Perceptrons e Adaline

Gustavo Alves Pacheco*

11821ECP011

1 Introdução

Dando sequência ao tópico de Redes Neurais Artificiais, é apresentado neste trabalho o processo de treinamento de um Perceptron e um Adaline, visando encontrar os pesos e bias finais para a base de dados representada na tabela 1, abaixo.

Table 1: Base de Dados

s_1	s_2	t
1.0	1.0	1
1.1	1.5	1
2.5	1.7	-1
1.0	2.0	1
0.3	1.4	1
2.8	1.0	-1
0.8	1.5	1
2.5	0.5	-1
2.3	1.0	-1
0.5	1.1	1
1.9	1.3	-1
2.0	0.9	-1
0.5	1.8	1
2.1	0.6	-1

Os Perceptrons foram propostos por Frank Rosenblatt, um psicólogo. É um tipo de rede neural destinada a fazer classificações lineares, como será exibido posteriormente. Neste algoritmo, existe uma atualização dos pesos em caso de erro na dedução pela máquina. Então, o treinamento passa a ser um processo iterativo, no qual a rede neural retorna a resposta primeiro, e depois é ajustada de acordo com a exatidão do resultado.

*gap1512@gmail.com

O Adaline é um neurônio que apresenta um algoritmo de treinamento baseado na regra delta ou LMS (least mean square). Proposto e implementado por Bernard Widrow e Ted Hoff na Stanford University, em 1960, possui como diferencial a possibilidade de trabalhar com entradas e saídas contínuas, sendo a atualização dos pesos proporcional à diferença entre o valor desejado e o obtido. Também é um classificador linear [1].

Além disso, alguns resultados serão exibidos de forma gráfica, ou seja, será necessário implementar, também, uma função para plotagem dos dados. Finalmente, um novo conceito é introduzido, o de learning rate, o qual será abordado com maiores detalhes nas seções a seguir.

2 Objetivos

- Aprimorar o conhecimento sobre Redes Neurais Artificiais e obter experiência prática na implementação das mesmas.
- Implementar o algoritmo de treinamento de um Perceptron e de um Adaline.
- Realizar o treinamento destas redes para a tabela 1.

3 Materiais e Métodos

Para implementação da rede neural foi utilizada a linguagem de programação Common Lisp, compilando-a com o SBCL (Steel Bank Common Lisp). Como interface de desenvolvimento, foi utilizado o Emacs em Org Mode, configurado com a plataforma SLIME (The Superior Lisp Interaction Mode for Emacs) para melhor comunicação com o SBCL. Foi utilizada uma abordagem bottom-up para o desenvolvimento. O código produzido segue majoritariamente o paradigma funcional, sendo este trabalho como um todo uma obra de programação literária. Uma parte das funções já foram implementadas em Regra de Hebb.

4 Perceptron

Inicialmente, o Perceptron será implementado. Como no treinamento de um perceptron é utilizada a execução da rede neural, a função `running-single` deve estar presente em `iterative-training`. `Running-single` é definida da seguinte forma:

```lisp
(defun running-single (input weights threshold net-fn activation-fn)
  (funcall activation-fn (funcall net-fn weights input) threshold))
```

Desta forma, é possível executar uma rede neural para um único conjunto de entradas:

```lisp
;;(running-single input weights threshold net-fn activation-fn)
(running-single '(1 1 1) '(2 2 0) 0 #'net #'activation)
```

```
1
```

Como as funções `net` e `activation` ainda são as mesmas, não há necessidade de modificá-las. Já a função de `training` era chamada da seguinte maneira:

```lisp
;;(training source target weights)
(training '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1)) '(1 -1 -1 -1) '(0 0
↪  0))
```

2	2	-2

Como não é possível passar a função de ajuste dos pesos e o comportamento geral do treinamento é diferente, `training` será alterada. Primeiramente, a função de atualização dos pesos de um único par `source target` é implementada:

```lisp
(defun perceptron-update (source target output weights learning-rate)
  (if (eq output target)
      (list weights nil)
      (list (mapcar #'(lambda (weight source)
                        (+ weight (* learning-rate target source)))
                    weights source)
            t)))
```

Sendo a chamada da seguinte forma:

```lisp
;;(perceptron-update source target output weights learning-rate)
(perceptron-update '(-1 -1 1) -1 0 '(1 1 1) 1)
```

(2 2 0)	T

Esta função retorna dois valores. O primeiro corresponde ao valor atualizado dos pesos, enquanto o segundo informa se alguma alteração foi feita. Isto será útil na determinação da parada da iteração. Abaixo a implementação de `iterative-training`. Tal função é adequada para ambos os algoritmos, ou seja, consegue se ajustar tanto à execução do perceptron quanto da adaline. Para isto, algumas regras devem ser impostas.

Primeiramente a função de update precisa retornar uma lista do tipo `'(new-value change-p)`, indicando os novos valores de pesos e se houve alguma atualização nos mesmos. Além disso, deve receber o `source`, o `target`, o `output` obtido, os pesos antigos e a taxa de aprendizagem.

A segunda regra se refere à função de condição de parada, a qual deve receber o valor antigo dos pesos, o valor novo (do tipo `'(new-value change-p)`), o valor de p corrente, a tolerância da alteração de pesos, o número de ciclos atual e o máximo. O valor de p determina como está a execução daquele ciclo até o momento. Ele é quem será atualizado pela função de parada. Assim, um retorno de `true` fará o algoritmo continuar a execução.

As funções de net e de ativação devem continuar com a mesma assinatura das já implementadas. Assim, `iterative-training` é definida:

```
(defun perceptron-stop-condition (old update current-p tolerance
↪  current-cicles max-cicles)
  (declare (ignorable old tolerance current-cicles max-cicles))
  (or (second update) current-p))

(defun iterative-training (source-list target-list initial-weights
↪  threshold learning-rate tolerance max-cicles update-fn stop-fn
↪  net-fn activation-fn)
  (let (quadratic-error quadratic-error-aux)
    (labels ((rec (w p src trg cicle)
               (if (and src trg)
                   (let* ((output (running-single (car src) w threshold
                   ↪  net-fn activation-fn))
                          (target (car trg))
                          (update (funcall update-fn (car src) target
                           ↪  output w learning-rate)))
                     (push (expt (- target output) 2)
                      ↪  quadratic-error-aux)
                     (rec (first update)
                          (funcall stop-fn w update p tolerance cicle
                           ↪  max-cicles)
                          (cdr src) (cdr trg) cicle))
                   (progn
                     (push (list cicle
                                 (apply #'+ quadratic-error-aux)
                                 1)
                           quadratic-error)
                     (setf quadratic-error-aux nil)
                     (if p
                         (rec w nil source-list target-list (1+ cicle))
```

```
                        w)))))
      (values (rec initial-weights t source-list target-list 0)
              (nreverse quadratic-error)))))
```

Para a porta lógica `and`, tem-se a seguinte chamada:

```
;;(iterative-training source-list target-list initial-weights
;;                        threshold learning-rate tolerance max-cicles
;;                        update-fn stop-fn net-fn activation-fn)
(iterative-training
 '((1 1 1) (1 -1 1) (-1 1 1) (-1 -1 1)) '(1 -1 -1 -1) '(0 0 0) 0 1 0 0
 #'perceptron-update #'perceptron-stop-condition #'net #'activation)
```

1 1 -1

Com estes pesos, podemos utilizar a função `running`, para verificar a saída:

```
;;(running inputs weights threshold net-fn activation-fn)
(running '((1 1 1) (1 -1 1) (-1 1 1) (-1 -1 1)) '(1 1 -1) 0 #'net
↪  #'activation)
```

1 -1 -1 -1

Como o resultado obtido foi o mesmo da função lógica `and`, o treinamento foi bem sucedido.

Assim, realiza-se o mesmo processo para a base de dados da 1, tratada no código pela variável `tbsrc`.

```
;;(iterative-training source-list target-list initial-weights
;;                        threshold learning-rate tolerance max-cicles
;;                        update-fn stop-fn net-fn activation-fn)
(iterative-training
 tbsrc '(1 1 -1 1 1 -1 1 -1 -1 1 -1 -1 1 -1) '(0 0 0) 0 1 0 0
 #'perceptron-update #'perceptron-stop-condition #'net #'activation)
```

-2.6 2.1999998 1

Testando:

```
;;(running inputs weights threshold net-fn activation-fn)
(running tbsrc '(-2.6 2.1999998 1) 0 #'net #'activation)
```

1 1 -1 1 1 -1 1 -1 -1 1 -1 -1 1 -1

Logo, os valores de w_1, w_2 e b são respectivamente: -2.6, 2.1999998 e 1.

Para a parte de plotagem, o pacote `eazy-gnuplot` será utilizado. A função abaixo recebe um caminho de saída, uma tabela de pontos e uma fronteira do tipo `'((x_i y_i) (x_f y_f))` (a qual será convertida em uma reta) e os imprime na tela:

```
(defun scatter-plot (output table boundary)
  (with-plots (*standard-output* :debug nil)
    (gp-setup :terminal '(:pngcairo) :output output)
    (gp :set :palette '("defined (-1 'red', 1 'blue')"))
    (plot
     (lambda ()
       (loop
          for p in boundary
          do (format t "~&~{~a~^ ~}" p)))
     :title "Boundary"
     :with '(:lines))
    (plot
     (lambda ()
       (loop
          for p in table
          do (format t "~&~{~a~^ ~}" p)))
     :title "Points"
     :with '(:points :pt 7 :lc :palette)))
  output)
```

A função a seguir retorna os dois pontos necessários para traçar a fronteira de separação linear, indo de x_{min} a x_{max}.

```
(defun linear-boundary (weights threshold min max)
  (destructuring-bind (w1 w2 b) weights
    (labels ((equation (x) (/ (- threshold b (* x w1)) w2)))
      (list (list min (equation min))
            (list max (equation max))))))
```

Para a 1, utilizando os pesos encontrados, representados por `w-perceptron`:

```
;;(scatter-plot output table boundary)
;;(linear-boundary weights threshold min max)
(scatter-plot "plots/scatter-plot-perceptron.png" tb1
              (linear-boundary w-perceptron 0 0.3 2.8))
```

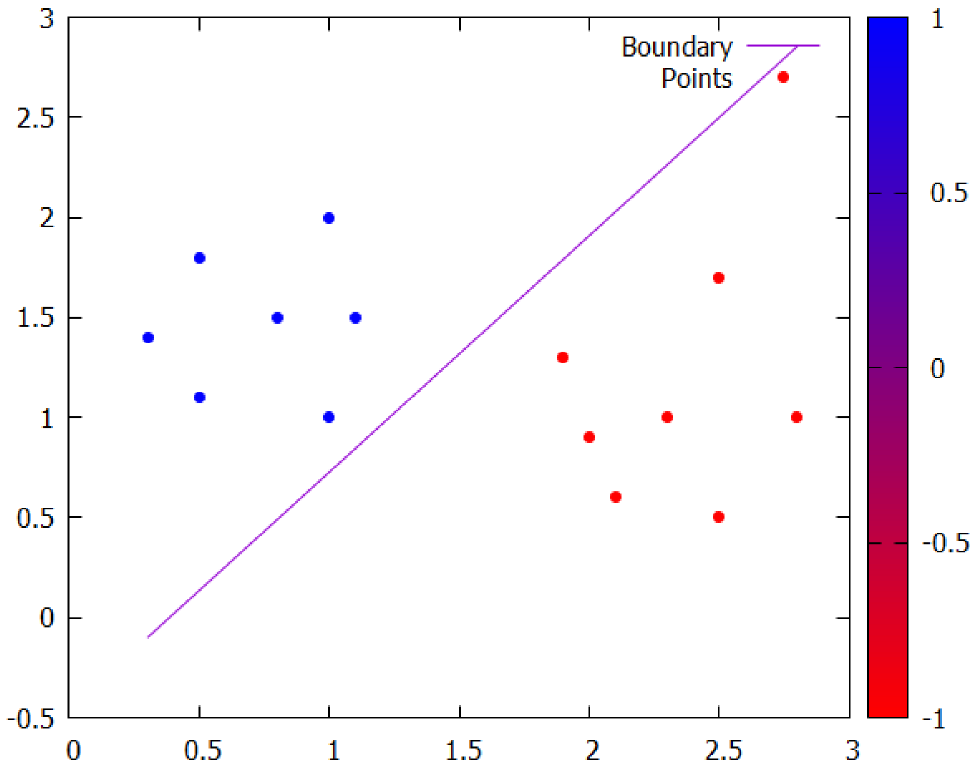

Figure 1: Perceptron

5 Adaline

Para o Adaline, é necessário uma função para inicialização aleatória dos pesos, para isto:

```
(defun random-weights (n min max)
  (let ((range (float (- max min))))
    (loop for i from 1 upto n collecting (+ min (random range)))))
```

Seguindo a mesma lógica anterior, a função de atualização dos pesos deve ser implementada (seguindo as regras colocadas):

```
(defun adaline-update (source target output weights learning-rate)
  (let ((er (- target output)))
    (list (mapcar #'(lambda (weight source)
                      (+ weight (* learning-rate er source)))
                  weights source)
          t)))
```

Além disso, a condição de parada. Vale notar que a ativação durante o treinamento deve ser uma função identidade, visto que deseja-se a saída contínua, e não a discreta (-1 ou 1).

```
(defun adaline-activation (net threshold)
  (declare (ignore threshold))
  net)

(defun adaline-stop-condition (old update current-p tolerance
↪  current-cicles max-cicles)
  (if (> current-cicles max-cicles)
      nil
      (let ((min 1))
        (mapcar #'(lambda (o-w n-w)
                    (let ((s (- n-w o-w)))
                      (when (< s min)
                        (setf min s))))
                old (first update))
        (or (> min tolerance) current-p))))
```

Assim, é necessário apenas chamar a função `iterative-training`, utilizando estas novas funções:

```lisp
;;(iterative-training source-list target-list initial-weights
;;                    threshold learning-rate tolerance max-cicles
;;                    update-fn stop-fn net-fn activation-fn)
(iterative-training
 tbsrc '(1 1 -1 1 1 -1 1 -1 -1 1 -1 -1 1 -1) (random-weights 3 -1 1)
 0 0.05 0.03 1000
 #'adaline-update #'adaline-stop-condition #'net #'adaline-activation)
```

-1.1579641	0.1814173	1.324608

Executando a rede com estes pesos (representados por `w-adaline`):

```lisp
;;(running inputs weights threshold net-fn activation-fn)
(running tbsrc w-adaline 0 #'net #'activation)
```

1	1	-1	1	1	-1	1	-1	-1	1	-1	-1	1	-1

A plotagem dos pontos de treinamento, em conjunto com a fronteira de separação é a seguinte:

```lisp
;;(scatter-plot output table boundary)
;;(linear-boundary weights threshold min max)
(scatter-plot "plots/scatter-plot-adaline.png" tb1
              (linear-boundary w-adaline 0 0.3 2.8))
```

Vale observar que devido à inicialização aleatória dos pesos, o resultado final pode apresentar variações. Entretanto, a fronteira de separação em ambos os casos é bem semelhante. O código abaixo mostra uma lista com os valores obtidos após alterações na taxa de aprendizagem, indo de 0 até 1.

```lisp
(let ((initial-weights (random-weights 3 -1 1)))
  (loop for i from 0 upto 0.5 by 0.05 collecting
       (iterative-training
        tbsrc '(1 1 -1 1 1 -1 1 -1 -1 1 -1 -1 1 -1) initial-weights 0 i
         ↪  0.03 10
        #'adaline-update #'adaline-stop-condition #'net
         ↪  #'adaline-activation)))
```

Para valores mais altos de α, o treinamento não obtém o sucesso desejado.

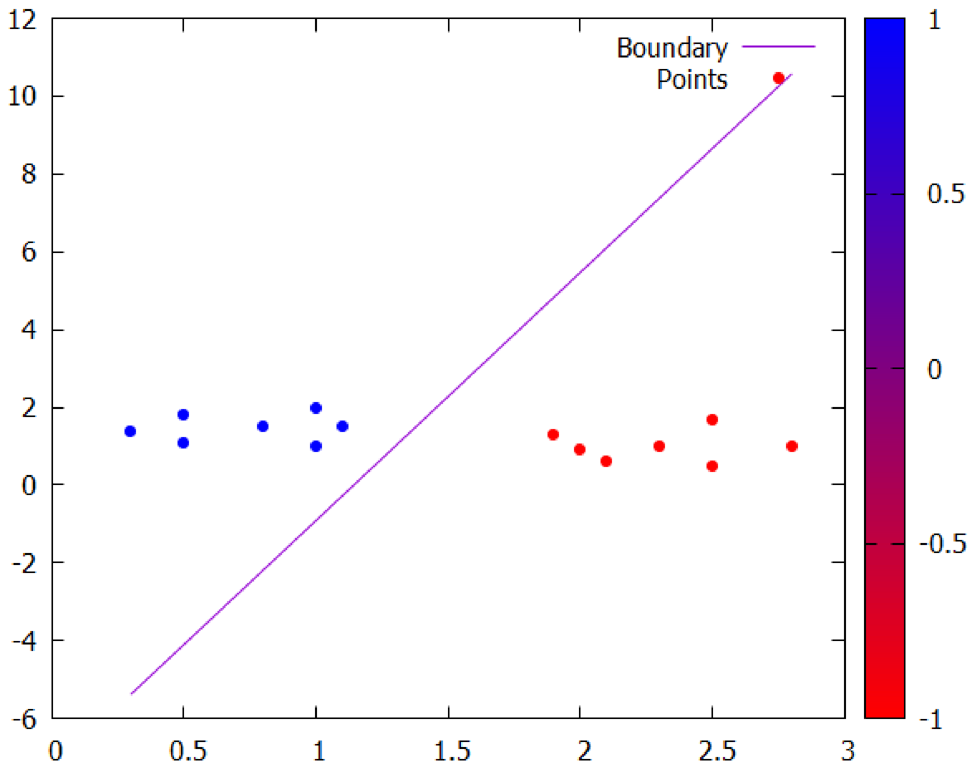

Figure 2: Adaline

Table 2: Pesos obtidos alterando apenas a taxa de aprendizagem

0.82413125	0.9934561	0.095668316
-0.8756049	0.78826696	0.19479822
-0.96819305	0.6121521	0.524767
-0.98187864	0.5292687	0.701697
-0.94292223	0.5089053	0.7110107
-0.8802576	0.50095713	0.58896744
-0.7868547	0.4735576	0.32382825
-0.29932243	0.5710494	-0.33042914
7.1326523	3.184187	-7.2279396
599845250.0	781572600.0	763032450.0

```
;;(iterative-training source-list target-list initial-weights
;;                      threshold learning-rate tolerance max-cicles
;;                      update-fn stop-fn net-fn activation-fn)
;;(scatter-plot output table boundary)
(multiple-value-bind (weights er)
    (iterative-training
     tbsrc '(1 1 -1 1 1 -1 1 -1 -1 1 -1 -1 1 -1) (random-weights 3 -1
      ↪  1) 0 0.05 0.03 1000
     #'adaline-update #'adaline-stop-condition #'net
      ↪  #'adaline-activation)
  (declare (ignorable weights))
  (scatter-plot "plots/scatter-plot-adaline-error.png" er nil))
```

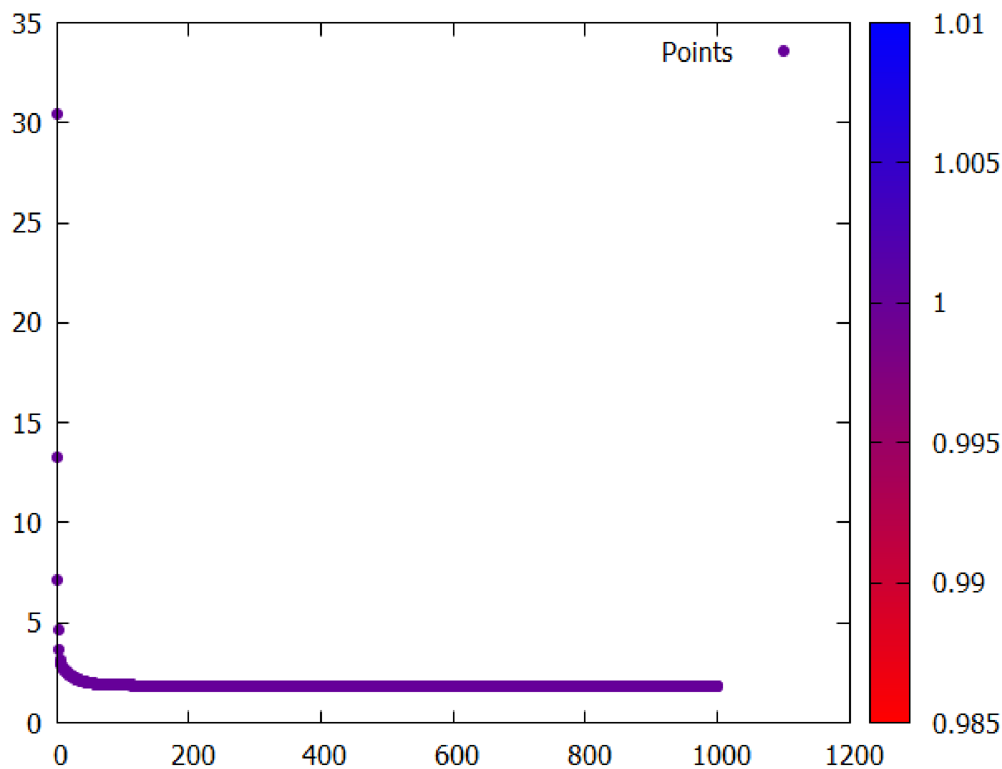

Figure 3: Erro quadrático em tempo de treinamento

6 Conclusão

Pelos resultados obtidos, comprova-se a eficácia de tais métodos para classificações lineares. A plotagem dos pontos de treinamento em conjunto com a fronteira de separação demonstra muito bem este comportamento.

Em relação à taxa de aprendizagem, tal valor apresentou uma forte influência na saída dos pesos através do treinamento por Adaline. Quando o valor de α crescia o suficiente (geralmente acima de 0.5), os resultados ficavam errados, apresentando valores exorbitantes. Além disso, os valores de pesos aleatórios conferem a cada execução um caráter único.

A plotagem da soma dos erros quadráticos de cada ciclo exibiu o comportamento desejado, convergindo bem rapidamente a um valor de tolerância.

References

[1] K. Yamanaka. Aprendizagem de máquina (machine learning - ml). Universidade Federal de Uberlândia.