Perceptron*

Jason Brownlee
jasonb@CleverAlgorithms.com
The Clever Algorithms Project
http://www.CleverAlgorithms.com

November 16, 2010
Technical Report: CA-TR-20101116b-1

Abstract

The Clever Algorithms project aims to describe a large number of Artificial Intelligence algorithms in a complete, consistent, and centralized manner, to improve their general accessibility. The project makes use of a standardized algorithm description template that uses well-defined topics that motivate the collection of specific and useful information about each algorithm described. This report describes the Perceptron algorithm using the standardized algorithm template.

Keywords: `Clever, Algorithms, Description, Optimization, Perceptron`

1 Introduction

The Clever Algorithms project aims to describe a large number of algorithms from the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics in a complete, consistent and centralized manner [1]. The project requires all algorithms to be described using a standardized template that includes a fixed number of sections, each of which is motivated by the presentation of specific information about the technique [2]. This report describes the Perceptron algorithm using the standardized algorithm template.

2 Name

Perceptron

3 Taxonomy

The Perceptron algorithm belongs to the field of Artificial Neural Networks and more broadly Computational Intelligence. It is a single layer feedforward neural network (single cell network) that inspired many extensions and variants, not limited to Adalines and Widrow Hoff learning rules.

4 Inspiration

The Perceptron is inspired by the information processing of a single neural cell (called a neuron). A neuron accepts input signals via the dendrites, a chemical process occurs within the cell based on the input signals, and the cell may or may not produce an output signal on its axon. The point where one cells axon interfaces another cells dendrite is called the synapse, which may fire if the cell is activated.

5 Strategy

The information processing objective of the technique is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state. State is maintained in a set of weightings on the input signals. The weights are used to represent an abstraction of the mapping of input vectors to the output signal for the examples that the system was exposed to during training.

6 Procedure

The Perceptron is comprised of a data structure (weights) and separate procedures for training and applying the structure. The structure is really just a vector of weights (one for each expected input) and a bias term.

Algorithm 1 provides a pseudo-code for training the Perceptron. A weight is initialized for each input plus an additional weight for a fixed bias constant input that is almost always set to 1.0. The activation of the network to a given input pattern is calculated as follows:

$$activation \leftarrow \sum_{k=1}^{n} (w_k \times x_{ki}) + w_{bias} \times 1.0 \qquad (1)$$

where n is the number of weights and inputs, x_{ki} is the k^{th} attribute on the i^{th} input pattern, and w_{bias} is the bias weight. The weights are updated as follows:

$$w_i(t+1) = w_i(t) + \alpha \times (e(t) - a(t)) \times x_i(t) \qquad (2)$$

where w_i is the i^{th} weight at time t and $t+1$, α is the learning rate, $e(t)$ and $a(t)$ are the expected and actual output at time t, and x_i is the i^{th} input. This update process is applied to each weight in turn (as well as the bias weight with its contact input).

Algorithm 1: Pseudo Code for the Perceptron algorithm (training weights).

Input: ProblemSize, InputPatterns, $iterations_{max}$, $learn_{rate}$
Output: Weights

Weights $\leftarrow$ `InitializeWeights`(ProblemSize);
for $i = 1$ **to** $iterations_{max}$ **do**
  $Pattern_i \leftarrow$ `SelectInputPattern`(InputPatterns);
  $Activation_i \leftarrow$ `ActivateNetwork`($Pattern_i$, Weights);
  $Output_i \leftarrow$ `TransferActivation`($Activation_i$);
  `UpdateWeights`($Pattern_i$, $Output_i$, $learn_{rate}$);
end
return Weights;

7 Heuristics

- The Perceptron can be used to approximate arbitrary linear functions and can be used for regression or classification problems.
- The Perceptron cannot learn a non-linear mapping between the input and output attributes. The XOR problem is a classical example of a problem that the Perceptron cannot learn.
- Input and output values should be normalized such that $x \in [0, 1)$.
- The learning rate ($\alpha \in [0, 1]$) controls the amount of change each error has on the system, lower learning rages are common such as 0.1.
- The weights can be updated in an online manner (after the exposure to each input pattern) or in batch (after a fixed number of patterns have been observed).
- Batch updates are expected to be more stable than online updates for some complex problems.
- A bias weight is used with a constant input signal to provide stability to the learning process.
- A step transfer function is commonly used to transfer the activation to a binary output value $1 \leftarrow activation \geq 0$, otherwise 0.
- It is good practice to expose the system to input patterns in a different random order each enumeration through the input set.
- The initial weights are typically small random values, typically $\in [0, 0.5]$.

8 Code Listing

Listing 1 provides an example of the Perceptron algorithm implemented in the Ruby Programming Language. The problem is a contrived classification problem in a 2-dimensional domain $x \in [0, 1], y \in [0, 1]$ with two classes: 'A' ($x \in [0, 0.4999999], y \in [0, 0.4999999]$) and 'B' ($x \in [0.5, 1], y \in [0.5, 1]$).

The algorithm was implementated using an online learning method, meaning the weights are updated after each input pattern is observed. A step transfer function is used to convert the activation into a binary output $\in \{0, 1\}$. Random samples are taken from the domain to train the weights, and similary, random samples are drawn from the domain to demonstrate what the network has learned. A bias weight is used for stability with a constant input of 1.0.

```
def random_vector(minmax)
  return Array.new(minmax.length) do |i|
    minmax[i][0] + ((minmax[i][1] - minmax[i][0]) * rand())
  end
end

def normalize_class_index(class_no, domain)
  return (class_no.to_f/(domain.length-1).to_f)
end

def denormalize_class_index(normalized_class, domain)
  return (normalized_class*(domain.length-1).to_f).round.to_i
end

def generate_random_pattern(domain)
```

```ruby
  classes = domain.keys
  selected_class = rand(classes.length)
  pattern = {}
  pattern[:class_number] = selected_class
  pattern[:class_label] = classes[selected_class]
  pattern[:class_norm] = normalize_class_index(selected_class, domain)
  pattern[:vector] = random_vector(domain[classes[selected_class]])
  return pattern
end

def initialize_weights(problem_size)
  minmax = Array.new(problem_size + 1) {[0,0.5]}
  return random_vector(minmax)
end

def update_weights(problem_size, weights, input, out_expected, output_actual, learning_rate)
  problem_size.times do |i|
    weights[i] += learning_rate * (out_expected - output_actual) * input[i]
  end
  weights[problem_size] += learning_rate * (out_expected - output_actual) * 1.0
end

def calculate_activation(weights, vector)
  sum = 0.0
  vector.each_with_index do |input, i|
    sum += weights[i] * input
  end
  sum += weights[vector.length] * 1.0
  return sum
end

def transfer(activation)
  return (activation >= 0) ? 1.0 : 0.0
end

def get_output(weights, pattern, domain)
  activation = calculate_activation(weights, pattern[:vector])
  out_actual = transfer(activation)
  out_class = domain.keys[denormalize_class_index(out_actual, domain)]
  return [out_actual, out_class]
end

def train_weights(weights, domain, problem_size, iterations, lrate)
  iterations.times do |epoch|
    pattern = generate_random_pattern(domain)
    out_v, out_c = get_output(weights, pattern, domain)
    puts "> train got=#{out_v}(#{out_c}), exp=#{pattern[:class_norm]}(#{pattern[:class_label]})"
    update_weights(problem_size, weights, pattern[:vector], pattern[:class_norm], out_v, lrate)
  end
end

def test_weights(weights, domain)
  correct = 0
  100.times do
    pattern = generate_random_pattern(domain)
    out_v, out_c = get_output(weights, pattern, domain)
    correct += 1 if out_c == pattern[:class_label]
  end
  puts "Finished test with a score of #{correct}/#{100} (#{(correct/100)*100}%)"
end

def run(domain, problem_size, iterations, learning_rate)
  weights = initialize_weights(problem_size)
```

```
  train_weights(weights, domain, problem_size, iterations, learning_rate)
  test_weights(weights, domain)
end

if __FILE__ == $0
  problem_size = 2
  domain = {"A"=>[[0,0.4999999],[0,0.4999999]],"B"=>[[0.5,1],[0.5,1]]}
  learning_rate = 0.1
  iterations = 60

  run(domain, problem_size, iterations, learning_rate)
end
```

Listing 1: Perceptron algorithm in the Ruby Programming Language

9 References

9.1 Primary Sources

The Perceptron algorithm was proposed by Rosenblatt in 1958 [5]. Rosenblatt proposed a range of neural network structures and methods. The 'Perceptron' as it is known is in fact a simplification of Rosenblatt's models by Minsky and Papert for the purposes of analysis [3]. An early proof of convergence was provided by Novikoff [4]

9.2 Learn More

Minsky and Papert wrote the classical text titled "Perceptrons" in 1969 that is known to have discredited the approach, suggesting it was limited to linear discrimination, which limited research in the area for decades afterward [3].

10 Conclusions

This report described the Perceptron algorithm using the standardized algorithm template.

11 Contribute

Found a typo in the content or a bug in the source code? Are you an expert in this technique and know some facts that could improve the algorithm description for all? Do you want to get that warm feeling from contributing to an open source project? Do you want to see your name as an acknowledgment in print?

Two pillars of this effort are i) that the best domain experts are people outside of the project, and ii) that this work is subjected to continuous improvement. Please help to make this work less wrong by emailing the author 'Jason Brownlee' at jasonb@CleverAlgorithms.com or visit the project website at http://www.CleverAlgorithms.com.

References

[1] Jason Brownlee. The clever algorithms project: Overview. Technical Report CA-TR-20100105-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[2] Jason Brownlee. A template for standardized algorithm descriptions. Technical Report CA-TR-20100107-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[3] M. L. Minsky and S. A. Papert. *Perceptrons.* MIT Press, 1969.

[4] A. B. Novikoff. On convergence proofs on perceptrons. *Symposium on the Mathematical Theory of Automata*, 12:615–622, 1962.

[5] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Cornell Aeronautical Laboratory, Psychological Review*, 6:386–408, 1958.