Engenharia de Características

Gustavo Alves Pacheco*

11821ECP011

1 Introdução

Até o momento, as aplicações de rede neurais utilizavam conjuntos pequenos de entrada. Não apenas isto, como também tal conjunto já representava diretamente a grandeza que servia de entrada. Entretanto, na maioria das aplicações reais, o conjunto de dados bruto deve ser convertido no vetor de características, que alimentará a rede neural [1].

Em ocasiões será necessário converter categorias em estruturas binárias, por exemplo, para que tal valor não afete o resultado das multiplicações. Técnica esta conhecida como One-hot encoding. Nela, as categorias 1, 2 e 3 seriam convertidas em colunas 1 0 0, 0 1 0 e 0 0 1, respectivamente.

Em outras, pode ser necessário que os dados sejam normalizados antes de servirem como entradas da rede neural. Isto ocorre quando as grandezas das variáveis são muito diferentes. Assim, pode-se aplicar a técnica do Zscore (eq. 1), que desloca os dados para a média e faz o desvio padrão ser igual a 1, a do MinMax (eq. 2), na qual os novos valores variam entre 0 e 1, ou outras, como a da equação 3, por exemplo, que faz os dados variarem entre y_{min} e y_{max}.

$$z = \frac{x - \text{mean}(x)}{\text{stdev}(x)} \tag{1}$$

$$z = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{2}$$

$$z = \frac{(x - \min(x))(y_{max} - y_{min})}{\max(x) - \min(x)} + y_{min} \tag{3}$$

Outro aspecto interessante a apontar em aplicações reais é a utilização de conjuntos de treinamento, de validação e de teste, para melhorar a capacidade de generalização da rede neural. Assim, o modelo é treinado utilizando o conjunto de treinamento e avaliado utilizando o conjunto de validação, periodicamente. Enquanto apresentar melhorias na avaliação, o treinamento continua. Quando ocorre a convergência com o conjunto de validação, o conjunto de teste é aplicado na rede.

*gap1512@gmail.com

2 Objetivos

- Aprimorar o conhecimento sobre Redes Neurais Artificiais e obter experiência prática na implementação das mesmas.
- Treinar um perceptron com função de ativação logística (sigmóide binária) para classificar sinais de um sonar.

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, Regressão Linear e Multilayer Perceptron.

4 Desenvolvimento

O desenvolvimento a seguir utiliza a base de dados da UCI - Machine Learning Repository (sonar.csv). Busca-se identificar se um sinal corresponde à uma mina ou a uma rocha, de acordo com 60 variáveis. Assim, faz-se necessário a implementação de algumas funções para trabalhar com tal arquivo. São elas: `read-csv`, que deve transformar cada linha do arquivo em uma lista, `parse-double`, que deve converter uma string para um número de ponto flutuante, `shuffle`, para embaralhar as listas, `split-at-last`, para separar as entradas das saídas e `percentual-split`, para que, dada uma lista, consiga dividi-la de forma percentual.

No momento da leitura do arquivo, deseja-se converter a string para ponto flutuante. Portanto, esta será implementada primeiro.

```
(defun parse-double (string)
  (declare (optimize (speed 3)))
  (let ((*read-eval* nil))
    (with-input-from-string (str string)
      (read str nil nil))))
```

Para o primeiro elemento do arquivo, temos:

```
(parse-double "0.0200")
```

Assim, a função de leitura do arquivo é implementada:

```
(defun read-csv (filename &key (separator '(#\,)) (key #'parse-double))
    (with-open-file (stream filename)
      (loop for line = (read-line stream nil)
         while line
         collect (mapcar key (uiop::split-string line :separator
          ↪  separator)))))
```

Tendo o arquivo em formato de lista, é interessante que a mesma seja embaralhada e depois dividida entre conjunto de treinamento, de validação e de teste. Para isto, tem-se que o embaralhamento segue o algoritmo de Fisher-Yates, da seguinte forma:

```
(defun shuffle (list)
  (do ((result (copy-seq list) (swap result i j))
       (j 0 (random i))
       (i (1- (length list)) (1- i)))
      ((zerop i) result)))

(defun swap (list i j)
  (rotatef (nth i list) (nth j list))
  list)
```

A divisão da lista em parcelas ocorre da seguinte forma:

```
(defun percentual-split (list &rest percentages)
  (if (not (= (reduce #'+ percentages) 1))
      (error "Percentages must add to 1")
      (let ((lt (length list)))
        (labels ((rec (lst rates result)
                   (let ((rest (cdr rates)))
                     (if rest
                         (let ((size (floor (* lt (car rates)))))
                           (rec (nthcdr size lst)
                                rest
                                (cons (subseq lst 0 size) result)))
                         (values-list (nreverse (cons lst
                                                      result)))))))
          (rec list percentages nil)))))
```

Os tamanhos de cada lista dependem da porcentagem da lista inicial. Portanto, em caso de frações, arrendonda-se para baixo, e o último elemento sempre recebe todo o resto (evitando que informações sejam perdidas). No geral, apresenta uma boa aproximação da divisão percentual.

E por último, a função utilitária `split-at-last`:

```
(defun split-at-last (list)
  (list (butlast list)
        (last list)))

(defun rock-mine (item)
  (if (string= item "R")
      0
      1))

(defun multiple-split-at-last (list)
  (loop for i in list
     for (inputs output) = (split-at-last i)
     collecting inputs into source
     collecting (rock-mine (car output)) into target
     finally (return (values source target))))
```

Assim, a leitura do arquivo é seguida pela conversão das strings em ponto flutuante, um embaralhamento das linhas e um agrupamento entre os três tipos de conjuntos (treinamento, validação e teste), que ficam salvos nas variáveis a seguir.

```
(defvar *training-set*)
(defvar *validation-set*)
(defvar *test-set*)
(defvar *data* (shuffle (read-csv
↪  #p"c:/home/ufu/amaq/feature-engineering/data/sonar.csv")))

(multiple-value-setq (*training-set* *validation-set* *test-set*)
  (percentual-split *data* 0.7 0.15 0.15))
```

Assim, para aplicação da função `iterative-retropropagation` ao conjunto de teste, é necessário apenas implementar `binary-sigmoid` e sua derivada, `binary-sigmoid^1`:

```
(defun binary-sigmoid (x)
  (/ (1+ (exp (- x)))))
```

```
(defun binary-sigmoid^1 (x)
  (let ((f (binary-sigmoid x)))
    (* f (- 1 f))))
```

O treinamento é então realizado da seguinte forma (devido ao tamanho da lista de pesos, a saída não será mostrada):

```
(multiple-value-bind (inputs outputs)
    (multiple-split-at-last *training-set*)
  (multiple-value-bind (weights err)
      (iterative-retropropagation
       (random-mlnn-list -0.5 0.5 (length inputs) 6 1)
       inputs
       outputs
       #'binary-sigmoid
       #'binary-sigmoid^1
       0.05 10000 0.0001)
    (scatter-plot "plots/training-quadratic-error.png" err nil)
    weights))
```

Para a etapa de validação, será utilizada a seguinte estratégia: Primeiro a rede é treinada, utilizando a chamada acima. Após a conclusão, será executada a função `mlnn-output` com cada um dos elementos do conjunto de teste, e a taxa de acerto será calculada. Este processo (treino e avaliação) se repetirá até que a taxa de acerto seja maior que certo valor, ou o número de repetições estoure certo limite.

```
(defun mlnn-train-validation (training-set validation-set fn fn^1
                              output-fn threshold learning-rate
                              training-cycles training-tolerance
                              validation-cycles validation-tolerance
                              min max &rest configs)
  (multiple-value-bind (validation-inputs validation-outputs)
      (multiple-split-at-last validation-set)
    (multiple-value-bind (training-inputs training-outputs)
        (multiple-split-at-last training-set)
      (do ((hit-rate-validation (- 1 validation-tolerance))
           (best-hit-rate 0)
           (best-err 0)
           (best-weights nil)
           (i 0 (1+ i)))
          ((or (>= i validation-cycles)
```

```
                (> best-hit-rate hit-rate-validation))
           (values best-weights best-err best-hit-rate))
         (multiple-value-bind (weights err)
             (iterative-retropropagation (apply #'random-mlnn-list min
                                                max (length (car
                                                             training-inputs))
                                                configs)
                                         training-inputs
                                         training-outputs fn
                                         fn^1 learning-rate
                                         training-cycles
                                         training-tolerance)
           (let ((hit-rate (mlnn-hit-rate weights validation-inputs
                                          validation-outputs fn
                                          output-fn threshold)))
             (format t "Training #~a Hit Rate: ~a~%" i hit-rate)
             (when (> hit-rate best-hit-rate)
               (setf best-weights weights
                     best-err err
                     best-hit-rate hit-rate))))))))

(defun binary-activation (net threshold)
  (if (>= net threshold) 1 0))

(defun mlnn-hit-rate (mlnn-list inputs outputs activation-fn output-fn
threshold)
  (let ((hits (mapcar #'(lambda (in out)
                          (if (= (funcall output-fn
                                          (mlnn-output in mlnn-list
                                          activation-fn)
                                          threshold)
                                 out)
                              1 0))
          inputs outputs)))
  (/ (reduce #'+ hits) (length hits))))
```

Assim, a função `training-validation-test` é definida:

```
(defun mlnn-train-validation-test (training-set validation-set
                                   test-set fn fn^1 output-fn
                                   threshold learning-rate
```

```
                                    training-cycles training-tolerance
                                    validation-cycles
                                    validation-tolerance min max &rest
                                                                    ↪  configs)
  (multiple-value-bind (weights err hit-rate)
      (apply #'mlnn-train-validation training-set validation-set fn
             fn^1 output-fn threshold learning-rate training-cycles
             training-tolerance validation-cycles validation-tolerance
             min max configs)
    (multiple-value-bind (test-inputs test-outputs)
        (multiple-split-at-last test-set)
      (let ((test-hit-rate (mlnn-hit-rate weights test-inputs
      ↪  test-outputs fn output-fn threshold)))
        (format t "Test Hit Rate: ~a~%" test-hit-rate)
        (values weights err hit-rate test-hit-rate)))))
```

Para evitar que este documento fique poluído com as informações de peso, a função abaixo grava os mesmos em um arquivo:

```
(defun save-weights (filename weights)
  (with-open-file (stream filename :direction :output :if-exists
                          :supersede :if-does-not-exist :create)
    (write weights :stream stream)))

(defun load-weights (filename)
  (with-open-file (stream filename)
    (read stream)))
```

5 Conclusão

Finalmente, a chamada da função é:

```
(multiple-value-bind (weights err hit-rate-validation hit-rate-test)
    (mlnn-train-validation-test *training-set*
                                *validation-set*
                                *test-set*
                                #'binary-sigmoid
                                #'binary-sigmoid^1
                                #'binary-activation 0.5
```

```
                                0.1 5000 0.001
                                10 0 -0.5 0.5 12 1)
  (scatter-plot "plots/training-quadratic-error.png" err nil)
  (save-weights "data/weights.list" weights)
  (format t "Best Hit Rate On Validation: ~$~%Best Hit Rate On Test:
  ↪  ~$~%"
          hit-rate-validation hit-rate-test))
```

```
Training #0 Hit Rate: 24/31
Training #1 Hit Rate: 25/31
Training #2 Hit Rate: 25/31
Training #3 Hit Rate: 26/31
Training #4 Hit Rate: 27/31
Training #5 Hit Rate: 25/31
Training #6 Hit Rate: 24/31
Training #7 Hit Rate: 26/31
Training #8 Hit Rate: 26/31
Training #9 Hit Rate: 26/31
Test Hit Rate: 27/32
Best Hit Rate On Validation: 0.87
Best Hit Rate On Test: 0.84
```

Alguns detalhes interessantes de se observar são:

1. O limiar de ativação foi colocado em 0.5.
2. Os pesos iniciais variam de -0.5 a 0.5, aleatoriamente.
3. São treinadas 10 redes neurais, e a que se desempenhar melhor na etapa de validação é escolhida.
4. A rede neural é composta de 1 camada escondida, com 12 neurônios.
5. Várias execuções foram feitas como forma de teste. A taxa de acerto ficou quase sempre bem próxima a 81%, sendo o máximo valor alcançado de 87%.
6. Em algumas situações, foi melhor diminuir a tolerância, e o número de ciclos de treinamento, para que a rede não ficasse excessivamente treinada, indo mal no conjunto de teste.
7. A rede acima colocada apresentou taxa de acerto de 87% no conjunto de validação e 84% no de teste.

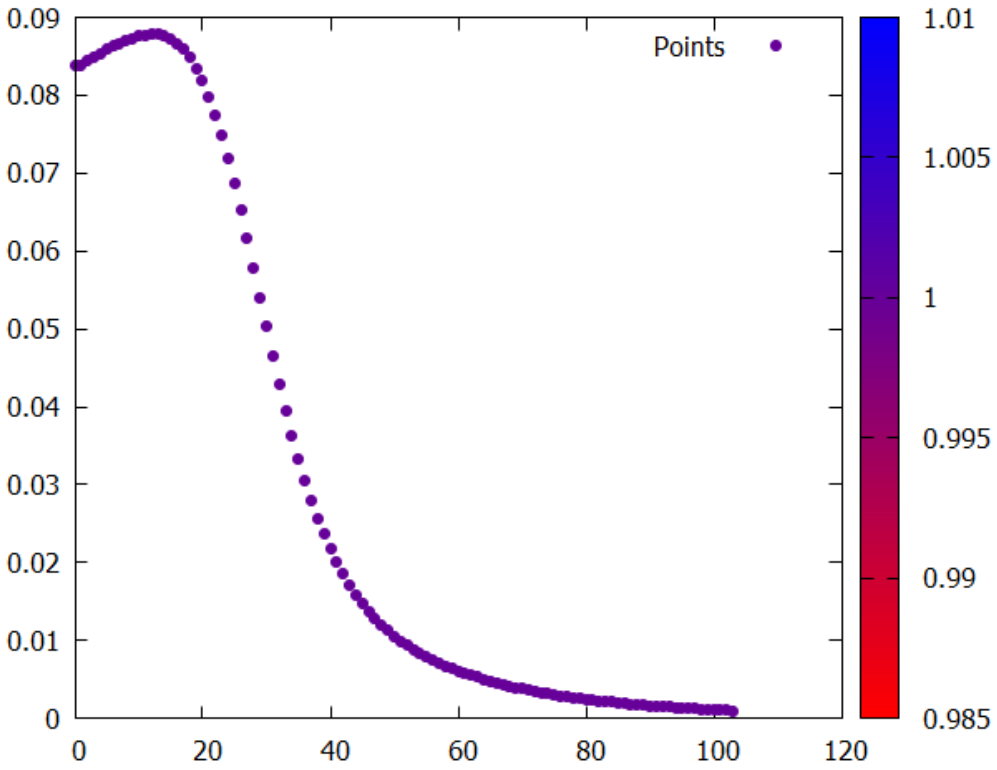

Figure 1: Erro Quadrático Em Tempo De Treinamento

References

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