Aprendizado de Funções Booleanas Linearmente Separáveis Utilizando Rede Adaline

Rafael Gonçalves Figueira

20 de Junho de 2018

Abstract

This article presents the study and development of a simple neural network for learning operations of Boolean algebra through the Adaline algorithm, which aims to demonstrate the learning ability of artificial neural networks and one of their possible applications. The results show that for linearly separable problems the Adaline neural network was able to learn quickly and arrive at the expected results, but in non-linearly separable scenarios such as "exclusive" (xor), the network was unable to be trained.

Resumo

Este artigo apresenta o estudo e desenvolvimento de uma rede neural simples para aprendizado de operações da álgebra booleana por meio de algoritmo Adaline, os quais, visa demonstrar a capacidade de aprendizado das redes neurais artificiais e uma das suas possíveis aplicações. Os resultados mostram que para problemas linearmente separáveis a rede neural Adaline conseguiu aprender rapidamente e chegar nos resultados esperados, mas em cenários não linearmente separáveis como o "ou exclusivo" (xor), a rede foi incapaz de ser treinada.

Palavras-chaves: Rede Neural, Adaline, Aprendizado de Máquina, Perceptron, Inteligência Artificial.

1 Introdução

Redes neurais artificiais são modelos computacionais inspirados no sistema nervoso de seres vivos. Possuem a capacidade de aquisição e manutenção do conhecimento (baseado em informações) e podem ser definidas por um conjunto de unidades de processamento, caracterizadas por neurônios artificiais, que são interligados por um grande número de interconexões (sinapses artificiais), sendo representadas por vetores/matrizes de pesos sinápticos. [1]

Uma RNA (Rede Neural Artificial) só pode resolver o problema para qual foi projetada, após passar por um processo de treino. Todo o desempenho das RNAs está relacionado com o processo de treino, na qual é realizado o ajuste dos pesos sinápticos de acordo com os objetivos da rede.

O primeiro modelo de rede neural (numa época em que ainda não havia a diferença atual entre neurociência computacional e redes neurais artificiais) foi proposto por Warren S. McCulloch e Walter Pitts, em um artigo publicado em 1943: "A logical calculus of the ideas immanent in nervous activity", Bulletin of Mathematical, Biophysics, 5: 115-133.

Considerada a primeira e mais primitiva estrutura de uma rede neural, o Perceptron é usado para classificar padrões linearmente separáveis (que podem ser separados por uma reta em um hiperplano), consistindo basicamente de um único neurônio com pesos sinápticos ajustáveis.

O Perceptron, embora seja uma rede simples, teve o potencial de atrair, quando de sua proposição, diversos pesquisadores que aspiravam investigar essa promissora área de pesquisa para a época, recebendo-se ainda especial atenção da comunidade científica que também trabalhava com inteligência artificial. [1]

Poucos meses após a publicação do teorema da convergência do Perceptron por Rosenblatt, os engenheiros da Universidade de Stanford Bernand Widrow e Marcian Hoff publicaram um trabalho descrevendo uma RNA muito parecida com o Perceptron, porém com as unidades de saída tendo funções de transferência lineares e com uma nova regra de aprendizado supervisionado, que ficou conhecida como regra de Widrow-Hoff (ou regra delta, ou ainda regra LMS). A RNA apresentada por eles foi batizada de Adaline (do inglês Adaptive Linear Element).

Este trabalho, em questão, procura aplicar o funcionamento da rede Adaline no aprendizado das funções booleanas *or* e *xor*.

2 Rede Neural

As redes neurais são compostas por nós ou unidades conectadas por ligações direcionadas. Uma ligação da unidade i para a unidade j serve para propagar a ativação x_i de i para j. Cada ligação também tem um peso numérico w_{ij} associado a ele, que determina a força e o sinal de conexão. Assim como em modelos de regressão linear, cada unidade tem uma entrada fictícia $x_0 = 1$ com peso associado w_{0j}. Cada unidade j primeiro calcula uma soma ponderada de suas entradas:

$$in_j = \sum_{i=0}^{n} w_{ij} x_i.$$

Em seguida, aplica uma função de ativação g a essa soma para obter a saída:

$$x_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{ij}x_i\right).$$

A ativação da função g tipicamente é tanto um limiar rígido, caso em que a unidade é chamada de perceptron, como uma função logística, caso em que por vezes é utilizado o termo perceptron sigmoide. [2]

RNAs têm capacidade computacional relacionada à aprendizagem e à generalização. Nesse sistema, o conhecimento é adquirido por um processo chamado "treinamento" ou "aprendizagem" que fica armazenado em forças de conexões entre os neurônios, chamadas pesos sinápticos. [3]

Segundo Braga et al., as RNAs são capazes de aprender através de um conjunto reduzido de exemplos e depois generalizar o conhecimento adquirido, sendo capaz de dar respostas coerentes para dados desconhecidos.

Um modelo básico de RNA possui os seguintes componentes:

- Conjunto de sinapses: conexões entre os neurônios da RNA. Cada uma delas possui um peso sináptico;
- Integrador: realiza a soma dos sinais de entrada da RNA, ponderados pelos pesos sinápticos;
- Função de ativação: restringe a amplitude do valor de saída de um neurônio;
- Bias: valor aplicado externamente a cada neurônio e tem o efeito de aumentar ou diminuir a entrada líquida da função de ativação.

Existem diversos tipos de funções de ativação, sendo que as mais populares são apresentadas a seguir [3]:

- Função limiar ou degrau: normalmente restringe a saída da RNA em valores binários [0,1] ou bipolares [-1,1]. Logo, pode ser representada por:

  $$\phi(u) = \begin{cases} 1 & \text{se } u \geq 0 \\ 0 & \text{se } u < 0 \end{cases}$$

  Um neurônio definido através dessa função de ativação é conhecido como o modelo de McCulloch-Pitts. A saída do neurônio com função limiar assume o valor de um se o potencial de ativação não é negativo e zero caso seja.

- Função sigmoide: trata-se da função mais comum. É definida como uma função crescente com balanceamento adequado entre o comportamento linear e não linear e assume um intervalo de variação entre 0 e 1 e é dada por:

  $$\phi(u) = \frac{1}{1 + exp(-au)}$$

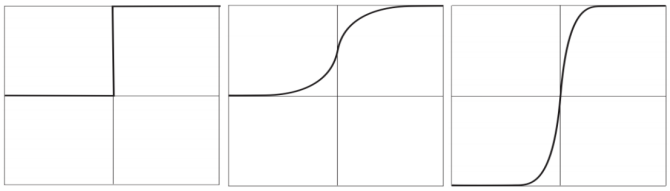
Sendo a o parâmetro da inclinação da função. Através da variação do parâmetro a são obtidas funções sigmoidais com diferentes declividades. Quando o parâmetro declividade se aproxima do infinito, a função se torna simplesmente uma função limiar.

- Função tangente hiperbólica: assume um intervalo de variação de -1 a 1 e é definida por:

$$\phi(u) = \mathrm{tg}\,(u)$$

Segundo Haykin, a característica da função tangente hiperbólica de assumir valores negativos traz benefícios analíticos.

Os comportamentos das funções de ativação estão apresentados na Figura 1.



(a) Degrau (b) Sigmoidal (c) Tangente hiperbólica

Figura 1: Ativação degrau, sigmoidal e tangente hiperbólica, respectivamente.

3 Processo de aprendizagem

O atributo com maior relevância de uma RNA é certamente a capacidade de aprender a partir dos dados de entrada que lhe são inseridos e melhorar seu desempenho através dos ajustes dos pesos sinápticos.

Durante o aprendizado, também chamado de treinamento, são realizados processos interativos (na qual cada entrada provoca uma resposta) e iterativo com um conjunto de parâmetros livres a fim de que a rede alcance o desempenho desejado. Todo o conhecimento obtido durante o aprendizado é armazenado na forma de pesos sinápticos das ligações neurais, aspecto esse que permite a rede ser replicada livremente.

No caso da rede neural Adaline, o aprendizado é feito através de um algoritmo supervisionado, sendo que para cada padrão de entrada deve existir um padrão de saída desejado. O objetivo do aprendizado é fazer com que a saída da rede seja igual ao esperado.

3.1 Regra de Widrow-Hoff

A principal característica que distingue a rede Adaline da rede Perceptron é a regra de Widrow-Hoff, muitas vezes referida como regra delta. Essa

característica de aprendizado supervisionado permite quantificar o desempenho através da função erro: se a RNA classificar corretamente todos os padrões, seu erro é 0; e quanto maior o número de classificações erradas, maior será o erro. [4]

A função erro, também chamada de custo em algumas literaturas, pode ser definida como:

$$E = E(d - \sum_i w_i a_i).$$

Sendo d a saída desejada e E a função erro.

4 Algoritmo

Para expressar graficamente o comportamento da rede Adaline sobre dados linearmente separáveis e não linearmente separáveis, foi desenvolvido um algoritmo simples para o ajustar os pesos w_i, respeitando a regra delta, dado a seguir:

1. Determina taxa de aprendizado e função de ativação definidos por r e g;
2. Lê os padrões de treino: os N padrões (x_i,d_i), onde x_i é o padrão de entrada e d_i o padrão de saída desejado;
3. Inicializa randomicamente valores entre -1 e 1 para os pesos w_i;
4. Para t = 1, 2, ..., repita os passos abaixo:
   (a) Pegue um padrão (x_i) de entrada com respectivo padrão (d_i) de saída;
   (b) Calcule a saída do neurônio: $s = g(x_i * w_i)$;
   (c) Calcule o erro: $e(t) = d(t) - s(t)$;
   (d) Modifique os pesos: $w_i = w_i + r * e(t) * x_i(t)$.

4.1 Dados linearmente separáveis

Executando o algoritmo para aprender a função booleana *or*, temos os pesos gerados aleatoriamente antes do treino:

$W_1 : -0.55601366$

$W_2 : 0.74146461$

$W_3 : -0.58656169$

E após a regra delta ser aplicada para ajustar os pesos:

W_1 : 0.64398634

W_2 : 0.74146461

W_3 : 0.61343831

Durante esse treino houveram ajustes dos pesos para minimizar o desvio entre a saída calculada pelo algoritmo e a saída desejada, sendo limitado a 100 iterações.

A Figura 2 e a Figura 3 demonstram o valor de erro em relação as iterações e os respectivos ajuste dos pesos.

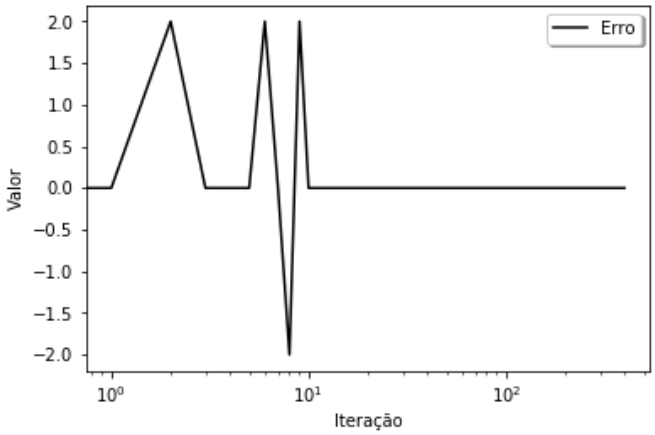


Figura 2: Erros durante treino da rede Adaline

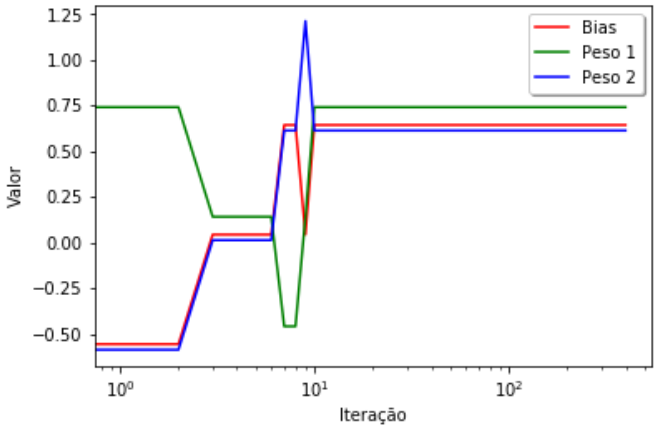


Figura 3: Ajuste dos pesos durante treino da Adaline

Através da análise dos gráficos, pode ser observado que na ausência de desvio da saída desejada com a saída calculada e erro se mantendo em zero,

não existe ajuste dos pesos, permanecendo estável até o fim das iterações delimitadas. Mostrando assim que o algoritmo conseguiu chegar em uma solução ótima para a entrada de dados linearmente separável.

4.2 Dados não linearmente separáveis

A fim de melhorar demonstrar o comportamento do algoritmo aprendendo a função booleana *xor*, foram inseridos os mesmos pesos gerados inicialmente durante treino da função *or*.

Logo temos os pesos antes do treino, sendo eles:

$W_1 : -0.55601366$

$W_2 : 0.74146461$

$W_3 : -0.58656169$

E os pesos obtidos após a última iteração:

$W_1 : -0.55601366$

$W_2 : -0.45853539$

$W_3 : -0.58656169$

Ao contrário da função *or*, durante todo treino os pesos se mantiveram instáveis, não obtendo assim uma uma solução ótima, o algoritmo continuou tentando ajustar os pesos até a última iteração.

A Figura 4 e a Figura 5 mostram o valor de erro em relação as iterações e os respectivos ajuste dos pesos.

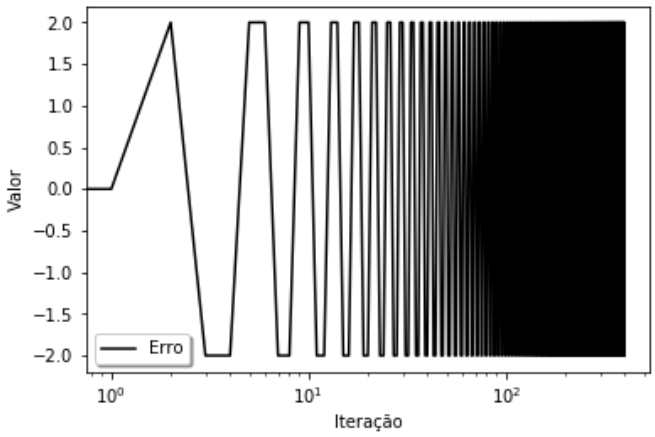


Figura 4: Erros durante treino da rede Adaline

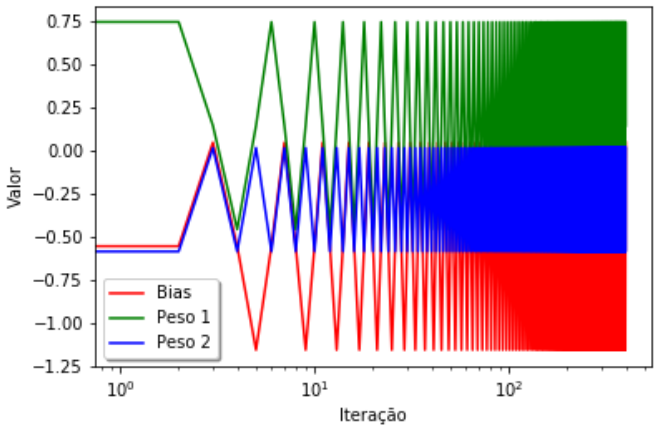


Figura 5: Ajuste dos pesos durante treino da Adaline

É possível observar nos gráficos a incapacidade da rede Adaline em aprender a função *xor*, uma vez que os dados de entrada não são linearmente separáveis. A rede se mantém ajustando os pesos enquanto sempre há desvio da saída desejada com a saída calculada.

5 Conclusão

Este presente trabalho teve como objetivo demonstrar o funcionamento da rede neural Adaline através de um algoritmo simples de aprendizado de máquina para aprender duas funções booleanas, sendo uma com um conjunto de dados linearmente separável (operação *or*) e outra não (operação *xor*).

Foi possível observar a velocidade da RNA em aprender a função *or*, uma vez que ela seja linearmente separável. todavia, uma vez que a função *xor* não seja linearmente separável, a rede foi incapaz de aprender. Existem N métodos de simular a função *xor*, muitas delas sem precisar utilizar algoritmos de aprendizado de máquina, entretanto, não foi o objetivo desse artigo abordar outros métodos.

O artigo não tem intenção de aprofundar conceitos sobre redes neurais artificiais ou demonstrar outros algoritmos de aprendizado de máquina, apenas demonstrar de forma prática e simples o funcionamento de uma rede neural Adaline através de exemplos mais didáticos, como o exemplo das operações da álgebra booleana.

Estima-se que esse projeto desperte interesse em entusiastas na área de inteligência computacional à buscarem mais conhecimento sobre o tema supracitado, em vista que é uma área multidisciplinar, tendo diversas aplicações comprovadas e muitas que ainda precisarão ser exploradas.

Referências

[1] Ivan Nunes da Silva. *Redes Neurais Artificiais Para Engenharia e Ciencias Aplicadas: Fundamentos Teoricos e Aspectos Praticos*. ARTLIBER, 2016.

[2] Stuart Russell. *Inteligência Artificial*. Elsevier, 2004.

[3] Simon Haykin. *Redes Neurais: Princípios e Prática*. Bookman, 2017.

[4] Antônio Roque. Psicologia conexionista, 2012.

[5] Fernando César C. de Castro e Maria Cristina F. de Castro. *O Perceptron*.

[6] José R. Campos. Implementação de redes neurais artificiais utilizando a linguagem de programação java.

Código do Algoritmo em Python

```
1   # -*- coding: utf-8 -*-
2   """
3   Título: Rede neural simples - Adaline
4   Autor: Rafael Goncalves
5   Data: 25/04/2018
6   """
7   import numpy as np
8   import matplotlib.pyplot as plt
9
10  # Taxa de aprendizado para o treino
11  taxaAprendizado = 0.3
12
13  # Seed dos números aleatórios para cálculos
        deterministicos
14  np.random.seed(5)
15
16  # Listas usadas para armazenar os erros e os pesos
17  erros=[]
18  bias =[]
19  peso1=[]
20  peso2=[]
21
22  # Função degrau
23  def step(x):
24   if (x > 0):
25    return 1
26   return -1
27
28  # Primeiro exemplo. OR, linearmente separável
29  # Dados de entrada representando o operador logico
        OR (com o BIAS fixo de 1)
30  entradas = np.array([[1,-1,-1],
31                       [1, 1,-1],
32                       [1,-1, 1],
33                       [1, 1, 1]])
34
35  # Saída dos dados. Resulta 1 se uma das duas
        entradas for 1
36  saidas = np.array([[-1,
37                       1,
38                       1,
39                       1]]).T
40
41
42  # Segundo exemplo. XOR, não linearmente separável
43  # Dados de entrada representando o operador logico
```

```
        XOR (com o BIAS fixo de 1)
44      entradas = np.array([[1,-1,-1],
45                           [1, 1,-1],
46                           [1,-1, 1],
47                           [1, 1, 1]])
48
49      # Saída dos dados. Resulta 1 se as entradas forem
            diferentes.
50      saidas = np.array([[-1,
51                           1,
52                           1,
53                          -1]]).T
54
55
56      # Inicializa os pesos aleatoriamente com média 0
57      pesos = 2 * np.random.random((3,1)) - 1
58      print ("\nPesos aleatórios antes do treino: \n",
            pesos)
59      # Loop de treino
60      for i in range(100):
61
62       for entrada,saidaDesejada in zip(entradas, saidas):
63
64         # Alimenta (feedforward) e calcula o somatório da
               Adaline
65        somatorio = (entrada[0]*pesos[0]) +
             (entrada[1]*pesos[1]) + (entrada[2]*pesos[2])
66
67        # Processa a saída atraves da função degrau
68        saidaAdaline = step(somatorio)
69
70        # Calcula o erro gerado
71        erro = saidaDesejada - saidaAdaline
72
73        # Armazena os erros e os pesos
74        erros.append(erro)
75        bias.append (pesos[0][0])
76        peso1.append(pesos[1][0])
77        peso2.append(pesos[2][0])
78
79        # Atualiza os pesos de acordo com a regra do Delta
80        pesos[0] = pesos[0] + taxaAprendizado * erro *
             entrada[0]
81        pesos[1] = pesos[1] + taxaAprendizado * erro *
             entrada[1]
82        pesos[2] = pesos[2] + taxaAprendizado * erro *
             entrada[2]
83
84      print ("\nNovos pesos após o treino: \n", pesos,
```

```
        "\n")
85      for entrada,saidaDesejada in zip(entradas, saidas):
86
87       # Alimenta a entrada para frente (feedforward) e
             calcula a saída da Adaline
88       somatorio = (entrada[0]*pesos[0]) +
             (entrada[1]*pesos[1]) + (entrada[2]*pesos[2])
89
90       # Processa a saída atraves da função degrau
91       saidaAdaline = step(somatorio)
92
93       print ("Saída calculada: ", saidaAdaline, "  Saída
             desejada: ", saidaDesejada)
94
95      # Plota os erros durante o treinamento
96      ax = plt.subplot(111)
97      ax.set_xscale("log")
98      #ax.set_ylim([-2,2])
99      plt.plot(erros,'#000000')
100     plt.legend(('Erro',),shadow=True)
101     #plt.title("Erros durante treino da Adaline")
102     plt.xlabel('Iteração')
103     plt.ylabel('Valor')
104     plt.show()
105
106     # Plota as variações dos pesos durante o treino
107     ax = plt.subplot(111)
108     ax.set_xscale("log")
109     #ax.plot(erros, c='#000000', label='Erro', alpha=0.3)
110     plt.plot(bias,'r',peso1,'g',peso2,'b')
111     plt.legend(('Bias','Peso 1', 'Peso 2'),shadow=True)
112     #plt.title("Ajuste dos pesos durante treino da
            Adaline")
113     plt.xlabel('Iteração')
114     plt.ylabel('Valor')
115     plt.show()
```