Relatório 5 - Redes de multicamadas com retropropação de erro

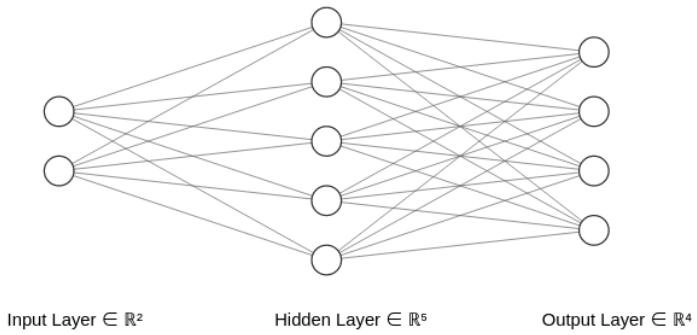
1 Introdução

Os resultados das funções desse relatório foram obtidos com base na seguinte tabela provida pelo professor:

X1	X2	T
1	1	-1
-1	1	1
1	-1	1
-1	-1	-1

As redes de um única camada apesar de poderosas possuem uma severa restrição, elas não resolvem problemas não linearmente separáveis. Sendo assim, a utilização de redes de multicamadas seria lógica, entretanto em 1969 Minsky e Papert mostraram que as redes de duas camadas apesar de resolverem a restrição não tinham uma solução para o problema de ajuste dos pesos da entrada para a camada escondida. Somente em 1986 que Rumelhart, Hilton e Willian apresentariam uma solução para esse problema, a ideia seria de corrigir os erros da camada escondida retropropagando o erro da saída.

Neste relatório foi implementada uma rede de multicamadas que utilizava dessa mesma ideia para aprender a obter as saídas de uma porta lógica XOR. A rede neural implementada apresenta a seguinte arquitetura:



2 Objetivo

- Treinar uma rede neural usando o algoritmo da retropropagação do erro para o caso da função lógica ou exclusivo bipolar
- Desenhar a arquitetura da rede neural
- Plotar a curva do erro quadrático
- Apresentar os pesos encontrados

3 Desenvolvimento

Para realizar o treinamento da rede neural foi utilizada a linguagem Common Lisp com um paradigma primordialmente funcionou o que fez com que o programa tivesse que sofrer algumas alteração visando seu melhor funcionamento e eficiência. As duas principais foram nos pesos, para que cada neurônio ficasse com uma lista de pesos foi preciso inverter as linhas e colunas das matrizes de pesos. Assim, a matriz de pesos da camada escondida é dada pelo número de neurônios na camada escondida pelo número de neurônios na camada de entrada e a matriz de pesos da camada de saída é dada pelo número de neurônios na camada de saída por número de neurônios na camada escondida. Ademais, como cada neurônio possui uma lista de pesos foi preferível colocar o valor do bias como um elemento no final dessa lista para facilitar os cálculos.

Com isso em mente foi criada uma função para gerar os pesos com esses variando de 0.5 a -0.5 e com bias começando em 1.

```
(defun random-weights (i j)
  (make-random-weights i j '()))
(defun make-random-weights (i j list)
  (cond
    ((eql i 0) list)
    (t (make-random-weights (- i 1) j  (append list (list
    ↪  (make-random-weights-aux j '()))))))))
(defun make-random-weights-aux (j list)
  (if (eql j 0) (append list (list 1)) (make-random-weights-aux (- j 1)
  ↪  (append list (list (- (random 0.5) (random 0.5)))))))
```

Para que o calculo do bias junto com o pesos funcionasse foi necessário também adicionar 1 no final das entradas. Tendo isso em mente e visando facilitar o teste da funções foram criadas as seguintes variáveis:

```
(defvar table-in '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1)))
(defvar table-out '(-1 1 1 -1))
(defvar test-ij (random-weights 5 2))
(defvar test-jk (random-weights 1 5))
```

Também para possibilitar as operações com os pesos e o bias ao mesmo tempo foram criadas as seguintes funções:

```
(defun w-separated (ajusted-weights)
  (butlast ajusted-weights))
(defun b-separated (ajusted-weights)
  (first (last ajusted-weights)))
```

Com isso pronto, foi hora de criar as funções que fariam algumas operações matemáticas recorrentes, essas sendo, respectivamente, o cálculo do erro quadrado, da função sigmoid e da derivada da função sigmoid.

```
(defun squared-error (target output)
  (* 1/2 (expt (- target output) 2)))

(defun sigmoid (output)
  (- (/ 2 (+ 1 (exp (* -1 output)))) 1))

(defun derivated-sigmoid (output)
  (* 1/2 (+ 1 (sigmoid output)) (- 1 (sigmoid output))))
```

Para que as funções principais do código funcionassem foi necessário criar uma série de funções que calculassem valores necessários e sempre utilizados:

```
;; Returns the list of y-in-j
;; List of lists, list -> List
(defun y-in-j (weights-ij input)
  (mapcar #'(lambda (w-ij)
              (+ (b-separated w-ij)
                 (apply #'+ (map 'list  #'* (w-separated w-ij)
                 ↪  input))))
          weights-ij))

;; Returns the list of the zj
;; List-> List
(defun zj (y-in-j)
  (mapcar #'(lambda (y) (sigmoid y)) y-in-j))

;; Returns the y-in-k
;; List, list -> Number
(defun y-in-k (weights-jk zj)
  (+ (b-separated weights-jk)
     (apply #'+ (map 'list  #'* (w-separated weights-jk) zj))))

;; Returns the yk of the MLN
;; Number -> Number
(defun foward (y-in-k)
  (sigmoid y-in-k))

;; Returns the vakue of delta-k
;; Number, number -> Number
(defun delta-k (target y-in-k)
  (* (- target (sigmoid y-in-k))
     (derivated-sigmoid y-in-k)))

;; Returns a list of the delta-w-jk of the jk weights and of the bias
;; Number, number, list -> list
(defun delta-w-jk (learning-rate delta-k zj)
  (mapcar #'(lambda (z) (* learning-rate delta-k z)) (append zj '(1))))

;; Returns the little-delta-j
;; Number, list -> number
(defun little-delta-j (delta-k weights-jk)
  (apply #'+
```

```
          (mapcar #'(lambda (w) (* delta-k w)) weights-jk)))

;; Returns a list of the delta-v of each neuronium in the hiden layer
;; List, number -> list
(defun delta-j (y-in-j little-delta-j)
  (mapcar #'(lambda (y) (* (derivated-sigmoid y) little-delta-j))
          y-in-j))

;; Returns a list of lists of the delta-w for each weight and bias of
↪  the hiden layer
;; Number, list, list -> list of lists
(defun delta-w-ij (learning-rate delta-j input)
  (mapcar #'(lambda (d)
              (mapcar #'(lambda (x) (* learning-rate d x)) input))
          delta-j))
```

Em posse dessas funções o ajuste de pesos foi muito simples

```
;; Ajust all the weights and bias of the neuroniums of the hiden layer
;; List of lists, list of lists -> list of lists
(defun ajust-ij (weights-ij delta-w-ij)
  (mapcar #'(lambda (w-ij d-ij)
              (mapcar #'(lambda (w d) (+ w d)) w-ij d-ij))
          weights-ij delta-w-ij))

;; Ajust all the weights and bias of the neuroniums of the exit layer
;; List of lists, list -> list of lists
(defun ajust-jk (weights-jk delta-w-jk)
  (mapcar #'(lambda (w-jk)
              (mapcar #'(lambda (w d-jk) (+ w d-jk)) w-jk delta-w-jk))
          weights-jk))
```

Com a ajuste pronto foi possível criar uma função que treinasse os neurônios até que se atingisse um certo número de ciclos ou um erro quadrático menor que o solicitado.

```
;; Trains the weights based on a number of cicles or minimum squared
↪  error
(defun training (inputs targets num-hiden-layer learning-rate
↪  max-cicles min-error)
  (let*
      ((weights-jk (random-weights 1 num-hiden-layer))
```

```
      (weights-ij (random-weights num-hiden-layer (list-length (first
       ↪  inputs)))))
  (test-cicles inputs targets weights-ij weights-jk learning-rate
   ↪  max-cicles min-error 0 (list 0))))
;; Tests if the number maximum of cicles was reached
(defun test-cicles (inputs targets weights-ij weights-jk learning-rate
↪  max-cicles min-error cicles error)
  (cond
    ((or
      (eql cicles max-cicles)
      (and (<= (first (last error)) min-error) (< 0 cicles)))
     (values weights-ij weights-jk cicles (rest error)))
    (t (multiple-value-bind (w-ij w-jk er)
           (ajust-all-weights inputs targets weights-ij weights-jk
            ↪  learning-rate 0)
         (test-cicles inputs targets w-ij w-jk learning-rate max-cicles
          ↪  min-error (1+ cicles) (append error (list er)))))))
;; Ajust all the weights and bias for all the inputs and targets
(defun ajust-all-weights (inputs targets weights-ij weights-jk
↪  learning-rate error)
  (cond
    ((null inputs) (values weights-ij weights-jk error))
    (t (let*
           ((y-j (y-in-j weights-ij (first inputs)))
            (y-k (y-in-k (first weights-jk) (zj y-j)))
            (d-k (delta-k (first targets) y-k))
            (d-ij (delta-w-ij learning-rate (delta-j y-j
             ↪  (little-delta-j d-k (first weights-jk))) (first
             ↪  inputs)))
            (d-jk (delta-w-jk learning-rate d-k (zj y-j))))
           (ajust-all-weights (rest inputs) (rest targets)
                              (ajust-ij weights-ij d-ij)
                              (ajust-jk weights-jk d-jk)
                              learning-rate (+ error (squared-error (first
                               ↪  targets) (foward y-k)))))))))
```

Ao aplicar o treinamento em uma rede com 5 neurônios na camada escondida, para taxa de aprendizado de 0.1, 1000 ciclos e erro quadrático mínimo de 0.0001 obteve-se os seguintes pesos:

Pesos e bias da camada escondida

Pesos: [2.9514215 -2.4504688] Bias: -2.5781941
Pesos: [-2.7489762 3.233215] Bias: -2.6790142
Pesos: [-2.699009 -2.5031939] Bias: 3.066937
Pesos: [2.433608 2.6610575] Bias: 2.5115561
Pesos: [2.99594 -2.50141] Bias: -2.6279352

Pesos e bias da camada escondida

Pesos: [1.3210002 3.1485379 1.8491082 1.1586045 2.0722046] Bias: 1.3965353

Com o treinamento dos pesos pronto, foi hora de testar suas saídas em relação ao target com a seguinte função:

```
(defun compare-outputs (inputs targets weights-ij weights-jk)
  (cond
    ((null inputs)
     (format t "~% Weights and bias of the hiden layer: ~% ~%")
     (map 'nil #'(lambda (weights) (format t "Weights: [~{~a~^ ~}] | b:
     ↪  ~a ~%"
                                         (w-separated weights)
                                         (b-separated weights)))
                                         ↪  weights-ij)
     (format t "~% Weights and bias of the exit layer: ~% ~%")
     (map 'nil #'(lambda (weights) (format t "Weights: [~{~a~^ ~}] | b:
     ↪  ~a ~%"
                                         (w-separated weights)
                                         (b-separated weights)))
                                         ↪  weights-jk))
    (t
     (format t "Expected: ~a | Result ~a ~%"
             (first targets)
             (foward (y-in-k (first weights-jk) (zj (y-in-j weights-ij
             ↪  (first inputs))))))
     (compare-outputs (rest inputs) (rest targets) weights-ij
     ↪  weights-jk))))
```

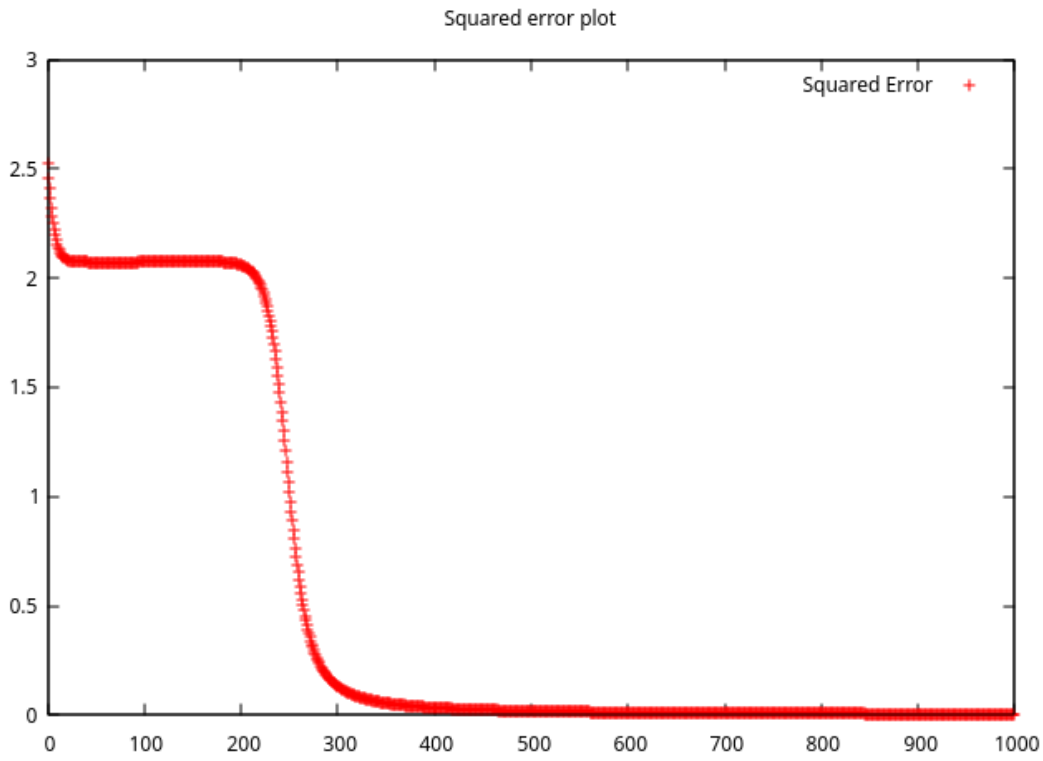
Os resultados podem ser encontrado abaixo.

Por fim foram feitas as funções para plotar o gráfico do erro em relação ao número de ciclos:

```
(defun plot-error (cicles error)
  (plot-error-aux (loop for x upto cicles collect x) error))
```

```
(defun plot-error-aux (x y)
  (progn
        (vgplot:plot x y "r+; Squared Error")
        (vgplot:title "Squared error plot")))
```

Que quando executada com os erros e ciclos de uma rede com 5 neurônios na camada escondida, com taxa de aprendizagem de 0.1, número máximo de ciclos de 1000 e erro mínimo de 0.0001 produz o seguinte gráfico:



4 Conclusão

Foi possivel observer, por meio dessa implementação o poder de uma rede neural de multicamadas com retropropagação de erro, que resolveu o problema do XOR que tanto assolava as outras implementações com muita eficiência. Os resultados obtidos foram muitos satisfatórios, sendo muito próximos do target e tendo apenas um erro esperado. Ademais, foi observado que aumentar a quantidade de neurônios na camada escondida e o número de ciclos aumentam a precisão dos resultados.

5 Resultados

5.1 5 Neurônios na camada escondida e 1000 ciclos

Expected: -1 Result -0.96680653
Expected: 1 Result 0.94989264
Expected: 1 Result 0.9590138
Expected: -1 Result -0.9538826

Weights and bias of the hiden layer

Weights: [2.9514215 -2.4504688] b: -2.5781941
Weights: [-2.7489762 3.233215] b: -2.6790142
Weights: [-2.699009 -2.5031939] b: 3.066937
Weights: [2.433608 2.6610575] b: 2.5115561
Weights: [2.99594 -2.50141] b: -2.6279352

Weights and bias of the exit layer

Weights: [1.3210002 3.1485379 1.8491082 1.1586045 2.0722046] b: 1.3965353

5.2 5 Neurônios na camada escondida e 100000 ciclos

Expected: -1 Result -0.992062
Expected: 1 Result 0.99190116
Expected: 1 Result 0.9918879
Expected: -1 Result -0.99763787

Weights and bias of the hiden layer

Weights: [3.203272 -4.219955] b: -3.1515963
Weights: [-4.199405 3.174768] b: -3.128478
Weights: [-4.1721287 3.1294773] b: -3.084306
Weights: [2.931099 -4.0561976] b: -2.886259
Weights: [2.6811938 2.6778238] b: 3.9665596

Weights and bias of the exit layer

Weights: [4.031347 2.9809232 2.8429143 1.7927235 2.2010393] b: 3.552816