element
;;         B.  At any rate, always print the worst error and the mean
   ↪ error
;;         C.  If the worst error is better (lower) than
   ↪ A-GOOD-MINIMUM-ERROR,
;;             quit all loops and prepare to exit the function --
;;             the error condition was met.
;;The function should return a list of two items: the final V matrix
;;and the final W matrix of the learned network."
(defun net-build (datum num-hidden-units alpha initial-bounds
   ↪ max-iterations modulo &optional print-all-errors)
      ;; use my two helper functions, extract-input-and-output-sizes
         ↪ to get appropriate sizes
      ;; use init-neural-layers to actually make the matrix
      (dprint "initial bounds is:")
      (dprint initial-bounds)
      ;;(print (extract-input-and-output-sizes datum))
      (let ((i-o-size (extract-input-and-output-sizes datum)))
            (dprint i-o-size "i-o-size is:")

            (init-neural-layers num-hidden-units (first i-o-size) 1
               ↪  (second i-o-size) initial-bounds)))

(defvar *all-errors* '())
(defvar *all-classification-errors* '())
(defun do-statistics (layers testing-data)
      (save-current-classification-error layers testing-data)
      (save-current-total-error layers testing-data))
(defun save-current-classification-error (layers testing-data)
      (let ((total-incorrect 0))
            (loop for a from 0 to  (- (length testing-data) 1)  do(
               ↪ progn
                  (let ((layer-outputs (forward-propagate (first
                     ↪  (nth a testing-data)) layers )))
```

```
                                    (setf nn-answer (first (first (second (
                                       ↪ second layer-outputs)))))
                                    (setf correct-answer (first (first (
                                       ↪ second (nth a testing-data)))))


                                    (if (> (abs (- nn-answer correct-answer
                                       ↪ )) .39)
                                          (setf total-incorrect (+
                                             ↪ total-incorrect 1))
                                          nil
                                    ))))
            (dprint (setf *all-classification-errors* (append *
               ↪ all-classification-errors* (list (float (/
               ↪ total-incorrect  (length testing-data)))))) "growing *
               ↪ all-errors* :")))
;;

(defun save-current-total-error (layers testing-data)
        (let ((total-error 0))
                (loop for a from 0 to  (- (length testing-data) 1)  do(
                   ↪ progn
                        (let ((layer-outputs (forward-propagate (first
                           ↪  (nth a testing-data)) layers )))
                             (dprint (setf total-error (+
                                ↪ total-error (net-error (first (
                                ↪ second (second layer-outputs))) (
                                ↪ first (second (nth a testing-data
                                ↪ ))))))) "intermediate total error
                                ↪ accumulating: simple-general"))))
        (dprint (setf *all-errors* (append *all-errors* (list (float (/
           ↪  total-error (length testing-data)))))) "growing *
           ↪ all-errors* :")))
;; For this function, you should pass in the datum just like it's
   ↪ defined
;; in the example problems below (that is, not in the "column vector"
   ↪ format
;; used by NET-BUILD.  Of course, if you need to call NET_BUILD from
   ↪ this function
;; you can alway convert this datum to column-vector format using
   ↪ CONVERT-datum within
;; the SIMPLE-GENERALIZATION function.
;;
;; Yes, this is ridiculously inconsistent.  Deal with it.  :-)

;;; IMPLEMENT THIS FUNCTION
;; "Given a set of datum, trains a neural network on the first half
;;of the datum, then tests generalization on the second half, returning
```

```
;;the average error among the samples in the second half.  Don't print
   ↪ any errors,
;;and use a modulo of MAX-ITERATIONS."
(defun simple-generalization (training-set testing-set num-hidden-units
   ↪  alpha initial-bounds max-iterations)
        ;;(dprint training-set "training set:")
        ;;(dprint testing-set "testing set:")
        ;;(print (forward-propagate (first (first (convert-datum *xor*)
           ↪ )) (net-build (convert-datum *xor*) 3 .2 9 90 2)))
        ;;net-build (datum num-hidden-units alpha initial-bounds
           ↪ max-iterations modulo &optional print-all-errors)
        ;;(setf *debug* t)
        (let ((total-error 0) (layers (net-build training-set
           ↪ num-hidden-units alpha initial-bounds max-iterations 1)))
             (loop for i from 1 to max-iterations do(progn
                     ;;(print i )
                     ;;(print max-iterations)
                     (setf total-error 0)
                     (shuffle training-set)
                     (dprint i "looping:")
                      (setf total-error (first (last (do-statistics
                         ↪ layers testing-set))))

                     ;;train on half the data
                      (loop for a from 0 to (- (length training-set)
                         ↪  1) do(progn
                               (let ( (layer-outputs (
                                  ↪ forward-propagate (first (nth a
                                  ↪ training-set )) layers )))
                                    (dprint (setf layers (
                                       ↪ back-propagate
                                                 (dprint
                                                    ↪ layer-outputs
                                                    ↪  "
                                                    ↪ supplied␣
                                                    ↪ layer␣
                                                    ↪ outputs␣
                                                    ↪ to␣
                                                    ↪ back-prop
                                                    ↪ :")
                                                    ↪ layers (
                                                    ↪ second (
                                                    ↪ nth a
                                                    ↪ training-set
                                                    ↪ )) alpha)
                                                    ↪ ) "
                                                    ↪ resulting
                                                    ↪ ␣layers␣
```

```
                                                    ↪ after␣
                                                    ↪ back-prop
                                                    ↪ ")))))))

                    (dprint (setf total-error (first (last (do-statistics
                       ↪ layers testing-set)))) "total␣error␣after␣testing
                       ↪ ")
                    (/ total-error (length training-set))));;doesnt mean
                       ↪ anything right now

            ;;need to get num inputs, num outputs from datum.
            ;;let layer-datum
(defun full-data-training (datum num-hidden-units alpha initial-bounds
   ↪ max-iterations)

            ;;(print (forward-propagate (first (first (convert-datum *xor*)
               ↪ )) (net-build (convert-datum *xor*) 3 .2 9 90 2)))
            ;;net-build (datum num-hidden-units alpha initial-bounds
               ↪ max-iterations modulo &optional print-all-errors)
            ;;(setf *debug* t)
            (setf path (make-pathname :name "nn-sean.dat"))
            (setf str  (open path :direction :output









            (let ((total-error 0)
                                  (layers (net-build datum
                                     ↪ num-hidden-units alpha
                                     ↪ initial-bounds max-iterations 1))
                                     ↪ )
                    (loop for i from 1 to max-iterations do(progn
                            (setf total-error 0)
                            (shuffle datum)
                             (dprint i "looping:")

                             (loop for a from 1 to (- (length datum) 1) do(
                                ↪ progn
                                    (let ((layer-outputs (
                                       ↪ forward-propagate (first (nth a
                                       ↪ (dprint datum "hey␣this␣is␣the␣
                                       ↪ dataset␣i'm␣grabbing␣the␣nth␣of:
                                       ↪ "))) layers )))
                                         ;;(dprint
```

```
                (setf layers (
                  ↪ back-propagate
                    ;;(dprint
                      layer-outputs
                          ;;"
                            ↪ supplied
                            ↪
                            ↪ layer
                            ↪
                            ↪ outputs
                            ↪
                            ↪ to
                            ↪
                            ↪ back-prop
                            ↪ :")
                            ↪
                        layers
                            ↪ (
                            ↪ second
                            ↪
                            ↪ (
                            ↪ nth
                            ↪
                            ↪ a
                            ↪
                            ↪ datum
                            ↪ )
                            ↪ )
                            ↪
                            ↪ alpha
                            ↪ )
                            ↪ )
                    ;;"resulting layers
                      ↪ after back-prop")
              (format str "~A~%" (net-error (
                ↪ first (second (second
                ↪ layer-outputs))) (first (
                ↪ second (nth a datum)))))
                ↪ ;;there's got to be a
                ↪ better way to format this
              (dprint (setf total-error (+
                ↪ total-error (net-error (
                ↪ first (second (second
                ↪ layer-outputs))) (first (
                ↪ second (nth a datum))))))
                ↪  "intermediate␣total␣
                ↪ error␣accumulating")
```

```
                                    )))))
                    (/ total-error (length datum)))
            (close str))



;;; IMPLEMENT THIS FUNCTION FOR EXTRA CREDIT
;;"Given a set of datum, performs k-fold validation on this datum for
;;the provided value of k, by training the network on (k-1)/k of the
    ↪ datum,
;;then testing generalization on the remaining 1/k of the datum.  This
    ↪ is
;;done k times for different 1/k chunks (and building k different
    ↪ networks).
;;The average error among all tested samples is returned.  Don't print
    ↪ any errors,
;;and use a modulo of MAX-ITERATIONS."
(defun k-fold-validation (data k num-hidden-units alpha initial-bounds
    ↪ max-iterations))


;;;; Some useful preprocessing functions

(defun scale-list (lis)
  "Scales a list so the minimum value is 0.1 and the maximum value is
     ↪ 0.9.  Don't use this function, it's just used by scale-datum."
  (let ((min (reduce #'min lis))
        (max (reduce #'max lis)))
    (mapcar (lambda (elt) (+ 0.1 (* 0.8 (/ (- elt min) (- max min)))))
            lis)))

(defun scale-datum (lis)
  "Scales all the attributes in a list of samples of the form ((
     ↪ attributes) (outputs))"
  (transpose (list (transpose (mapcar #'scale-list (transpose (mapcar
     ↪ #'first lis))))
                   (transpose (mapcar #'scale-list (transpose (mapcar
                      ↪ #'second lis)))))))

(defun convert-datum (raw-datum)
  "Converts raw datum into column-vector datum of the form that
can be fed into NET-LEARN.  Also adds a bias unit of 0.5 to the input."
        (dprint "RAW DATUM FOLLOWS:")
        (dprint raw-datum)
  (mapcar #'(lambda (datum)
              (mapcar #'(lambda (vec)
                          (mapcar #'list vec))
                      (list (cons 0.5 (first datum))
```

```
                                    (second datum))))
          raw-datum))

(defun average (lis)
  "Computes the average over a list of numbers.  Returns 0 if the list
     ↪ length is 0."
  (if (= (length lis) 0)
      0
      (/ (reduce #'+ lis) (length lis))))

;;; Load the Test Data from an erternal file === MUCH MORE COVENIENT
   ↪ than leaving it here!
(load "./nn-test.lisp")
(defparameter *set* *voting-records*)


(defun test-cases ()
        (let ((temp *debug*))
                ;;i dont want debug messages on for printing test cases
                (setf *debug* '())

                (print "NET-ERRROR: output should be (-3 -1 1 3)")
                (print (net-error '(1 2 3 4) '(4 3 2 1)))
                (print "NET-BUILD: output should be: ((4x3 matrix) (4*1
                   ↪  matrix)) with values between -9 and 9.  this
                   ↪ represents 4 hidden nodes, 3 inputs, and 1 output
                   ↪ ")
                (print (net-build (convert-datum *nand*) 4 .2 1 90 2))
                (print "FORWARD-PROPAGATE: should return values from
                   ↪ each layer so ((input vector) (
                   ↪ some-hidden-layer-vector) (answer-vector))")

                ;;(first (first data)) gets the first set of input. (
                   ↪ first (second data)) would get the first output
                   ↪ set
                (print (forward-propagate (first (first (convert-datum
                   ↪ *nand*))) (net-build (convert-datum *nand*) 3 .2
                   ↪ 9 90 2)))

        ;;      (print "full data training test, should print out final
           ↪  average error hopefully close to zero")
        ;;      (print (full-data-training (convert-datum *
           ↪ voting-records*) 4 .2 1 1000))

        ;;      (print "simple-general training test, should print out
           ↪ final average error hopefully close to zero")
        ;;      (print (simple-generalization (convert-datum *
           ↪ voting-records*) 4 .02 1 1000))
```

```
                ;;set the debug state to whatever it was before i set
                   ↪ it to nil
                (setf *debug* temp)))


#|
;;; some test code for you to try

;;; you'll need to run this a couple of times before it globally
   ↪ converges.
;;; When it *doesn't* converge what is usually happening?
(net-build (convert-datum *nand*) 3 1.0 5 20000 1000 t)

(net-build (convert-datum *xor*) 3 1.0 5 20000 1000 t)


;; how well does this converge on average?  Can you modify it to do
   ↪ better?
(net-build (convert-datum *voting-records*) 10 1.0 2 5000 250 t)


;;; how well does this generalize usually?  Can you modify it to
   ↪ typically generalize better?
(simple-generalization *voting-records* ...)  ;; pick appropriate
   ↪ values

;;; how well does this generalize usually?  Can you modify it to
   ↪ typically generalize better?
(simple-generalization *mpg* ...) ;; pick appropriate values

;;; how well does this generalize usually?  Can you modify it to
   ↪ typically generalize better?
(simple-generalization *wine* ...)  ;; pick appropriate values

|#


;;test cases for each function

;;main?
(setf *debug* nil)
(dprint "******** STARTING TEST CASES **********")
;;(test-cases)
(dprint "*nand* shuffle")
(shuffle *nand*)
;;;These seemed to be repeated up in test cases ...
;;(print "******** STARTING FULL-DATA-TRAINING **********")
```

```
;;(print (full-data-training (convert-datum *xor*) 4 .2 1 1))
;;(print "******** STARTING NET-BUILD **********")
;;(print (net-build (convert-datum *xor*) 4 .2 9 90 2))
;;(print "******** STARTING FORWARD-PROPOGATE **********")
;;(print (forward-propagate (dprint (first (first (convert-datum *xor*)
   ↪ )) "CONVERTED-DATA") (net-build (convert-datum *xor*) 3 .2 9 90
   ↪ 2)))
(print *all-errors*)
(print (format t "blah: ~S ~A"  2 "monkey feet"))
(print (concatenate 'string "Karl" (format nil "blah~S"  2)))
(setf *debug* nil)

(loop for neurons in '(12 13) do(progn
        (loop for alpha in '(.005 0.01 0.02 0.04 0.06 0.1 0.15 0.2 0.5)
           ↪  do(progn
                ;;(print alpha)
                (dotimes (i 50)
                        (simple-generalization (subseq (convert-datum *
                           ↪ set*) 0 (- (floor (length *set*) 2.0) 1))
                           ↪  (subseq (convert-datum *set*) (floor (
                           ↪ length *set*) 2.0) (- (length *set*) 1))
                           ↪ neurons alpha 1 1000)
                        (print "getting a file out")
                        (with-open-file (str  (print (format nil "
                           ↪ AvgErr-alpha~ANeurons~ATrial~A.txt" alpha
                           ↪  neurons i))
                                              :direction :output
                                              :if-exists :supersede
                                              :if-does-not-exist :
                                                 ↪ create)
                        (format str  "~a" *all-errors*))

                        (with-open-file (str  (print (format nil "
                           ↪ Classification-alpha~ANeurons~ATrial~A.
                           ↪ txt" alpha neurons i))
                                              :direction :output
                                              :if-exists :supersede
                                              :if-does-not-exist :
                                                 ↪ create)

                        (format str  "~a" *all-classification-errors*))

                        (setf *all-classification-errors* '())

                        (setf *all-errors* '()))
                ;;(print alpha)

        ))))
```

```
  ;;(print "simple-general training test, should print out final
     ↪ average error hopefully close to zero")
  ;;(print (simple-generalization (convert-datum *voting-records*) 4
     ↪ .02 1 1000))


(dprint "HELLO: this is the end. Goodbye.")

;;(print (full-data-training (convert-datum *xor*) 4 .2 1 1))
```