Redes Multicamadas

Gustavo Alves Pacheco*

11821ECP011

1 Introdução

Até o momento, foram estudadas redes de uma única camada. Tais redes apresentam uma restrição crítica: Não são adequadas para resolver problemas que não sejam linearmente separáveis. Assim, buscaram-se alternativas que removam esta restrição, e em 1969, Minsy e Papert demonstram que a adição de uma camada à essa rede neural possibilita a resolução de problemas não linearmente separáveis.

Entretanto, somente em 1986, Rumelhart, Hilton e Willian resolveriam o problema de ajuste de pesos da entrada para a camada escondida, que impossibilitava a implementação de tais redes. Assim, a solução proposta foi a de retropropagação do erro da saída.

Nesta estratégia, é utilizada a generalização da regra delta para funções de ativação não-linares. Assim, tanto a tangente hiperbólica 1 quanto as sigmoides binaria 2 ou bipolar 4 podem ser utilizadas como função de ativação. O algoritmo de aprendizado se baseia na minimização do erro quadrático 6 (dos neurônios de saída) pelo método do gradiente descendente 7 [1]. Durante a implementação de tal rede neural, mais detalhes serão abordados.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1}$$

$$f_1(x) = \frac{1}{1 + \exp(-x)} \tag{2}$$

$$f'_1(x) = f_1(x)[1 - f_1(x)] \tag{3}$$

$$f_2(x) = \frac{2}{1 + \exp(-x)} - 1 \tag{4}$$

$$f'_1(x) = \frac{1}{2}[1 + f_2(x)][1 - f_2(x)] \tag{5}$$

*gap1512@gmail.com

$$E = \frac{1}{2} \sum_{k \in K} (a_k - t_k)^2 \tag{6}$$

$$w_{jk_{novo}} \leftarrow w_{jk_{antigo}} - \eta \frac{\partial E}{\partial w_{jk_{antigo}}} \tag{7}$$

2 Objetivos

- Aprimorar o conhecimento sobre Redes Neurais Artificiais e obter experiência prática na implementação das mesmas.
- Implementar uma rede neural multicamadas utilizando o algoritmo da retropropagação do erro para a função lógica XOR (tab. 1).
- Desenhar a arquitetura da rede neural
- Plotar a curva do erro quadrático
- Apresentar os pesos encontrados

Table 1: Função Lógica XOR

X_1	X_2	Y
1	1	-1
1	-1	1
-1	1	1
-1	-1	-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. Parte das funções já foram implementadas em Regra de Hebb, Perceptron e Adaline e Regressão Linear.

4 Desenvolvimento

Apesar de possuir similaridades com a função `iterative-training` implementada anteriormente, o treinamento com retropropagação do erro apresenta diferenças que justificam a implementação de uma nova funcionalidade. Uma rede neural multicamadas será representada como uma lista de camadas. Cada camada será representada por uma lista de neurônios e cada neurônio, por uma lista de pesos, que correspondem às ligações do mesmo com a camada anterior. Assim, a função de treinamento deve receber entre os parâmetros, uma configuração da rede neural. Isto se dá na forma da lista de pesos iniciais. A mesma deverá retornar a lista de pesos atualizada.

Primeiramente, é necessário implementar a estratégia do feedforward, ou seja, dado um conjunto de entradas, determinar a saída da rede neural. A partir da estratégia bottom-up, primeiro implementa-se a saída de um único neurônio, depois de uma camada e por último, da rede.

```
(defun neuron-output (inputs neuron-list fn)
  (let ((net (reduce #'+ (mapcar #'* inputs neuron-list))))
    (values (funcall fn net)
            net)))

(defun layer-output (inputs layer-list fn)
  (mapcar #'(lambda (neuron)
              (neuron-output inputs neuron fn))
          layer-list))

(defun mlnn-output (inputs mlnn-list fn)
  (do ((result inputs (layer-output result (car layer-list) fn))
       (layer-list mlnn-list (cdr layer-list)))
      ((not layer-list) (car result))))
```

Como será utilizada a função de ativação sigmoide bipolar (4), a mesma deve ser implementada (junto de sua derivada 5):

```
(defun bipolar-sigmoid (x)
  (- (/ 2 (+ 1 (exp (- x)))) 1))

(defun bipolar-sigmoid^1 (x)
  (let ((f (bipolar-sigmoid x)))
    (* 1/2 (1+ f) (- 1 f))))
```

Assim, `mlnn-output` (que faz o papel do feedforward) é chamada da seguinte forma:

```
(mlnn-output '(-1 -1 1)
             '(((2 2 2) (3 3 3))
               ((1 2) (2 1) (3 1))
               ((2 4 5)))
             #'bipolar-sigmoid)
```

```
-0.999874
```

Algumas observações importantes:

1. É considerado nesta implementação que o último elemento da lista de pesos é sempre correspondente ao bias, portanto em todas as listas de entrada é estritamente necessário a inclusão de um valor unitário na última posição.

2. Na chamada acima, a lista de pesos passada para a função corresponde a uma rede neural com duas camadas escondidas, sendo 3 entradas na rede neural, 2 neurônios na primeira camada escondida, três na segunda e a apenas um na última.

3. A saída (única) de um neurônio alimenta todos os neurônios da próxima camada.

4. A função `neuron-output` retorna dois valores: `y_k` e `net`. Isto é necessário para o cálculo de δ.

5. `mlnn-output` assume que exista apenas um neurônio na última camada, ignorando as saídas dos outros neurônios.

Prosseguindo a implementação da função de treinamento, começa-se agora a atualização dos pesos. As funções referentes à camada de saída possuem em seu nome, o indicador k, enquanto as outras recebem o indicador j. Observa-se que não há diferença entre o algoritmo de atualização das camadas, mas sim entre o cálculo da lista de δ's. As equações estão representadas abaixo primeiro em forma matemática e depois em código `lisp`.

$$\delta_k = f'(y_{in_k})(t_k - y_k)$$

```
(defun small-delta-k (target net output fn^1)
  (* (- target output) (funcall fn^1 net)))
```

$$\delta_j = f'(z_{in_j}) \sum_{k=1}^{m} \delta_k w_{jk}$$

```
(defun small-delta-j (net delta-list weight-list fn^1)
  (* (reduce #'+ (mapcar #'* delta-list weight-list)) (funcall fn^1
  ↪  net)))
```

$$\Delta w = \alpha \delta z$$

```
(defun new-weight (learning-rate small-delta input)
  (* learning-rate small-delta input))
```

Assim, temos que a atualização de uma lista de pesos ocorre da seguinte forma:

```
(defun new-weight-list (old-weight-list inputs learning-rate
↪  small-delta)
  (mapcar #'(lambda (old-weight input)
              (+ old-weight (new-weight learning-rate small-delta
              ↪  input)))
          old-weight-list inputs))
```

E finalmente uma camada é atualizada:

```
(defun new-layer-list (old-layer-list small-delta-list inputs
↪  learning-rate)
  (mapcar #'(lambda (old-weight-list small-delta)
              (new-weight-list old-weight-list inputs learning-rate
              ↪  small-delta))
          old-layer-list small-delta-list))
```

Assim, falta implementar apenas a função `new-mlnn-list`, que atualiza toda a estrutura da rede neural. Esta função é recursiva, para que o retorno da última camada seja usado pelas camadas anteriores. Assim, é utilizado a `stack` da recursão para realizar tanto o *feedforwarding* quanto o *retropropagation*.

```
(defun new-mlnn-list (old-mlnn-list source target fn fn^1
↪  learning-rate)
  (labels ((loop-delta (old-layer-list next-layer-list inputs
  ↪  delta-list)
             (do ((i 0 (1+ i))
                  (neuron old-layer-list (cdr neuron))
```

```
          result)
         ((not neuron) (nreverse result))
       (let ((weights (mapcar #'(lambda (y) (nth i y))
                              next-layer-list)))
         (push (small-delta-j (neuron-output inputs (car
         ↪  neuron) fn)
                              delta-list weights fn^1)
               result))))
(rec (layers inputs)
  (let ((x (cdr layers)))
    (if x
        (let ((old-layer-list (car layers)))
          (multiple-value-bind (next-layers delta
          ↪  quadratic-error)
              (rec x (layer-output inputs old-layer-list
              ↪  fn))
            (let ((delta-j-list (loop-delta old-layer-list
                                            (car
                                            ↪  next-layers)
                                            inputs
                                            delta)))
              (values (cons (new-layer-list old-layer-list
                                            delta-j-list
                                            inputs
                                            learning-rate)
                            next-layers)
                      delta-j-list quadratic-error))))
        (let* ((quadratic-error 0)
               (last-layer (car layers))
               (delta-k-list
                (mapcar #'(lambda (neuron)
                            (multiple-value-bind (output
                            ↪  net)
                                (neuron-output inputs neuron
                                ↪  fn)
                              (setf quadratic-error
                                    (+ quadratic-error
                                       (expt (- output
                                       ↪  target)
                                             2)))
                              (small-delta-k target net
                              ↪  output fn^1)))
                        last-layer)))
```

```
                    (values (list (new-layer-list last-layer
                                                  delta-k-list
                                                  inputs
                                                  learning-rate))
                                  delta-k-list
                                  (/ quadratic-error 2)))))))
    (rec old-mlnn-list source)))
```

Apesar de inicialmente parecer críptica, a função `new-mlnn-list` é bem direta. É composta da definição de uma função `loop-delta`, que serve para mapear os δ's (a partir de uma lista de δ's) às entradas correspondentes, e a recursão em si. Dentro da recursão, é verificado se o `cdr` de inputs é vazio. Caso não seja, a recursão é invocada, antes que outras operações sejam feitas (garantindo, assim, que a próxima camada seja conhecida pelo escopo atual) e em seguida, a atualização é realizada. Caso `cdr` seja vazio, a função está na última camada, e deve atualizar os pesos conforme as regras anteriores. Visto que é necessário conhecer a lista de δ's da camada seguinte para atualização da camada atual, faz-se com que a recursão retorne múltiplos valores, o primeiro sendo o acumulador da lista de resultado e o segundo, a lista de δ's.

Fazendo com que a função seja aplicada a um conjunto de entradas, tem-se:

```
(defun multiple-source-new-mlnn-list (initial-mlnn-list source-list
↪  target-list fn fn^1 learning-rate)
  (do ((mlnn-list initial-mlnn-list)
       delta
       (err 0)
       (source source-list (cdr source))
       (target target-list (cdr target)))
      ((or (not source) (not target)) (values mlnn-list err))
    (setf (values mlnn-list delta err)
          (new-mlnn-list mlnn-list (car source) (car target) fn fn^1
          ↪  learning-rate))))
```

E, finalmente, o loop pelo número de ciclos (ou tolerância):

```
(defun iterative-retropropagation (initial-mlnn-list source-list
↪  target-list fn fn^1 learning-rate cycles tolerance)
  (do ((i 0 (1+ i))
       (mlnn-list initial-mlnn-list)
       (err 0)
       err-list)
      ((or (> i cycles)
```

```
          (and (< err tolerance)
               (> i 0)))
      (values mlnn-list
              (nreverse err-list)))
    (setf (values mlnn-list err)
          (multiple-source-new-mlnn-list mlnn-list source-list
          ↪  target-list fn fn^1 learning-rate))
    (push (list i err 1) err-list)))
```

Para que o treinamento seja invocado, é necessário passar como parâmetro uma lista inicial de pesos. A geração desta lista é feita aleatoriamente pelas funções a seguir:

```
(defun random-layer-list (min max n-neurons n-weights)
  (loop repeat n-neurons
        collecting (random-weights n-weights min max)))

(defun random-mlnn-list (min max n-inputs &rest configs)
  (do ((layers configs (cdr layers))
       (last-n n-inputs (car layers))
       result)
      ((not layers) (nreverse result))
    (push (random-layer-list min max (car layers) last-n)
          result)))
```

5 Testes E Resultados

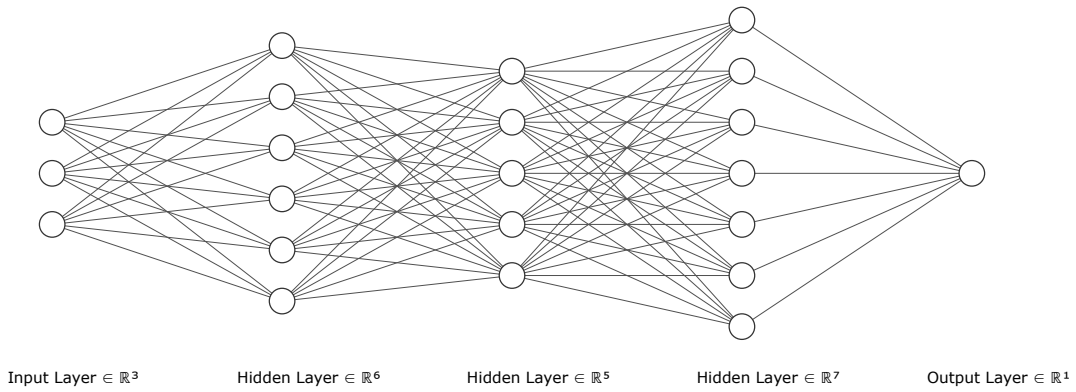

Figure 1: Arquitetura da rede neural

A arquitetura da rede neural utilizada para o teste é composta de 5 camadas, conforme ilustrado na figura 1, feita utilizando a plataforma desenvolvida por Alexander

Lenail. Esta configuração se mostrou bastante consistente no cálculo dos pesos, quase sempre encerrando a execução antes do limite de ciclos (de 10000), alcançando o valor de tolerância para o erro (0.0001). Assim, para a tabela 1, temos que os pesos são:

```
(defvar *w-mlnn*
  (multiple-value-bind (weights err)
    (iterative-retropropagation
     (random-mlnn-list -0.5 0.5 3 6 5 7 1)
     tb1-inputs
     tb1-outputs
     #'bipolar-sigmoid
     #'bipolar-sigmoid^1
     0.1 10000 0.0001)
  (scatter-plot "plots/quadratic-error.png" err nil)
  weights))
```

```
(((0.4869996 -0.6817667 -1.4005007) (0.074305326 -0.483169 -0.40500578)
  (0.023479043 -0.4342643 -0.6534313) (1.2541872 1.7165424 0.8937489)
  (1.6773621 1.2063798 -0.8329718) (-0.37802422 0.790806 1.0868403))
 ((0.8914958 0.0036922265 0.73515767 1.5324852 -1.2370629 -0.4898624)
  (-0.5903269 -0.6321171 0.17624325 -1.2522582 1.2760658 0.16868031)
  (-0.11736977 0.22771285 0.3662537 -1.3631465 0.25780293 0.24779646)
  (-1.047311 -0.06770132 -0.10483085 -1.9215018 2.0045712 0.5493633)
  (-0.57138854 0.04254329 -0.2203835 -1.3203738 1.1855516 0.97648937))
 ((0.8070346 -0.19385321 -0.57281303 -0.8588553 -0.6553931)
  (0.94319886 -0.96333796 -0.5291039 -1.581342 -0.67157316)
  (0.87945515 -0.59751284 -1.0149999 -1.3152233 -1.0888902)
  (1.1373106 -0.5224316 -0.320092 -0.9939536 -0.8736586)
  (0.4807968 -0.39199415 0.12299164 -0.0074652154 -0.78108734)
  (-0.7202096 0.92851317 0.13649674 1.0095271 0.44452295)
  (0.6271453 -0.9651741 -0.3079915 -1.2298515 -0.6896953))
 ((1.3987603 2.216685 2.1773589 1.7749593 0.7688803 -1.5186741 1.7302262)))
```

É interessante notar que o padrão observado na imagem 2 (gerada pela chamada acima) se repetiu em algumas execuções do algoritmo. Outros padrões também foram encontrados, e em algumas vezes, o resultado não foi alcançado. Percebe-se que a configuração inicial é bem significativa para a obtenção de um resultado fidedigno. Os pesos acima encontrados podem ser interpretados da seguinte forma: O primeiro nível da lista representa as camadas, portanto contém 4 elementos (a primeira camada corresponde às entradas da rede neural, portanto não aparece na lista de pesos). Dentro de cada elemento, é possível encontrar os pesos para cada neurônio. Assim, tomando como exemplo o item $(-0.347899140.7749249 - 0.8395959)$ (primeiro elemento da primeira

camada), interpreta-se que -0.34789914 é o peso para a primeira entrada, 0.7749249 para a segunda e -0.8395959 para a terceira (b, nesse caso), do primeiro neurônio da primeira camada escondida.

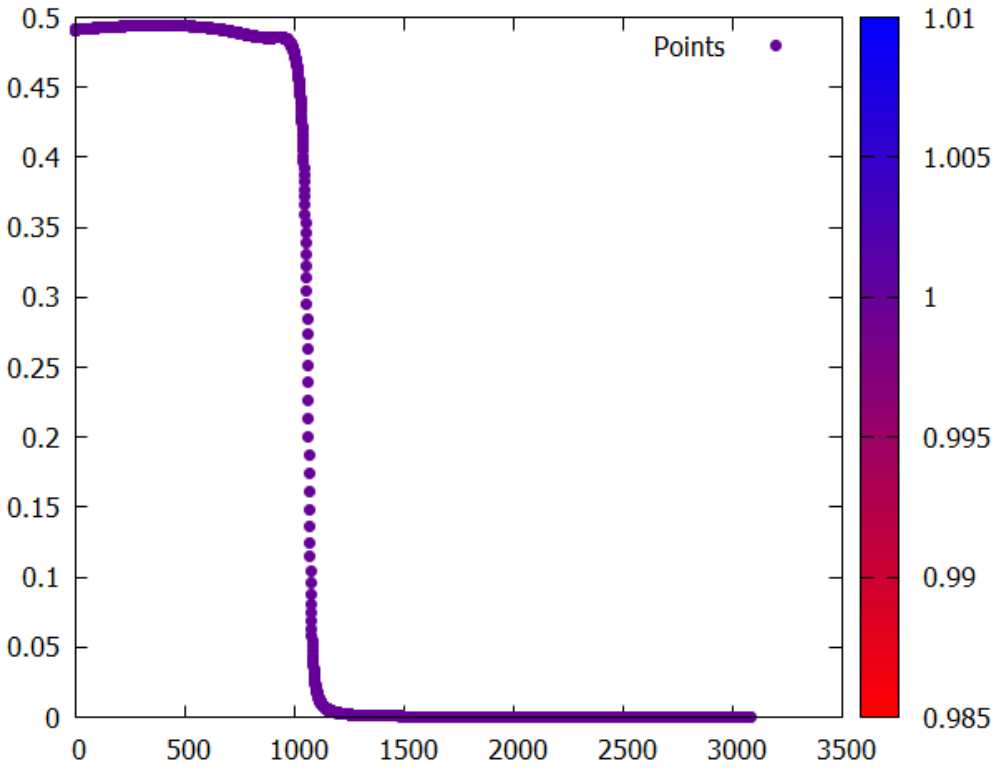

Figure 2: Curva do erro quadrático em um treinamento de rede neural multicamadas para a função lógica XOR

Assim, chamando a função `mlnn-output` com os pesos encontrados, temos o seguinte resultado:

```
(loop for i in tb1-inputs
   collecting (mlnn-output i *w-mlnn* #'bipolar-sigmoid))
```

-0.98566204 0.98679173 0.9861623 -0.9858991

Este bem próximo aos valores esperados:

-1 1 1 -1

6 Conclusão

Durante a implementação e fase de testes, observou-se que a determinação dos pesos é fortemente influenciada pela configuração inicial da rede, tanto a arquitetura quanto os valores iniciais dos pesos. Assim, tal definição é essencial para alcançar resultados que descrevam o modelo com fidelidade.

Apesar desta incerteza na definição dos parâmetros da rede neural, os resultados foram extremamente satisfatórios, sendo possível resolver o problema da função lógica `XOR`, a qual é, por natureza, não linearmente separável. Esta constatação abrange significativamente a gama de aplicações das redes neurais, não sendo mais limitada à este fato.

Em relação à implementação, foi possível codificar com sucesso uma função de treino de uma rede neural arbitrária com retropropagação do erro. Tal rede neural pode ter qualquer configuração, desde que esteja em conformidade ao formato apresentado. Esta função também permite a utilização de outras funções de ativação, sendo simples e direto o treinamento para outras redes, com outras ativações.

References

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