ASSIGNMENT 02

COMSATS University Islamabad
Sahiwal Campus

Usama Sarwar

FA17-BS(CS)-090-B

Ms. Raheela

Pattern Recognition

November 13, 2020

Table of Contents

1. Delta Rule | Gradient Descent Learning

1.1 Delta-rule Learning

- More X?Y with linear methods

1.2 Widrow-Hoff rule/delta rule

- Taking baby-steps toward an optimal solution
- The weight matrix will be changed by small amounts in an attempt to find a better answer.
- For an auto associator network, the goal is still to find W so that W.x x.
- But approach will be different
- Try a W, compute predicted x, then make small changes to W so that next time predicted x will be closer to actual x.

1.3 Delta rule

- Functions more like nonlinear parameter fitting - the goal is to exactly reproduce the output, Y, by incremental methods.
- Thus, weights will not grow without bound unless learning rate is too high.
- Learning rate is determined by modeler – it constrains the size of the weight changes.

1.4 Delta rule details

- Apply the following rule for each training row
- DWh (error)(input activations)
- DWh(target - input)inputT
- Autoassociation
- DWh(x - W.x) xT

- Heteroassociation
- DWh(y - W.x) xT

1.5 Auto associative example

- Two inputs (so, two outputs)
- 1,.5?1,.5 0,.5 ? 0,.5
- W 0,0,0,0
- Present first item
- W.x 0,0 x - W.x 1, .5 error
- .1 error xT .1,.05,.05,.025, W .1,.05,.05,.025
- Present first pair again
- W.x .125, .0625 x - W.x .875, .4375 error
- .1 error xT .0875,.04375,. 04375,.021875, so W .1875,.09375,.09375,.046875
- Continue by presenting both vectors 100 times each (remember - 1, .5, 0, .5)
- W .9609, .0632, .0648, .8953
- W.x1 .992, .512
- W.x2 .031, .448
- 200 more times
- W .999, .001, .001, .998
- W.x1 1.000, .500
- W.x2 .001, .499

1.6 Capacity of autoassociators trained with the delta rule

- How many random vectors can we theoretically store in a network of a given size?
- pmax lt N where N is the number of input units and is presumed large.
- How many of these vectors can we expect to learn?
- Most likely smaller than the number we can expect to store, but the answer is unknown in general.

1.7 Hetero associative example

- Two inputs, one output
- 1,.5?1 0,.5 ?0 .5,.7 ?1
- W 0,0
- Present first pair
- W.x 0,0 x - W.x 1, .5 error
- .1 error xT .1, .05, so W .1, .05
- Present first pair again
- W.x .1, .025 x - W.x .9, .475 error
- .1 error xT .09, .0475, so W .19, .0975
- Continue by presenting all 3 vectors 100 times each (remember - right answers are 1, 0, 1)
- W .887, .468
- Answers 1.121, .234, .771
- 200 more times
- W .897, .457

- Answers 1.125, .228, .768

1.8 Bias unit

- Definition
- An omnipresent input unit that is always on and connected via a trainable weight to all output (or hidden) units
- Functions like the intercept in regression
- As a practice, a bias unit should nearly always be included.

1.9 Delta rule and linear regression

- As specified the delta rule will find the same set of weights that linear regression (multiple or multivariate) finds.
- Differences?
- Delta rule is incremental - can model learning.
- Delta rule is incremental - not necessary to have all data up front.
- Delta rule is incremental - can have instabilities in approach toward a solution.

1.10 Delta rule and the Rescorla-Wagner rule

- The delta rule is mathematically equivalent to the Rescorla-Wagner rule offered in 1972 as a model of classical conditioning.
- DWh(target - input)inputT
- For Rescorla-Wagner, each input treated separately.
- DVAa(l - input)1 -- only applied if A is present
- DVBa(l - input)1 -- only applied if B is present
- where l is 100 if US, 0 if no US

1.11 Delta rule and linear separability

- Remember the problem with linear models and linear separability.
- Delta rule is an incremental linear model, so it can only work for linearly separable problems.

1.12 Delta rule and nonlinear regression

- However, the delta rule can be easily modified to include nonlinearities.
- Most common - output is logistic transformed (ogive/sigmoid) before applying learning algorithm
- This helps for some but not all nonlinearities
- Example helps with AND but not XOR
- 0,0 -gt 0 0,1-gt0 1,0-gt0 1,1-gt1 (can learn cleanly)
- 0,0 -gt 0 0,1-gt1 1,0-gt1 1,1-gt0 (cannot learn)

1.13 Stopping rule and cross-validation

- Potential problem - overfitting the data when too many predictors.
- One possible solution is early stopping – don’t continue to train to minimize training error but stop prematurely.
- When to stop?
- Use cross-validation to determine when.

1.14 Delta Rule - Summary

- A much stronger learning algorithm than traditional Hebbian learning.
- Requires accurate feedback on performance.
- Learning mechanism requires passing feedback backward through system.

- A powerful, incremental learning algorithm, but limited to linearly separable problems

1.15 Python Code

```
import
numpy
as  np
        import  pandas  as  pd
        import  matplotlib . pyplot  as  plt
        from  sklearn . datasets  import  load_boston
        from  sklearn . preprocessing  import  StandardScaler
        from  statsmodels . tools . tools  import  add_constant


        class  perceptron ():

            def  __init__ ( self , inp , target , epochs ):

                self . X  =  inp
                self . y  =  target
                self . n  =  inp . shape [ 0 ]
                self . p  =  inp . shape [ 1 ]

                self . weights  =  np . random . normal ( loc = 0.0 , scale = ( np .
        sqrt ( 2  /  self . p )),
                                                  size = self . p )   # glorot #
        [intercept, w1, w2, ..., wp]
                self . age  =  0.001
                self . epochs  =  epochs

            def  relu ( self , x ):
                if  x  > =  0 :
                    return  x
                else :
                    return  0

            def  derivative_relu ( self , x ):
                if  x  <  0 :
                    return  0
                elif  x  >  0 :
                    return  1

            def  fit ( self ):

                for  epoch  in  range ( self . epochs ):
                    outputs  = []
```

```
            for  i , observation  in  enumerate ( self . X ):
                pa  =  np . sum ( np . dot ( self . weights , observation . T
))
                y_pred  =  self . relu ( pa )
                error  =  self . y [ i ] -  y_pred
                # backprop
                self . weights  + =  self . eta  *  error  *  observation  # *
self.derivata_relu (pa) # delta rule

    def  pred ( self , x ):
        return  self . relu ( np . sum ( np . dot ( self . weights , x . T )))


# ====================== DATASET CONSTRUCTION ========================
====================
dataset  =  load_boston ()
df  =  pd . DataFrame ( data = dataset . Data , columns = dataset .
Feature_names )
y  =  dataset . target

chas  =  df [ 'CHAS' ]. values
df . drop ( labels = [ 'CHAS' ], axis = 1 , inplace = True )

# - standardization
scaler  =  StandardScaler ()
scaler . fit ( df . values )
X_train_stan  =  add_constant ( scaler . Transform ( df . Values ))
X_train_stan  =  np . c_ [ X_train_stan , chas ]

p  =  perceptron ( X_train_stan , y , 300 )
p . fit ()
print ( p . weights )

res  = []
y_preds  = []
for  i , el  in  enumerate ( X_train_stan ):
    y_pred  =  p . pred ( el )
    res . append ( p . y [ i ] -  y_pred )
    y_preds . append ( y_pred )

plt . scatter ( y_preds , res )
plt . title ( 'Residual plot of train samples' )
plt . xlabel ( 'y pred' )
plt . ylabel ( 'residuals' )
plt . show ()
```

2. Hebbian Learning Rule

This rule, one of the oldest and simplest, was introduced by Donald Hebb in his book *The Organization of Behavior* in 1949. It is a kind of feed-forward, unsupervised learning.

Basic Concept − This rule is based on a proposal given by Hebb, who wrote −

"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

From the above postulate, we can conclude that the connections between two neurons might be strengthened if the neurons fire at the same time and might weaken if they fire at different times.

Mathematical Formulation − According to Hebbian learning rule, following is the formula to increase the weight of connection at every time step.

$$\Delta w_{ji}(t)=\alpha x_i(t).y_j(t)\Delta wji(t)=\alpha xi(t).yj(t)$$

Here, $\Delta w_{ji}(t)\Delta wji(t)$ = increment by which the weight of connection increases at time step **t**

$\alpha\alpha$ = the positive and constant learning rate

$x_i(t)xi(t)$ = the input value from pre-synaptic neuron at time step **t**

$y_i(t)yi(t)$ = the output of pre-synaptic neuron at same time step **t**

3. Perceptron Learning Rule

This rule is an error correcting the supervised learning algorithm of single layer feedforward networks with linear activation function, introduced by Rosenblatt.

Basic Concept − As being supervised in nature, to calculate the error, there would be a comparison between the desired/target output and the actual output. If there is any difference found, then a change must be made to the weights of connection.

Mathematical Formulation − To explain its mathematical formulation, suppose we have 'n' number of finite input vectors, xnn, along with its desired/target output vector tnn, where n = 1 to N.

Now the output 'y' can be calculated, as explained earlier on the basis of the net input, and activation function being applied over that net input can be expressed as follows −

$$y=f(y_{in})=\{1,0,y_{in}>\theta y_{in}\leqslant\theta y=f(yin)=\{1,yin>\theta 0,yin\leqslant\theta$$

Where **θ** is threshold.

The updating of weight can be done in the following two cases −

Case I − when **t ≠ y**, then

$$w(new)=w(old)+txw(new)=w(old)+tx$$

Case II − when **t = y**, then

No change in weight