Backpropagation Algorithm in Lisp

Bibek Panthi

November 16, 2020

Contents

This tutorial was used as reference to implement Backpropagation algorithm.

```
(defpackage :backprop
  (:use :cl))

(in-package :backprop)
```

1 Forward Propagate

1.1 Neuron Activation

activation = sum (weights * inputs) + bias

```
(defun activation (weights inputs)
  (assert (= (length inputs) (1- (length weights))))
  (loop with activation = (elt weights 0)
        for x across inputs
        for i from 1
        summing (* (aref weights i) x)))
```

1.2 Neuron Transfer - Activation function

For now we use sigmod activation function. output = $\frac{1}{1+\exp(-\text{activation})}$

```
(defun transfer (activation)
  (/ (1+ (exp (- activation)))))
```

1.3 Network

Before we implement forward propagation we need a data structure to store the weights and outputs of the network

1.3.1 Weights

```
(defstruct network
  weights
  outputs
  errors)

(defun random-vector (size)
  "Create a random vector of given `size'"
  (let ((weights (make-array size :element-type 'double-float)))
    (loop for i from 0
          repeat size do
            (setf (aref weights i) (/ (random 100) 100d0)))
    weights))

(defun initialize-network-weights (num-neurons)
  "Create a randomly initialized fully connected network
     with number of neurons in each layers given by `num-neurons'
     first element of `num-neurons' = no of inputs
     last element of `num-neurons' = no of outputs'"
  (let ((network (make-array (1- (length num-neurons)))))
    ;; loop over the layers
    (loop for n in num-neurons
          for m in (rest num-neurons)
          for i from 0
          for weights-matrix = (make-array m) do
            ;; loop over the neurons in the layer
            (loop for weights = (random-vector (1+ n))
                  for i from 0 below m do
```

```lisp
                    (setf (aref weights-matrix i) weights))
            (setf (aref network i) weights-matrix))
    network))

(defun weight-vector (network i j)
  "Return the weight vector of 'j' the neuron of 'i' the layer
(first hidden layer is 0-th layer)"
  (aref (aref (network-weights network) i) j))
```

1.3.2 Output and Errors

```lisp
(defun initialize-network (num-neurons)
  (let ((weights (initialize-network-weights num-neurons)))
    (make-network :weights weights
                  :outputs (make-array (1- (length num-neurons))
                                       :initial-contents
                                       (loop for n in (rest num-neurons)
                                             collect (make-array n :element-type 'double-float)
                  :errors (make-array (1- (length num-neurons))
                                      :initial-contents
                                      (loop for n in (rest num-neurons)
                                            collect (make-array n :element-type 'double-float))
(defun output (network)
  "Output of the last layer of the network"
  (let ((outputs (network-outputs network)))
    (aref outputs (- (length outputs) 1))))
```

1.4 Forward Propagation

```lisp
(defun forward-propagate (network input)
  (loop for layer-weights across (network-weights network)
        for layer-outputs across (network-outputs network) do
          (map-into layer-outputs
                    (lambda (weights)
                      (transfer (activation weights input)))
                    layer-weights)
          (setf input layer-outputs)
        finally (return layer-outputs)))
```

1.5 Testing Forward Propagation

We create a neural network with 4 inputs a single hidden layer with 2 neurons and an output layer with 2 neurons. Its initialized with random weights and biases and the an input is feed-forwarded finally we get two output values

```lisp
(let ((network (initialize-network (list 4 2 2))))
  (forward-propagate network (vector 1 3 4 8)))
```

0.792232215073208d0   0.7556908891941516d0

2 Back Propagation Error

2.1 Derivative of transfer function

We were using sigmod activation function whose derivative is very cheaply calcuated from the output of transfer functions o as $o(1 - o)$.

```lisp
(defun transfer-derivative (output)
  (* output (- 1 output)))
```

2.2 Backpropagation

2.2.1 Theory

Loss function is defined as $L = \frac{1}{2}||\vec{o} - \text{expected}||^2$ where o is output vector i.e. outputs from the output layer

So, for the output layer the derivative of the loss function wrt the activation value at the output layer is

error = (output - expected) * $\text{transfer}_{\text{derivative}}$(output)

$$\frac{\partial L}{\partial a_i} = (o_i - \text{expected})\frac{df(a_i)}{da_i}$$

and the contribution of kth neuron of a hidden layer in the error of the output layer is given by

error = ($\text{weight}_{\text{kj}}$ * error_{j}) * $\text{transfer}_{\text{derivative}}$($\text{output}_{\text{j}}$)

this is because of the linear nature of the connection and application of chain rule.

- $\text{weight}_{\text{kj}}$ is the weight connecting kth neuron of hidden layer to jth neuron of output layer (or next hidden layer)
- error_{j} is the error from jth output neuron (or the neuron of next hidden layer)

The functional dependence of loss function on the activation of the kth neuron of the hidden layer is

- $L = L(\vec{o})$
- $o_j = f(a_j)$
- $a_j = \vec{w}.\vec{o}_{\text{previous layer}}$
- $o_{\text{previous layer},k} = f(a_k)$

and hence by chain rule

$$\frac{\partial L}{\partial a_k} = \frac{df(a_k)}{da_k}\sum_j \frac{\partial a_j}{\partial(f(a_k) = o_k)} * \frac{\partial L}{\partial a_j}$$

$$\text{error}_k = \frac{\partial L}{\partial a_k} = \frac{df(a_k)}{da_k} * \sum_j w_{jk} * \text{error}_j$$

2.2.2 Code

```lisp
(defun backpropagate-error (network expected)
  (with-slots (weights outputs errors) network
    ;; errors at output neurons
    (let ((err (aref errors (1- (length errors)))))
      (map-into err
                (lambda (o e)
                  (* (- o e)
                     (transfer-derivative o)))
                (aref outputs (1- (length outputs)))
                expected))

    ;; error at neurons in hidden layers
    ;; loop thorugh layers
    (loop for i from (- (length errors) 2) downto 0
          for err_i+1 = (aref errors (1+ i))
          for err_i = (aref errors i)
          for output_i = (aref outputs i)
          for weights_i = (aref weights i) do
```

```
            ;; loop thorugh each neuron in the layer
            (loop for o across output_i
                  for j from 0 do
                    ;; set error
                    (setf (aref err_i j)
                          (* (transfer-derivative o)
                             (loop for err across err_i+1
                                   for k from 0
                                   summing (* (aref (aref weights_i k) j)
                                              err))))))))
```

2.3 Test Backprop

```
(let ((network (initialize-network (list 4 2 2))))
  (forward-propagate network (vector 1 3 4 8))
  (backpropagate-error network (vector 1 1))
  network)
```

```
#S(NETWORK
   :WEIGHTS #(#(#(0.57d0 0.02d0 0.76d0 0.21d0 0.56d0)
                #(0.6d0 0.93d0 0.96d0 0.51d0 0.62d0))
              #(#(0.38d0 0.54d0 0.96d0) #(0.97d0 0.9d0 0.47d0)))
   :OUTPUTS #(#(0.9995096986821933d0 0.9999798038829305d0)
              #(0.8175320922581244d0 0.7973073162040141d0))
   :ERRORS #(#(-1.7235016475997057d-5 -6.262334168591013d-7)
             #(-0.027219352785169 76d0 -0.03275683215785833d0)))
```

3 Training the Network

the network is trained using stochastic gradient descent.

this involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, backpropagating the error and updating the network weights.

this part is broken down into two sections:

- update weights.
- train network.

3.1 updaing weights

we have calculated the derivative of loss function with respect to activation of each neuron and stored in the errors array.

to update the weights note that $a_j = (w_{j1}, w_{j2}, ...).(1, \text{input}_1, ...)$ So,

$$\frac{\partial L}{\partial w_{jk}} = \frac{\partial L}{\partial a_j} * input_k$$

```
(defun update-weights (network input learning-rate)
  ;; loop across layer
  (loop for weights across (network-weights network)
        for output across (network-outputs network)
        for err across (network-errors network) do
          ;; loop across neurons
          (loop for e across err
                for i from 0
                for neuron-weights across weights do
```

```
                  (loop for w across neuron-weights
                        for k from 0 do
                          (setf (aref neuron-weights k)
                                (- w (* e learning-rate
                                        (if (= k 0) 1 (aref input (1- k))))))))

          ;; input for next layer is output of current layer
          (setf input output)))
```

3.2 training

As mentioned, the network is updated using stochastic gradient descent.

This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset.

Because updates are made for each training pattern, this type of learning is called online learning. If errors were accumulated across an epoch before updating the weights, this is called batch learning or batch gradient descent.

```
(defun train-network (network data learning-rate epochs)
  (loop for epoch from 1 to epochs
        for total-error = 0d0 do
          (loop for (input expected-output) in data do
            (forward-propagate network input)
            ;; calculate error
            (incf total-error
                  (loop for output across (output network)
                        for expected across expected-output
                        summing (* 1/2 (expt (- output expected) 2))))
            (backpropagate-error network expected-output)
            (update-weights network input learning-rate))
          (format t "~&epoch=~d, ~tlearning-rate=~,3f ~terror=~,3f"
                  epoch learning-rate total-error)))
```

3.3 Testing training

Input:

x1	x2	class
2.7810836	2.550537003	0
1.465489372	2.362125076	0
3.396561688	4.400293529	0
1.38807019	1.850220317	0
3.06407232	3.005305973	0
7.627531214	2.759262235	1
5.332441248	2.088626775	1
6.922596716	1.77106367	1
8.675418651	-0.242068655	1
7.673756466	3.508563011	1

```
(defparameter *network* nil)
(let ((network (initialize-network (list 2 2 2)))
      (data (loop for (x1 x2 o) in data
                  collect (list (vector x1 x2)
                                (vector (if (= o 0) 1 0)
                                        (if (= o 0) 0 1))))))
  (train-network network data .5 20)
  (setf *network* network))
```

```
epoch=1,  learning-rate=0.500  error=2.905
epoch=2,  learning-rate=0.500  error=2.780
epoch=3,  learning-rate=0.500  error=2.668
epoch=4,  learning-rate=0.500  error=2.561
epoch=5,  learning-rate=0.500  error=2.447
epoch=6,  learning-rate=0.500  error=2.316
epoch=7,  learning-rate=0.500  error=2.165
epoch=8,  learning-rate=0.500  error=1.994
epoch=9,  learning-rate=0.500  error=1.809
epoch=10,  learning-rate=0.500  error=1.618
epoch=11,  learning-rate=0.500  error=1.432
epoch=12,  learning-rate=0.500  error=1.260
epoch=13,  learning-rate=0.500  error=1.106
epoch=14,  learning-rate=0.500  error=0.972
epoch=15,  learning-rate=0.500  error=0.856
epoch=16,  learning-rate=0.500  error=0.758
epoch=17,  learning-rate=0.500  error=0.674
epoch=18,  learning-rate=0.500  error=0.602
epoch=19,  learning-rate=0.500  error=0.541
epoch=20,  learning-rate=0.500  error=0.489
```

4 Predict

Making predictions with a trained neural network is easy enough.

We can do this by selecting the class value with the larger probability. This is also called the arg max function.

```lisp
(defun argmax (vector)
  (loop with h = (aref vector 0)
        with hi = 0
        for i from 1 below (length vector)
        for v = (aref vector i) do
          (when (> v h)
            (setf h v
                  hi i))
        finally (return hi)))

(defun predict (network input)
  (forward-propagate network input)
  (argmax (output network)))
```

4.1 Testing on previous data

```lisp
(loop for (x1 x2 e) in data do
  (format t "~&Expected: ~d ~tGot: ~d" e (predict *network* (vector x1 x2))))
```

```
Expected: 0  Got: 0
Expected: 0  Got: 0
Expected: 0  Got: 0
Expected: 0  Got: 0
Expected: 0  Got: 0
Expected: 1  Got: 1
Expected: 1  Got: 1
Expected: 1  Got: 1
Expected: 1  Got: 1
Expected: 1  Got: 1
```

5 Lets apply to real world database - Wheat Seeds Database

5.1 Download the dataset and normalize it

Info about the data is here: `http://archive.ics.uci.edu/ml/datasets/seeds`

```
curl http://archive.ics.uci.edu/ml/machine-learning-databases/00236/seeds_dataset.txt \
     > /tmp/dataset.txt
```

```
(defparameter *data* nil)
;; read data
(with-open-file (stream #p"/tmp/dataset.txt")
  (setf *data*
        (loop for input = (map 'vector
                               (lambda (col)
                                 (declare (ignore col))
                                 (read stream nil nil))
                               #(1 2 3 4 5 6 7))
              for class = (read stream nil 0)
              for output = (cond
                             ((= class 1) (vector 1 0 0))
                             ((= class 2) (vector 0 1 0))
                             ((= class 3) (vector 0 0 1)))
              until (not (aref input 0))
              collect (list input output))))

;; normalize data
(loop for col from 0 to 6
      for min = (reduce #'min *data* :key (lambda (r)
                                            (aref (first r) col)))
      for max = (reduce #'max *data* :key (lambda (r)
                                            (aref (first r) col)))
      do
        (loop for r in *data* do
          (setf (aref (first r) col) (/ (- (aref (first r) col) min)
                                        (- max min)))))
```

5.2 Train with all data

```
(defun accuracy (data network)
  "Evaluate accuracy of `network''s prediction on the `data'"
  (truncate (/ (count-if (lambda (datum)
                           (destructuring-bind (input output) datum
                             (= (predict network input)
                                (position 1 output))))
                         data)
               (length data))
            0.01))

(defparameter *network*
  (initialize-network (list 7 5 3)))

(train-network *network* *data* 0.3 500)

(accuracy *data* *network*)
```

```
94
```

94% accuracy

5.3 Split Database for k-fold cross validation; k = 5

```
(defun rand (start upper-limit)
  "returns a random integer i such that start <= i < upper-limit"
  (+ start (random (- upper-limit start))))

(defun shuffle (seq)
  "Permutes the elements of array in place"
  (let ((n (length seq)))
    (loop for i from 0 below n do
      (rotatef (elt seq  i) (elt seq (rand i n))))
    seq))

(defun split (data i j)
  "Returns test (between 'i' and 'j' index)and train data"
  (list
   (loop for d in data
         for k from 0
         when (<= i k j)
           collect d)
   (loop for d in data
         for k from 0
         unless (<= i k j)
           collect d)))
```

5.4 Evaluate Algorithm

```
(defun evaluate (data network-neurons number-folds learning-rate epochs)
  (shuffle data)
  (let ((n (truncate (length data) number-folds)))
    (print n)
    (loop repeat number-folds
          for i from 0 by n
          for (test train) = (split data i (+ i n -1))
          for network = (initialize-network network-neurons) do
            (print (list (length test) (length train)))
            (train-network network
                           train
                           learning-rate
                           epochs)
          collect (accuracy test network))))
```

Lets evaluate a single hidden layer neural network with 5 neurons in the hidden layer; taking learning-rate = 0.2 and 500 epochs. And spliting the data 5 times

```
(evaluate *data* (list 7 5 3) 5 0.3 500)
```

95 92 97 85 92

i.e. on average

```
(truncate (reduce #'+ (first r))
          (length (first r)))
```

```
92
```

92% accuracy