# Introduction

In large neural networks simulating more complex functions, finding the weights of each synapse is too complex to be done manually, instead the back propagation algorithm is commonly used to find weights which will approximate the desired behaviour.

Back propagation works by calculating the difference between each output of the network and the expected output for the inputs to the network, then propagating this error backwards through the network to adjust the weights and reduce the error in the future. By repeating this train and test loop over a large range of valid inputs, the neural network will “learn” the expected behaviour and the errors will decrease. Back propagation can be run for any number of valid inputs, however the network will eventually reach a state where back propagation can no longer reduce the average error.

In some cases, the inputs used to train the network can be picked randomly and the expected output can be calculated directly from this, e.g. a network trained to add numbers or apply the sin function to its input. Other cases require the training examples to be pre computed, for example, in training a network to recognise the character in an image, the character in each image will have to be manually given as the expected output.

This graph shows how the average total error of a network decreases as back propagation teaches the network using 50,000 examples. The network trained is a two input and two output network with 40 neurons in a hidden layer used to convert Cartesian coordinates to polar coordinates.

# The Algorithm

## Initialisation

Before the back propagation is run, the weights all synapses in the network are initialised to small random values, usually between -1 and 1. The range of the initial values is not important so long as the weights are not all equal, in this case, back propagation will change each weight by the same amount and the behaviour of the network will never change significantly.

## Train Test Loop

Once all weights are initialised, apply an input in the training examples to the network and measure the error for each output of the network which is simply the difference between the actual outputs and the expected outputs.

### Error Propagation

The first step is to propagate the errors backwards by computing the delta value, or error value of each neuron.

First, calculate the delta values for each neuron in the medial layer by multiplying the weight of each synapse leaving the neuron by the error of the output which the synapse connects to. All weight-error products of a neuron are then summed to give the final delta value of that neuron.

[D1]

The network contains another layer, so this procedure is repeated again for the input layer, using the previously calculated delta value of a neuron in the medial layer in place of the error values.

[D2]

These steps are repeated until the delta value for each output and medial neuron has been calculated. The procedure described here is equivalent to running the network backwards by using the error values as an input which is feed to the output neurons while ignoring the activation function.

### Weight Adjustment

Finally, each synaptic weight is adjusted by the product of the learning rate, the input to the synapse, the delta value of the neuron which the synapse connects to and gradient of the activation function at the output of the neuron.

[D3]

The final set of synapses leading to the output layer are adjusted by the same product without the gradient term, as the output neurons do not apply the activation function.

The error propagation and weight adjustment is repeated until the errors reduce below a value acceptable to the designer of the network.

### Learning Rate

This value can fixed, usually around 0.1 to avoid overcompensation when adjusting weights, however the rate can be changed dynamically to achieve better results. For example, the learning rate could be decrease during execution to make rough adjustments at the start then fine tune the weights near end.

## Pseudo code

Below is pseudo code implementing back propagation on a network with one medial layer.

procedure backPropagation():

inputs := getInput()

expected := getExpected(inputs)

actual := runNetwork(inputs)

errors := calculateErrors(expected, actual)

// Calculate the delta values of the medial neurons.

for n := 0 to medialNeuronCount:

sum := 0

for o := 0 to outputCount:

sum := sum + weightsOut[n, o] \* errors[o]

endfor

deltas[n] := sum

endfor

// Adjust the weights between the input and medial neurons

for i := 0 to inputCount:

for n := 0 to medialNeuronCount:

weightsIn[i, n] := weightsIn[i, n] + learningRate

\* inputs[i] \* deltas[n] \* dActivation(medialOut[n])

endfor

endfor

// Adjust the weights between the medial and output neurons

for n := 0 to medialNeuronCount:

for o := 0 to outputCount:

weightsOut[n, o] := weightsOut[n, o] + learningRate

\* medialOut[n] \* errors[o]

endfor

endfor

repeat procedure until inputs exhausted or total error acceptable

endprocedure

# Intuition

The previous section describes how the back propagation algorithm teaches a network, but not why.

As stated before, back propagation applies an input to the network and measures the error between the expected and actual outputs. These errors can be brought down to zero if the network were given the same inputs again by adjusting the weights of synapses leading to the output neurons using the delta rule:

Where is the weight between medial neuron and output , is the error at and is the output of . However, this does not adjust the weights between the input and first medial layer, or between medial layers. So the error is propagated backwards through the layers

# Limitations

Searches for local, not global minimums.

Momentum

# Improving Results

Increasing number of neurons in a hidden layer does not necessarily reduce the error at which the network settles, the network can in fact perform worse if there are too many neurons in a hidden layer.

Use the implementation to create graphs of the average error for some function with varying numbers of hidden layers and neurons in each hidden layer.

2 x 15 medial neurons, rate = 0.5

# References