Back-propagation Algorithm*

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

November 17, 2010
Technical Report: CA-TR-20101117-1

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 Back-propagation algorithm using the standardized algorithm template.

Keywords: `Clever, Algorithms, Description, Optimization, Back-propagation`

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 Back-propagation algorithm using the standardized algorithm template.

2 Name

Back-propagation, Backpropagation, Error Back Propagation, Backprop, Delta-rule

3 Taxonomy

The Back-propagation algorithm is a supervised learning method for multi-layer feed-forward networks from the field of Artificial Neural Networks and more broadly Computational Intelligence. The name refers to the backward propagation of error during the training of the network. Back-propagation is the basis for many variations and extensions for training multi-layer feed-forward networks not limited to Vogl's Method (Bold Drive), Delta-Bar-Delta, Quickprop, and Rprop.

4 Inspiration

Feed-forward neural networks are inspired by the information processing of one or more neural cells (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 cell's axon interfaces another cell's dendrite is called the synapse, which may fire if the cell is activated. The Back-propagation algorithm is a training regime for multi-layer feed forward neural networks and is not directly inspired by the learning processes the biological system.

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. Each layer of the network provides an abstraction of the information processing of the previous layer, allowing the combination of sub-functions and higher order modeling.

6 Procedure

The Back-propagation algorithm is a method for training the weights in a multi-layer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer. The method is primarily concerned with adapting the weights to the calculated error in the presence of input patterns, and the method is applied backward from the network output layer through to the input layer.

Algorithm 1 provides a high-level pseudo-code for preparing a network using the Back-propagation training method. 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 a single neuron to a given input pattern is calculated as follows:

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

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. A logistic transfer function (sigmoid) is used to calculate the output for a neuron $\in [0, 1]$ and provide nonlinearities between in the input and output signals: $\frac{1}{1+exp(-a)}$, where a represents the neuron activation.

The weight updates use the delta rule, specifically a changed delta rule where error is backwardly propagated through the network, starting at the output layer and weighted back through the previous layers. Error derivatives are assigned to each neuron weight and the weights are updated to reduce the error, metered by a learning coefficient. The equations for the back propagation of error are not included for brevity.

7 Heuristics

- The Back-propagation algorithm can be used to train a multi-layer network to approximate arbitrary non-linear functions and can be used for regression or classification problems.

Algorithm 1: Pseudo Code for the Back-propagation algorithm (training weights).

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

```
Network ← ConstructNetworkLayers();
Network_weights ← InitializeWeights(Network, ProblemSize);
for i = 1 to iterations_max do
    Pattern_i ← SelectInputPattern(InputPatterns);
    Output_i ← ForwardPropagate(Pattern_i, Network);
    BackwardPropagateError(Pattern_i, Output_i, Network);
    UpdateWeights(Pattern_i, Output_i, Network, learn_rate);
end
return Network;
```

- Input and output values should be normalized such that $x \in [0, 1)$.
- 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 logistic function (sigmoid) transfer function is commonly used to transfer the activation to a binary output value, although other transfer functions can be used such as the hyperbolic tangent (tanh), Gaussian, and softmax.
- 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 $\in [0, 0.5]$.
- Typically a small number of layers are used such as 2-4 given that the increase in layers result in an increase in the complexity of the system and the time required to train the weights.
- The learning rate can be varied during training, and it is common to introduce a momentum term to limit the rate of change.
- The weights of a give network can be initialized with a global optimization method before being refined using the Back-propagation algorithm.

8 Code Listing

Listing 1 provides an example of the Back-propagation 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 implemented using an online learning method, meaning the weights are updated after each input pattern is observed. A logistic (sigmoid) transfer function is used to convert the activation into an output signal. Random samples are taken from the domain to train the weights, and similarly, random samples are drawn from the domain to demonstrate what the network has learned.

A three layer network is demonstrated with 2 nodes in the input layer, 2 nodes in the hidden layer and 1 node in the output layer, which is more than sufficient for the problem. A bias

weight is used on each neuron for stability with a constant input of 1.0. The learning process is separated into four steps: forward propagation, backward propagation of error, calculation of error derivatives (assigning blame to the weights) and the weight update. This separation facilities easy extensions such as adding a momentum term and/or weight decay to the update process.

```
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)
  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) {[-0.5,0.5]}
  return random_vector(minmax)
end

def activate(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 1.0 / (1.0 + Math.exp(-activation))
end

def transfer_derivative(output)
  return output * (1.0 - output)
end

def forward_propagate(network, pattern, domain)
  network.each_with_index do |layer, i|
    input = (i==0) ? pattern[:vector] : Array.new(network[i-1].size){|k|
        network[i-1][k][:output]}
    layer.each do |neuron|
      neuron[:activation] = activate(neuron[:weights], input)
      neuron[:output] = transfer(neuron[:activation])
    end
  end
  out_actual = network.last[0][:output]
```

```
  out_class = domain.keys[denormalize_class_index(out_actual, domain)]
  return [out_actual, out_class]
end

def backward_propagate_error(network, pattern)
  network = network.reverse
  network.each_with_index do |layer, i|
    back_signal = pattern[:class_norm]
    layer.each_with_index do |neuron, k|
      if i > 0
        prev_layer, back_signal = network[i-1], 0.0
        prev_layer.each_with_index do |prev_neuron, j|
          # only sum errors weighted by connection to k'th neuron
          back_signal += (prev_neuron[:weights][k] * prev_neuron[:error_delta])
        end
      end
      error = (back_signal - neuron[:output])
      neuron[:error_delta] = error * transfer_derivative(neuron[:output])
    end
  end
end

def calculate_error_derivatives_for_weights(network, pattern)
  network.each_with_index do |layer, i|
    input = (i==0) ? pattern[:vector] : Array.new(network[i-1].size){|k|
        network[i-1][k][:output]}
    layer.each do |neuron|
      derivatives = Array.new(neuron[:weights].length)
      input.each_with_index do |signal, j|
        derivatives[j] = neuron[:error_delta] * signal
      end
      derivatives[derivatives.length-1] = neuron[:error_delta] * 1.0
      neuron[:error_derivative] = derivatives
    end
  end
end

def update_weights(network, lrate)
  network.each do |layer|
    layer.each do |neuron|
      neuron[:weights].each_with_index do |w, j|
        neuron[:weights][j] = w + (lrate * neuron[:error_derivative][j])
      end
    end
  end
end

def train_network(network, domain, problem_size, iterations, lrate)
  iterations.times do |it|
    pattern = generate_random_pattern(domain)
    out_v, out_c = forward_propagate(network, pattern, domain)
    puts "> train got=#{out_v}(#{out_c}), exp=#{pattern[:class_norm]}(#{pattern[:class_label]})"
    backward_propagate_error(network, pattern)
    calculate_error_derivatives_for_weights(network, pattern)
    update_weights(network, lrate)
  end
end

def test_network(network, domain)
  correct = 0
  100.times do
    pattern = generate_random_pattern(domain)
    out_v, out_c = forward_propagate(network, pattern, domain)
```

```ruby
    correct += 1 if out_c == pattern[:class_label]
  end
  puts "Finished test with a score of #{correct}/#{100} (#{correct}%)"
end

def create_neuron(num_inputs)
  neuron = {}
  neuron[:weights] = initialize_weights(num_inputs)
  return neuron
end

def create_layer(num_neurons, num_inputs)
  return Array.new(num_neurons){create_neuron(num_inputs)}
end

def run(domain, problem_size, iterations, hidden_layer_size, learning_rate)
  network = []
  network << create_layer(problem_size, problem_size)
  network << create_layer(hidden_layer_size, problem_size)
  network << create_layer(1, hidden_layer_size)

  train_network(network, domain, problem_size, iterations, learning_rate)
  test_network(network, 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
  hidden_layer_size = 2
  iterations = 100

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

Listing 1: Back-propagation algorithm in the Ruby Programming Language

9 References

9.1 Primary Sources

The backward propagation of error method is credited to Bryson and Ho in [3]. It was applied to the training of multi-layer networks and called back-propagation by Rumelhart, Hinton and Williams in 1986 [6, 7]. This effort and the collection of studies edited by Rumelhart and McClelland that helped to define the field of Artificial Neural Networks in the late 1980's [8, 9].

9.2 Learn More

A seminal book on the approach was "Backpropagation: theory, architectures, and applications" by Chauvin and Rumelhart that provided an excellent introduction (chapter 1) but also a collection of studies applying and extending the approach [4]. Reed and Marks provide an excellent treatment of feed-forward neural networks called "Neural Smithing" that includes chapters dedicated to Back-propagation, the configuration of its parameters, error surface and speed improvements [5].

10 Conclusions

This report described the Back-propagation algorithm using the standardized algorithm template. The implementation of the algorithm was based on a past implementation for the WEKA machine learning workbench http://wekaclassalgos.sourceforge.net. The proposed demonstration implementation in Ruby does not converge and remains an area for further work. Additionally, the sample problem is too simple and must be elaborated to a non-linearly separable classification problem (such as spirals).

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] Arthur Earl Bryson and Yu-Chi Ho. *Applied optimal control: optimization, estimation, and control*. Taylor & Francis, 1969.

[4] Yves Chauvin and David E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Routledge, 1995.

[5] Russell D. Reed and Robert J. Marks II. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Mit Press, 1999.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1*, chapter Learning internal representations by error propagation, pages 318–362. MIT Press, 1986.

[8] David E. Rumelhart and James L. McClelland. *Parallel distributed processing: explorations in the microstructure of cognition. Foundations, Volume 1*. MIT Press, 1986.

[9] David E. Rumelhart and James L. McClelland. *Parallel distributed processing: Explorations in the microstructure of cognition. Psychological and biological models, Volume 2*. MIT Press, 1986.