Regra de Hebb

Gustavo Alves Pacheco*

11821ECP011

1 Introdução

A seguir, está descrito o desenvolvimento de um programa em Common Lisp para treinamento de um neurônio artificial de McCullock-Pitts utilizando a Regra de Hebb. Tal regra representa a primeira estratégia de treinamento proposta na literatura de Aprendizagem de Máquina. Seu desenvolvedor, Donald Hebb, era um psicólogo que descobriu que o condicionamento promove alterações na estrutura sináptica.

A plasticidade sináptica mede a eficiência na alteração sináptica, resultando em um modelo de aprendizado. Modelo este que atua sobre o neurônio de McCulloch-Pitts, composto de entradas reais (x_1 a x_n), conectadas ao núcleo `y` através de pesos (w_1 a w_n), que podem ser excitatórios ($w_i > 0$) ou inibitórios ($w_i < 0$). Do núcleo, tem-se a saída binária `f(y)`, que é uma função degrau, configurada através de um limiar θ, fixo, definido para que a inibição seja absoluta. Esta função compara o resultado da `net` com o limiar. A `net`, por sua vez, corresponde ao somatório do produto entre entrada e o peso da entrada correspondente, mais um termo `b` (*bias*), vide 1 [1].

$$net = \sum_{i=1}^{n} w_i * x_i + b \tag{1}$$

O treinamento consiste na determinação dos valores de w_i e b do neurônio, dado um conjunto de entradas (*source*) e suas respectivas saídas (*target*). Na regra de Hebbs, para cada item do grupo de treino tem-se um ajuste no valor de w_i e b, dado pela equação 2.

$$\Delta w_i = x_i * t \tag{2}$$

A seção de desenvolvimento mostra as etapas utilizadas para implementação de um algoritmo (seguindo a regra de Hebb) destinado ao treinamento de um neurônio em cada uma das 16 funções lógicas que podem ser construídas a partir de 2 entradas binárias.

*gap1512@gmail.com

2 Objetivos

- Aprimorar o conhecimento sobre Redes Neurais Artificiais e obter experiência prática na implementação das mesmas, partindo de uma rede com um único Neurônio Artificial.
- Aplicar a regra de Hebb a funções lógicas de duas variáveis em representação bipolar.
- Realizar o treinamento da rede neural para cada uma das 16 funções lógicas obtidas a partir de duas entradas binárias.

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.

4 Desenvolvimento

De maneira geral, o programa a ser implementado deve apresentar a função `neural-network`, que recebe uma função de treino (`training`) e uma função de execução (`running`), bem como os argumentos necessários para cada uma das duas funções.

```
(in-package :machine-learning)
```

Iniciando pela função de treino, é necessário que a mesma receba o `source`, o `target` e uma lista com os pesos iniciais, retornando, assim, uma lista com os pesos após o treinamento. Para um único item, implementa-se a regra de Hebb, conforme 2:

```
(defun hebb (source target weights)
  (mapcar #'(lambda (w x)
              (+ w (* x target)))
          weights source))
```

Sendo a chamada da função algo do tipo:

```
(hebb
 '(-1 1 1) ;;source
 1         ;;target
 '(0 0 0)) ;;initial-weights
```

-1 1 1

Definido o treino para um item do conjunto, é fácil expandir o comportamento para abranger uma lista, logo:

```
(defun training (source target weights)
  (do* ((w weights (hebb (car src) (car trg) w))
        (src source (cdr src))
        (trg target (cdr trg)))
       ((or (not src) (not trg)) w)))
```

A chamada dessa função para a porta lógica `or`, é a seguinte:

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

2 2 -2

Os valores de saída representam w_1, w_2 e b, respectivamente. Tendo em mãos os valores dos coeficientes, a função de execução deve ser definida, para que o neurônio desempenhe a tarefa para a qual foi treinado.

```
(defun net (weights input)
  (apply #'+ (mapcar #'* weights input)))

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

(defun running (inputs weights threshold net-fn activation-fn)
  (mapcar #'(lambda (i)
              (funcall activation-fn
                       (funcall net-fn
                                weights i)
                       threshold))
          inputs))
```

Nesta definição, `running` é uma função de alta ordem, permitindo que comportamentos diferentes sejam atingidos, dependendo dos parâmetros passados. `net` é a implementação direta da 1, enquanto `activation` representa a função degrau de ativação. A chamada da mesma, para a tabela `or`, utilizando os pesos encontrados no treinamento é a seguinte:

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

1 -1 -1 -1

Como o resultado foi o mesmo da tabela verdade para o operador `or`, o programa está executando corretamente. Vale observar que a terceira coluna do `inputs` e do `source` (terceiro elemento de cada sublista) deve sempre apresentar o valor 1, pois esta entrada correspondente ao peso `b`. Juntando as duas definições, temos:

```
(defun neural-network (training-fn source target
↪  initial-weights running-fn
                      inputs threshold net-fn activation-fn)
  (let ((w (funcall training-fn source target
  ↪  initial-weights)))
    (values (funcall running-fn inputs w threshold net-fn
    ↪  activation-fn)
            w)))
```

A qual é executada da seguinte maneira:

```
(neural-network
 #'training                             ;;training-fn
 '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1)) ;;source
 '(1 -1 -1 -1)                          ;;target
 '(0 0 0)                               ;;initial-weights
 #'running                              ;;running-fn
 '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1)) ;;inputs
 0                                      ;;threshold
 #'net                                  ;;net-fn
 #'activation)                          ;;activation-fn
```

1 -1 -1 -1

Vale observar que a função `neural-network` possui dois valores de retorno. O primeiro é a saída da rede neural, quando executada nas condições especificadas e o segundo é uma lista que contém os valores de coeficientes obtidos durante o treinamento.

Agora, resta testar a rede neural para as 16 configurações possíveis de entradas e saídas lógicas (com duas variáveis). Para que a visualização das comparações seja facilitada, uma camada será feita, por cima da função `neural-network`. Esta nova função, implementada abaixo, permite comparar o resultado desejado com o obtido após o treinamento.

```
(defun neural-network-comparison (training-fn source target
↪  initial-weights
                                  running-fn inputs threshold
                                  ↪  net-fn
                                  activation-fn)
  (multiple-value-bind (output weights)
      (neural-network training-fn source target initial-weights
      ↪  running-fn
                      inputs threshold net-fn activation-fn)
    (with-output-to-string (str)
      (format str
              "Obtained Weights: [~{~a~^ ~}]~%"
              weights)
      (mapcar #'(lambda (tar out)
                  (format str
                          "Expected: ~a | Obtained: ~a |
                          ↪  [~:[Fail~;Pass~]]~%"
                          tar out (eq tar out)))
              target
              output)
      str)))
```

Para a mesma chamada anterior, utilizando a nova função, obtemos a seguinte saída:

```
(neural-network-comparison
 #'training                              ;;training-fn
 '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1)) ;;source
 '(1 -1 -1 -1)                          ;;target
 '(0 0 0)                               ;;initial-weights
 #'running                              ;;running-fn
 '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1)) ;;inputs
 0                                      ;;threshold
 #'net                                  ;;net-fn
 #'activation)                          ;;activation-fn
```

```
Obtained Weights: [2 2 -2]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
```

Assim, a verificação do treino das 16 funções lógicas é trivial (ver seção 6 para saída desta execução).

```
(mapcar
 #'(lambda (target)
     (neural-network-comparison
      #'training
      ↪ ;;training-fn
      '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1))          ;;source
      target                                           ;;target
      '(0 0 0)
      ↪ ;;initial-weights
      #'running
      ↪ ;;running-fn
      '((1 1 1) (-1 1 1) (1 -1 1) (-1 -1 1))          ;;inputs
      0
      ↪ ;;threshold
      #'net                                            ;;net-fn
      #'activation))
      ↪ ;;activation-fn
 '((-1 -1 -1 -1) (-1 -1 -1 1) (-1 -1 1 -1) (-1 -1 1 1) ;;16
 ↪ possible targets
   (-1 1 -1 -1) (-1 1 -1 1) (-1 1 1 -1) (-1 1 1 1)
   (1 -1 -1 -1) (1 -1 -1 1) (1 -1 1 -1) (1 -1 1 1)
   (1 1 -1 -1) (1 1 -1 1) (1 1 1 -1) (1 1 1 1)))
```

5 Conclusão

Pelos resultados obtidos, é possível observar que 14 das 16 funções apresentaram a saída correta. Entretanto, duas divergências ocorreram. Tanto em `'(-1 1 1 -1)` quanto em `'(1 -1 -1 1)`. Nas duas situações, todos os pesos possuíam valor 0. Apesar desse ocorrido, foi possível treinar com sucesso o neurônio nos outros 14 casos, utilizando a representação bipolar e a regra de Hebb.

Tal constatação é bastante interessante, pois foi possível treinar uma máquina, através de operações extremamente simples, para que desempenhe uma tarefa de forma autônoma. Obviamente, o trabalho realizado pelo neurônio é de certa forma simples, mas utilizando os mesmos conceitos é possível expandir

as definições propostas, criando redes que aprendam a executar funções mais complexas.

6 Resultados

```
("Obtained Weights: [0 0 -4]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
"
 "Obtained Weights: [-2 -2 -2]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
"
 "Obtained Weights: [2 -2 -2]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
"
 "Obtained Weights: [0 -4 0]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
"
 "Obtained Weights: [-2 2 -2]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
"
 "Obtained Weights: [-4 0 0]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
"
 "Obtained Weights: [0 0 0]
Expected: -1 | Obtained: 1 | [Fail]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: 1 | [Fail]
"
 "Obtained Weights: [-2 -2 2]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
"
 "Obtained Weights: [2 2 -2]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
"
 "Obtained Weights: [0 0 0]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: 1 | [Fail]
Expected: -1 | Obtained: 1 | [Fail]
Expected: 1 | Obtained: 1 | [Pass]
"
 "Obtained Weights: [4 0 0]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
"
 "Obtained Weights: [2 -2 2]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
"
 "Obtained Weights: [0 4 0]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
"
 "Obtained Weights: [-2 2 2]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
"
 "Obtained Weights: [2 2 2]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: -1 | Obtained: -1 | [Pass]
"
 "Obtained Weights: [0 0 4]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
Expected: 1 | Obtained: 1 | [Pass]
")
```

References

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