A Feed-Forward Neural Network

Theodore Evans

7109836

School of Physics and Astronomy

The University of Manchester

Programming in C++ Project Report

May 2012

Abstract

This is a framework for generating feedforward multilayer perceptron-like neural networks. Based on parameters loaded from an external configuration file, a network is generated and trained through a backpropagation supervised learning algorithm to map sets of input variables to desired outputs. After a number of training cycles, the network tests its learned model on input data, this time without feedback. The results are written to file for plotting by external software.

1. Introduction

Artificial neural networks (ANNs) emulate the synaptic network structures found in the nervous systems of most multicellular animals. From their initial development as a means of insight into the workings of biological neural networks, ANNs have since found applications in data analysis and pattern recognition, and play an important role in the field of computational intelligence (AI). This project is an attempt at developing a flexible framework for creating multilayer perceptron-type neural networks, primarily for data classification applications.

2. Theory

The basic unit of the multilayer perceptron network used in this project is an artificial neuron of the type described by the McCulloch–Pitts model. Like its biological counterpart, each artificial neuron connects with a number of other nodes in the network: receiving an input from some and propagating an onward signal to others. Whether or not a single neuron continues to propagate a signal for a given set of inputs $(x_1,\ x_2,\ x_3\ \ldots\ x_n)$ is decided based on the set of *weights* assigned to those inputs $(w_1,\ w_2,\ w_3\ \ldots\ w_n)$. If the weighted sum (*activation function*) of the inputs $(w_1x_1 + w_2x_2 + w_3x_3 + \cdots w_nx_n)$ is above a given threshold τ, then a signal will propagate.

In practice, we would like to include τ as a term in the sum and modify it along with the input weights. This is implemented by defining $\tau \equiv w_bx_b$, and treating x_b like the output from a neuron that always equals -1. The desired response may then be achieved by applying a step-like *transfer function* (e.g. the logistic function) to the activation function.

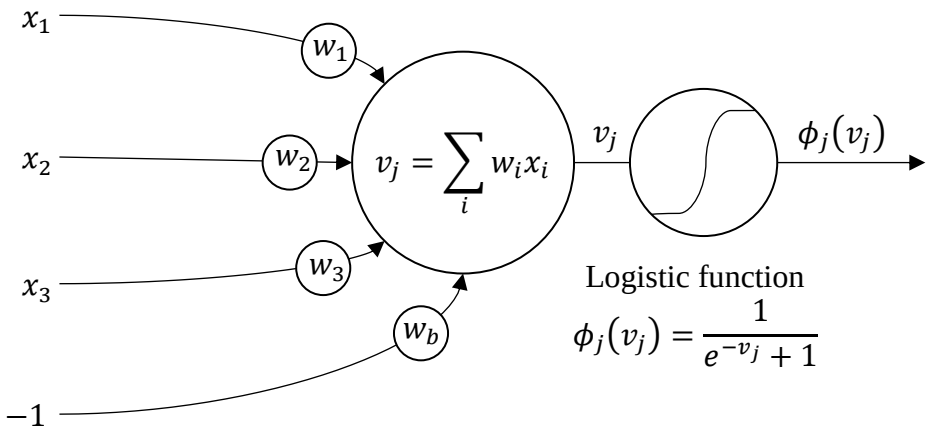


Fig 1. Information flow in a McCulloch–Pitts model neuron.

A network of neurons propagating a signal in this way can perform complex mappings between its inputs and outputs. Moreover, since the network does not require any particular expert knowledge about the nature of the relationship, it is flexible to learn new relationships simply by modifying the network weights.

Here we will only consider a *feedforward* network (no recurrent loops) trained by means of the *backpropagation* algorithm, a form of supervised learning.

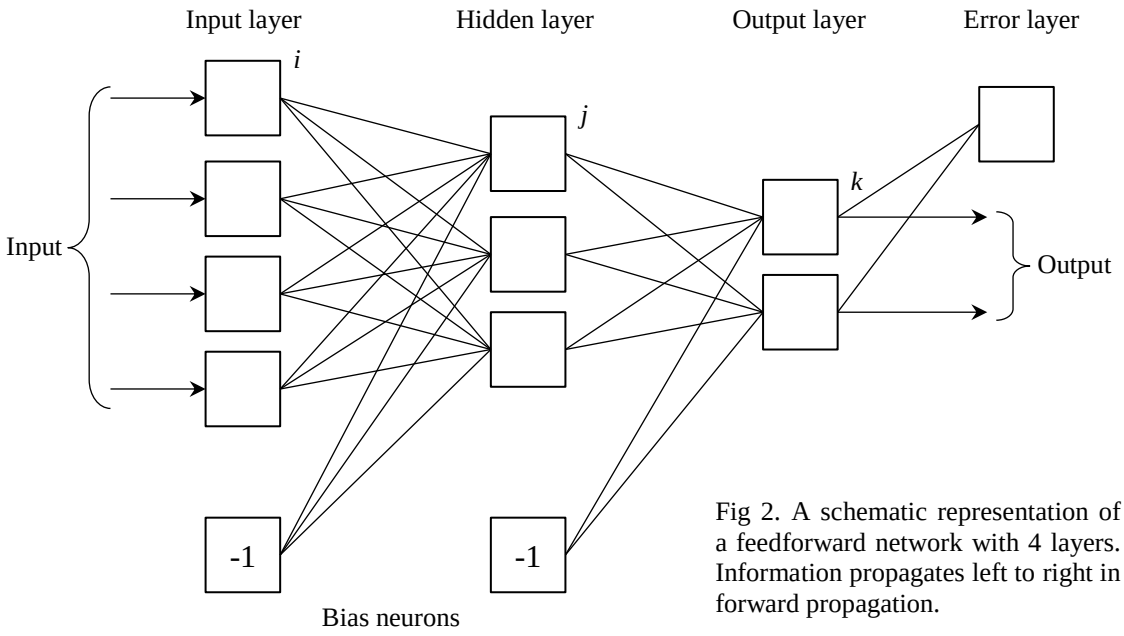


Fig 2. A schematic representation of a feedforward network with 4 layers. Information propagates left to right in forward propagation.

Having initialised the network with a random set of weights, supervised learning proceeds by propagating a signal from initial inputs through the network, and comparing the resulting output values o_i, with the desired values from a training data set t_i. These values are fed through to an additional node in the 'error layer', giving a total network error of

$$\mathcal{E} = \frac{1}{2}\sum_k (t_k - o_k)^2$$

Network weights are then adjusted to minimise this value using *gradient descent*. Without going into too much detail, the required change in a given connection weight between two neurons *j* and *k*, for a particular data point, is given by:

$$\Delta w_{kj} = -\eta \frac{\delta \mathcal{E}}{\delta v_k} \phi_j(v_j)$$

Where $v_{j,k}$ are the activation functions of *j* and *k*, and $\phi_j(v)$ is the transfer function for *j* acting on its activation function to give the total output for that neuron (equivalent to o_j). η is a coefficient denoting the step size, called the *learning rate*. For weights between hidden and output neurons, the derivative of the total error is given by

$$\frac{\delta \mathcal{E}}{\delta v_k} = -(t_k - o_k)\phi_k'(v_k)$$

Where ϕ_k' is the derivation of the activation function with respect to v_k. For weights between hidden and input neurons (*j* and *i*) this becomes more complicated, giving:

$$\frac{\delta \mathcal{E}}{\delta v_j} = \phi_j'(v_j) \cdot \sum_k \frac{\delta \mathcal{E}}{\delta v_k} w_{kj}$$

$$\Delta w_{ji} = -\eta \phi_j'(v_j)\phi_i(v_i) \cdot \sum_k (w_{kj} \cdot \Delta w_{kj})$$

In order to calculate the weight updates for the hidden layer, the changes in weights for the output layer must first be calculated and 'backpropagated' to previous layers. Once the weights have been updated, a set of inputs are again forward propagated through the

network. This process is iterated for each point in the training data until the total error drops below a predetermined threshold. This threshold cannot be arbitrarily low, since 'overfitting' of the model to the training data reduces the network's ability to generalise.

3. Code Implementation

The source code is separated into five files:

`config.h` `config.cpp`	The `Config` class and member functions for loading parameters from the program configuration file.
`neuralnet.h` `neuralnet.cpp`	All the classes and functions necessary for neural network functionality.
`main.cpp`	Containing the main loop and user interface.

3.1. Configuration (`config.h & config.cpp`)

The `Config` class acts as an interface to an .ini configuration file holding all the parameters that define a particular neural network. These parameters are read into two local STL `map` object, where we have defined types for the sake of convenience:

```
typedef std::map<std::string, double> nMap_t;
typedef std::map<std::string, std::string> sMap_t;
```

Numerical parameters are stored in an `nMap_t` container, string parameters in an `sMap_t`, and both are accessed by string keys. Each `Config` object has one of each map type as private member variables, as well as a `const char*` pointer to an array holding the configuration file name, all of which are populated during execution of the parameterised constructor:

```
Config(const char* filename) : filename_(filename)
```

The .ini file designated by `filename` has the format:

```
// abalone.ini, neural net configuration file
// type       name                value
string        dataFilename        abalone1.data
int           numHiddenLayers     1
int           numInputNeurons     10
…
```

On construction, an `ifstream` object is instantiated to read from this file. The type identifiers in the first column both determine which `map` that parameter should be read into, and act as a check for proper formatting. Comment lines prefixed by `//` are discarded. Two public accessor methods, `Config::getStringParameter` and `Config::getNumericalParameter` (one for each map container) are defined to extract parameters by key.

3.2 Neural Network (neuralnet.h & neuralnet.cpp)

The code implementation of the neural network has a hierarchical structure, with tiers linked by vectors of pointers.

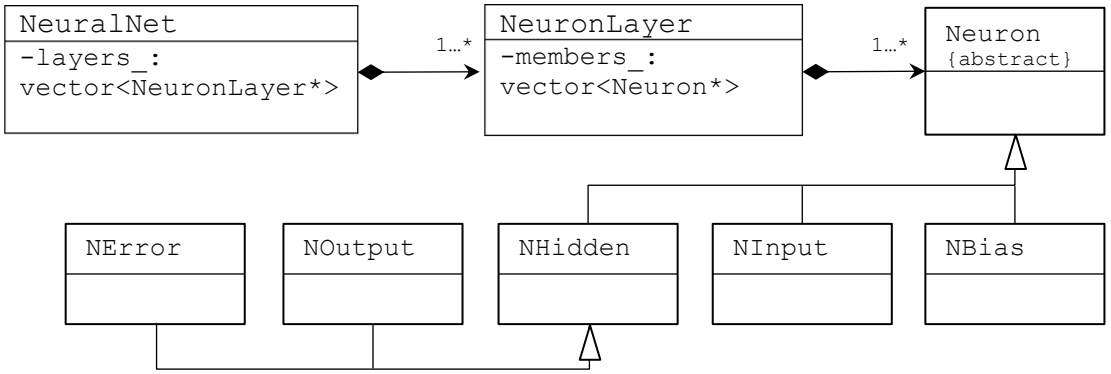


Fig 3. A UML class diagram showing relationships between classes in neuralnet.cpp

`Neuron` is an abstract base class, the main functionality of the network is realised by the derived classes shown in Fig. 3, each with methods specific to the function of individual layers.

3.2.1 Network Initialisation

The creation of the network is a top-down branching process, initiated by the `NeuralNet` constructor:

1. `NeuralNet::NeuralNet(const Config &config)` is called, initialising its private member variables with the corresponding configuration parameters extracted using accessor methods of `config`.
2. The `inputData_` and `trainingData_` vectors are resized in preparation for population by values from the data set. `inputData_` will hold values for input variables, while `trainingData_` will hold the desired network output value(s) during the supervised training phase. Objects refer to this data via pointers to elements of the vector. The values at these vectors are then refreshed as the input stream is updated.
3. `ifstream fetch_` is initialised with the appropriate data file path, and checked for validity. Comment lines are discarded.
4. `NeuronLayer` instances are dynamically created. Each layer constructor takes as arguments: a pointer to the network of which they are a member, an `int` index indicating position in the network, an enumerator indicating layer type, and an `int` indicating the number of nodes in that layer.
5. For each `NeuronLayer` instance created, the associated `members_` vector is populated with the correct number of instances of the class derived from `Neuron` appropriate to that layer type.
6. `initializeNetLinkage()` is called in the `NeuralNet` constructor, iterating through the `layers_` vector to call `initializeLayerLinkage()` from each

layer of the network, which in turn calls `generateLinks(const int n)` from each node in the network.

7. `generateLinks(const int n)` is overridden for slightly different functionality for different neuron classes. In the general case, it populates member vector(s) `sources_` and `sinks_` with pointers to nodes in one or both adjacent layers.
    i. Each `NHidden` and `NOutput` instance is linked to nodes in both posterior and anterior layers.
    ii. The `NError` node is only linked to the previous layer.
    iii. `NInput`, `NBias` do not require linkage; this function defaults to the empty method defined in `Neuron`.

    To simplify this step, each node is passed its layer index as the argument `int n`. This is the only time a node will explicitly need to know the index of any layer; the linkage structure handles all further communication.
8. `randomizeWeights()` is called from within the linking function, resizing the `weights_` vector for each node is to match the `sources_` vector and populating each element with a random `double` between -1 and 1.

The network is now initialised and ready to be trained.

3.2.2 Training the Network

To best understand the training algorithm, it is first necessary to look at the *feedforward* regime of information propagation. This is the process by which a set of input variables are fed through the neural network and integrated by sequential layers to give some kind of output.

A signal is forward-propagated through the network similarly to how network linkage was established previously, initiated by the `NeuralNet::feedforwardNet()` method.

1. `feedforwardNet()` calls `feedforwardLayer()` from each `NeuronLayer` pointed to in the `layers_` vector. These are filled in order from input to output; ensuring layers are activated in the correct order. This then calls `feedforward()` from each node in the network.
2. For instances of `NHidden`, `NOutput` and `NError`, `feedforward()` calls one of more of these three functions:
    i. `updateActivation()` calculates the sum of weights stored in the `weights_` vector multiplied by output values for the corresponding nodes in the `sources_` vector, which are extracted using the `output()` accessor method.

    NB. `NInput::output()` returns a dereferenced value for the pointer to its associated element in the `NeuralNet::inputData_` vector. `NBias::output()` returns a value of -1.
    ii. `updateOutput()` calculates the result of applying the transfer function, defined here in `sigmoid(const double t, const double`

`response)` (where for now, response is simply 1 in all cases) to `activation_`, and stores this in the member variable `output_`.

iii. `updateError()`: In `NOutput` nodes, this calculates the difference between the member `output_`, and the desired output value pointed to in the associated element of `NeuralNet::trainingData_`, storing this value in its error_ member.

`NError::updateError()` calls extract these errors for nodes in the output layer (using the `error()` accessor) and performs a sum of their squares (with a prefactor of 0.5) to give a total error for the system. This value is stored in the `NError::error_` member.

We'll come to the function of `updateError()` in `NHidden` nodes shortly, since it is not used during forward propagation.

3. The signal propagates through the network, giving network outputs accessible by `NOutput::output()`, and a total network error `NError::error()`.

Network training is initiated by calling of the `NeuralNet::trainNetwork()` method.

1. Pointers for the output and error `NeuronLayer` instances, and the single `NError` instance are defined.
2. `Ofstream fout_` is initialised with the output file name.
3. One-off calculations of values relating to data scaling are performed.
4. `updateInput()` is called, populating `inputData_` and `trainingData_` with the appropriate number of data file entries extracted from fetch_.
5. The `feedforward()` method is propagated through the network.
6. Output values for each node in the output layer (extracted by `NOutput::output()`), and the total network error (extracted by `NError::error()`) are inserted into `fout_` for writing to file.
7. `NeuralNet::updateNetWeights()` is called for each layer, calling the `updateWeights()` method for each node in the network in the same way that `feedforward()` was, except in the opposite direction; from output to input.
    a. `NOutput::updateWeights()` modifies the values held in its weights_ vector according to the backpropagation formula given in section 2, also storing the calculated values of Δw_{kj} in a member vector deltas_.
    b. `NHidden::updateWeights()` extracts these delta values from the output layer and uses them to update the values in its weights_ member, according to the backpropagation formulae.

    This step is iterated, alternately calling the `updateNetWeights()` and `feedforwardNet()` methods, until the network error falls below a threshold defined in the configuration.
8. Steps 4-7 are repeated over as many training examples as are specified in the configuration.
9. Steps 4-6 are then iterated over the remaining entries in the data file to test the fitness of the resulting model.

3.2.3 Additional features

3.2.3a Momentum

When updating the weights of each node, a fraction (~0.1) of the Δw from the previous iteration is included in each update. This is a technique called *momentum*, and improves the rate of convergence, whilst reducing oscillations around error minima.

3.2.3b Linear transfer function overrides

For continuous output data (rather than discreet classification) the standard `output()` and `updateWeights()` functions may be overridden in NOutput by methods using a simple linear transfer function (whereby `output()` just returns the `activation_` member, and the transfer derivative used in backpropagation is equal to 1) instead of a sigmoid.

4. Results

The training data used here was a multivariate data set from the UCI Machine Learning Repository, containing 4177 examples. More information on the dataset is available at

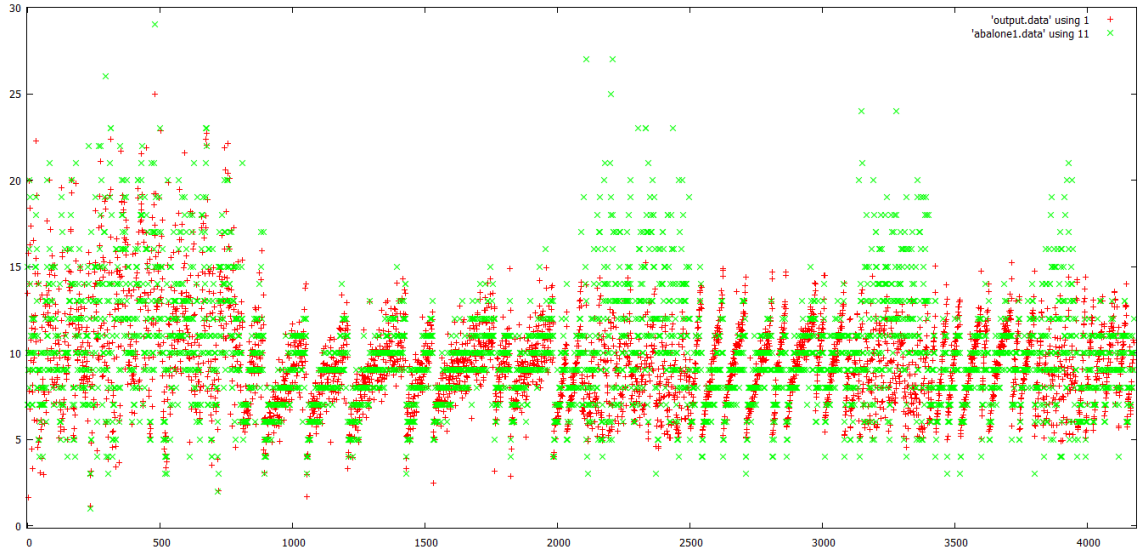
http://archive.ics.uci.edu/ml/datasets/Abalone

Each example consists of 8 physical attributes of abalone shellfish. The data required little processing in order to be processed by the network: the first attribute was of the form of a single character indicating Male, Female or Infant abalone. This was modified using a macro to convert these to binary values:

`M → 1 0 0, F → 0 1 0, I → 0 0 1`

The total number of input variables was then 10, with the 11th entry on each line being the age of the abalone, used as the desired network output for the purpose of training. For the data set *abalone1.data* and configuration file *abalone.ini* (see appendix), and using a linear transfer function for output neurons, a plot of results was attained.

Gnuplot: `plot 'output.data' using 1, 'abalone1.data' using 11)`



Neural network output, showing actual data in green, and points generated by the network in red. The *y*-axis is age in years; the *x*-axis is the example number. Training ends at example 2000, the remaining ~2000 points are produced by the trained model with no feedback of real results.

5. Conclusion

The network performed reasonably well at learning a mapping function to fit the example data set. The objective of the project was to build a flexible framework for such neural networks, with the ability to manage different types of data set with little or no qualitative modification. In theory, the program could be fed a classification-type data set and process it will only minimal input conditioning.

The rationale behind the object orientated approach to this project, and the use of a graph representation of inter-neuron relationships (as opposed to a matrix representation) was to allow for the program to be extended to simulate more complex structures; e.g. recurrent networks, Markov chains, and other representations that do not follow a strict layer-based feedforward architecture. The benefit of this approach is also demonstrated by the ease with which an alternative transfer function may be toggled simply by commenting out the overriding methods, defaulting to inherited functionality.

Improvements to the program would include:

- Implementation of smart pointers.
- Design of a fuzzy control system to modify configuration parameters at runtime in response to training progress.
- Save/load/display function for network weight matrices.
- Automatic writing of Gnuplot scripts to file.
- Integrated data preconditioning.

config.h

```
#ifndef CONFIG_H_INCLUDED
#define CONFIG_H_INCLUDED

#include <map>
#include <string>

namespace config
{
    typedef std::map<std::string, double> nMap_t;
    typedef std::map<std::string, std::string> sMap_t;

    class Config
    {
        friend std::ostream & operator<<(std::ostream &os, const Config &config);

        private:
            const char* filename_;
            nMap_t nParameters_;
            sMap_t sParameters_;

        public:
           ~Config();
            Config() {}
            Config(const char* filename);

            bool loadParameters();

            double getNumericalParameter(const std::string parameterName)   const;
    std::   string getStringParameter(const std::string parameterName)      const;
    };

    std::   ostream & operator<<(std::ostream &os, const Config &config);
};

#endif // CONFIG_H_INCLUDED
```

config.cpp

```
#include <iostream>
#include <fstream>
#include <string>
//#include <map>    // included in config.h

#include "config.h"

using namespace std;
using namespace config;

Config::~Config() {
    nParameters_.clear();   // Clear map objects
    sParameters_.clear();   //
}

Config::Config(const char* filename) : filename_(filename)
{
    bool loadSuccess = loadParameters();
    if (!loadSuccess)
    {
        string errorOpenConfig("Could not open configuration file ");
        errorOpenConfig += filename;
        throw errorOpenConfig;
    }

    cout << filename << " loaded." << endl;
}

bool Config::loadParameters()
{
    ifstream fetch(filename_);

    if (!fetch) return false;

    string parameterType(""), parameterName(""), sParameterValue("");
    double nParameterValue(0);

    while ( fetch.good() )
    {
        fetch >> parameterType;     // each line has format:
        fetch >> parameterName;     // [type] [name] [value]

        if  (parameterType  == "int" || parameterType == "double")
        {
            fetch >> nParameterValue;
            nParameters_[parameterName] = nParameterValue;
        }

        else if (parameterType  == "string")
        {
            fetch >> sParameterValue;
            sParameters_[parameterName] = sParameterValue;
        }

        else if (parameterType == "//") {
            fetch.ignore( 256, '\n');} // ignore comments

        else
        {   // throw an error for improper type or bad formatting
            string errorBadType(parameterName);
            errorBadType += "has invalid type \'";
            errorBadType += parameterType;
            errorBadType += "\'";
            throw errorBadType;
```

```
          }
    }

    return true;
}

double Config::getNumericalParameter(string parameterName) const
{
    if  ( nParameters_.find(parameterName) == nParameters_.end() )
    {
        string errorBadParameter("Invalid parameter \'");
        errorBadParameter += parameterName;
        errorBadParameter += "\'";
        throw errorBadParameter;
    }

    else return nParameters_.find(parameterName)->second;
}

string Config::getStringParameter(string parameterName) const
{
    if  ( sParameters_.find(parameterName) == sParameters_.end() )
    {
        string errorBadParameter("Invalid parameter \'");
        errorBadParameter += parameterName;
        errorBadParameter += "\'";
        throw errorBadParameter;
    }

    else return sParameters_.find(parameterName)->second;
}

std::ostream & config::operator<<(std::ostream &os, const Config &rhs)
{
    for (sMap_t::const_iterator sMapIt = rhs.sParameters_.begin(); sMapIt !=
rhs.sParameters_.end(); ++sMapIt) {
        os << sMapIt->first << ":  '" << sMapIt->second << "'" << endl;
    }

    for (nMap_t::const_iterator nMapIt = rhs.nParameters_.begin(); nMapIt !=
rhs.nParameters_.end(); ++nMapIt) {
        os << nMapIt->first << ":  " << nMapIt->second << endl;
    }

    return os;
}
```

config.cpp

```
#ifndef NEURALNET_H_INCLUDED
#define NEURALNET_H_INCLUDED

//#include <fstream>

namespace neuralnet
{
    enum LayerType {
        INPUT,
        BIAS,
        HIDDEN,
        OUTPUT,
        ERROR
    };

    class Neuron;
    class NeuronLayer;

    class NeuralNet
    {
        friend std::ostream & operator<<(std::ostream &os, const NeuralNet &net);

        protected: // inherited by NeuralNetIO
    std::   string dataFilename_;
    std::   string outputFilename_;
            size_t numInputNeurons_ ;
            size_t numHiddenLayers_ ;
            size_t numHiddenNeurons_;
            size_t numOutputNeurons_;
            size_t numTrainExamples_;
            size_t divergenceThreshold_;
            double learningRate_;
            double momentum_;
            double errorThreshold_;
            double dataScaleFactor_;

            size_t dataRow_;
            double dataBuffer_;

    std::   ifstream fetch_;
    std::   ofstream fout_;

    std::   vector<double>  inputData_;
    std::   vector<double>  trainingData_;

    std::   vector<NeuronLayer*> layers_;

        public:
            ~NeuralNet();
            NeuralNet(const config::Config &config);

            double* getInputPointer(const int index);  // ***
            double* getOutputPointer(const int index);

            bool initializeNetLinkage() const;

            void feedforwardNet()       const;
            void updateNetWeights()  const;

            bool loadData();
            bool updateInput();

            void trainNetwork();
            void runNetwork();

            NeuronLayer* operator[](const int layerIndex);
    };
```

```
    };
     std::   ostream & operator<<(std::ostream &os, const NeuralNet &net);

    ////////////////////////////////////////////////

    class NeuronLayer
    {
        friend std::ostream & operator<<(std::ostream &os, const NeuronLayer &layer);

        private:
            NeuralNet* network_;
            int index_;
            LayerType layerType_;

    std::   vector<Neuron*> members_;
    std::   vector<Neuron*>::iterator membersIt_;

        public:
            ~NeuronLayer();
            NeuronLayer() {}
            NeuronLayer(NeuralNet* net, const int index, const LayerType layerType, const int
numMembers);

            int         getIndex()   const;
            size_t      getSize()    const;
            LayerType   getType()    const;
            NeuralNet*  getNetwork() const;

            Neuron* operator[](const int neuronIndex) const;

            bool initializeLayerLinkage()   const;

            void feedforwardLayer();

            bool updateLayerWeights(const double learningRate, const double momentum)
const;
    };

    std::   ostream &operator<<(std::ostream &os, const NeuronLayer &layer);

    ////////////////////////////////////////////////

    class Neuron
    {
        friend std::ostream & operator<<(std::ostream &os, const Neuron &neuron);

        protected:
            int ID_, index_;
            neuralnet::NeuronLayer* layer_;
            double activation_, output_, error_;

        public:
            int generateID();
            int ID()    const;

            virtual ~Neuron() {}

            Neuron();
            Neuron(neuralnet::NeuronLayer* layer);

    virtual std::string type()      const = 0;

            int     layerIndex()    const;

    virtual double  activation()    const;
    virtual double  output()        const;
    virtual double  error()         const;
```

```
 virtual void    feedforward() {}

 virtual double  getWeight(const int index) {return 0;}
 virtual double  getDelta(const int index) {return 0;}

 virtual bool    updateWeights(const double learningRate, const double momentum);
 virtual void    generateLinks(const int n) {}

 };

 std::   ostream & operator<<(std::ostream &os, const Neuron &neuron);

 class NHidden : virtual public Neuron
 {
     protected:
 std::   vector<Neuron*> sources_;
 std::   vector<Neuron*> sinks_;
 std::   vector<double> weights_;
 std::   vector<double> deltas_;

     public:
         ~NHidden() {}

         NHidden();
         NHidden(neuralnet::NeuronLayer* layer);

virtual  void generateLinks(const int n);
         void randomizeWeights();

 std::   string type()       const;

         void feedforward();

 virtual void updateActivation();
 virtual void updateOutput();
 virtual void updateError();

         double getWeight(const int index);
         double getDelta(const int index);

         bool updateWeights(const double learningRate, const double momentum);
 };

 class NOutput : public NHidden
 {
     private:
         double* data_;

     public:
         ~NOutput() {}

         NOutput() {}
         NOutput(neuralnet::NeuronLayer* layer, double* const data);

 std::   string type()  const;

         void feedforward();
         void updateError();

         // linear transfer functions
         void updateOutput();
         bool updateWeights(const double learningRate, const double momentum);

 };

 class NError : public NHidden     // passive nodes that set the activation bias
 {
```

```
        public:
            ~NError() {}

            NError() {}
            NError(neuralnet::NeuronLayer* layer) : Neuron(layer) {}

            void generateLinks(const int n); // sources only

    std::   string type()  const;

            void feedforward();
            void updateError();

    };

    class NInput : public Neuron
    {
        private:
            double* data_;

        public:
            ~NInput() {}

            NInput() {}
            NInput(neuralnet::NeuronLayer* layer, double* const data);

    std::   string type()   const;
            double output() const;

    };

    class NBias : public Neuron     // passive nodes that set the activation bias
    {
        private:

        public:
            ~NBias() {}

            NBias() {}
            NBias(neuralnet::NeuronLayer* layer) : Neuron(layer) {}

    std::   string type()   const;
            double output() const;
    };

    ////////////////////////////////////////////////

    double sigmoid(const double t, const double response);
    double sigmoidDerivative(const double t, const double response);
    double tanhDerivative(const double t);
}

#endif // NEURALNET_H_INCLUDED
```

neuralnet.cpp

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <string>
#include <map>
#include <cmath>
#include <cstdlib>

#include "config.h"
#include "neuralnet.h"

using namespace std;
using namespace neuralnet;
using namespace config;


//NeuralNet
    NeuralNet::~NeuralNet()
    {
        for (vector<NeuronLayer*>::iterator vecIt = layers_.begin(); vecIt < layers_.end();
++vecIt)
        {
            delete (*vecIt);
        }
        inputData_.clear();
        trainingData_.clear();
    }

    NeuralNet::NeuralNet(const Config &config) :

        // initialise parameters from static config variables
        dataFilename_    ( config.getStringParameter("dataFilename")),
        outputFilename_  ( config.getStringParameter("outputFilename")),
        numInputNeurons_ ( (size_t)config.getNumericalParameter("numInputNeurons")),
        numHiddenLayers_ ( (size_t)config.getNumericalParameter("numHiddenLayers")),
        numHiddenNeurons_( (size_t)config.getNumericalParameter("numHiddenNeurons")),
        numOutputNeurons_( (size_t)config.getNumericalParameter("numOutputNeurons")),
        numTrainExamples_( (size_t)config.getNumericalParameter("numTrainExamples")),
        divergenceThreshold_ ((size_t)config.getNumericalParameter("divergenceThreshold")),
        learningRate_    ( config.getNumericalParameter("learningRate")),
        momentum_        ( config.getNumericalParameter("momentum")),
        errorThreshold_  ( config.getNumericalParameter("errorThreshold")),
        dataScaleFactor_ ( config.getNumericalParameter("dataScaleFactor")),
        dataRow_(0)
    {
        // initialise associated input/output object
        inputData_.resize(numInputNeurons_);
        trainingData_.resize(numOutputNeurons_);

        cout << "Loading " << dataFilename_ << "... ";
        bool loadSuccess = loadData();
        if (!loadSuccess)
        {
            string errorOpenData("Could not open data file ");
            errorOpenData += dataFilename_;
            throw errorOpenData;
        }
        cout << "done" << endl;

        cout << "\nCreating network...\n";
        // NeuronLayer constructors take the arguments (network pointer, layer index, layer
type, number of members)
        // Input layer
```

```
        cout << "\tGenerating input layer... ";
        layers_.push_back(new NeuronLayer(this, 0, INPUT, numInputNeurons_));
        cout << "done\n";

        // hidden layer(s)
        cout << "\tGenerating hidden layers... ";
        for (size_t i(0); i < numHiddenLayers_; ++i) {
            layers_.push_back(new NeuronLayer(this, 1 + i, HIDDEN, numHiddenNeurons_));
        }
        cout << "done\n";

        // output layer
        cout << "\tGenerating output layer... ";
        layers_.push_back(new NeuronLayer(this, numHiddenLayers_ + 1, OUTPUT,
numOutputNeurons_));
        cout << "done\n";

        // error layer
        cout << "\tGenerating error layer... ";
        layers_.push_back(new NeuronLayer(this, numHiddenLayers_ + 2, ERROR, 1));
        cout << "done\n\n";

        cout << "Initializing synaptic connections...\n";
        bool linkingSuccess = initializeNetLinkage();
        if (!linkingSuccess) throw "Could not generate synaptic links.";
    }

    bool NeuralNet::initializeNetLinkage() const
    {
        if (layers_.empty()) return false;
        for (size_t i(0); i < layers_.size(); ++i)
        {
            cout << "\tLayer " << i << "... ";
            layers_[i]->initializeLayerLinkage();
            cout << "done\n";
        }
        return true;
    }

    bool NeuralNet::loadData()
    {
        fetch_.open(dataFilename_.c_str());

        if (!fetch_) return false;

        while (fetch_.peek() == '#') fetch_.ignore(256, '\n'); // ignore comments
        return true;
    }

    bool NeuralNet::updateInput() // add error checking!
    {
        ++dataRow_;
        for (size_t i(0); i < numInputNeurons_; ++i)
        {
            fetch_ >> dataBuffer_;
            inputData_[i] = dataBuffer_;
        }

        for (size_t i(0); i < numOutputNeurons_; ++i)
        {
            fetch_ >> dataBuffer_;
            trainingData_[i] = dataScaleFactor_ * dataBuffer_;
        }

        if (!fetch_.good()) return false;
        return true;
```

```
    void NeuralNet::feedforwardNet() const
    {
        for (size_t i(0); i < layers_.size(); ++i) {
            layers_[i]->feedforwardLayer();
        }
    }

    void NeuralNet::updateNetWeights() const
    {
        for (size_t i(layers_.size() - 1); i > 1; --i) {
            layers_[i]->updateLayerWeights(learningRate_, momentum_);
        }
    }

    double* NeuralNet::getInputPointer(const int index)  { return &inputData_[index];  }
    double* NeuralNet::getOutputPointer(const int index) { return &trainingData_[index]; }

    void NeuralNet::trainNetwork()
    {
        NeuronLayer* errorLayer = layers_[numHiddenLayers_ + 2];
        NeuronLayer* outputLayer = layers_[numHiddenLayers_ + 1];

        Neuron* errorNode = (*errorLayer)[0];

        fout_.open(outputFilename_.c_str());
        if (!fout_.good()) throw "Could not open output file.";

        size_t iteration(0);
        double error(0);
        // 1-off calculations
        double squareScaleFactor = dataScaleFactor_ * dataScaleFactor_;
        double reciprocalScaleFactor = 1 / dataScaleFactor_;
        double recipSquareScaleFactor = 1 / squareScaleFactor;
        double scaledErrorThreshold = squareScaleFactor * errorThreshold_;

        for (size_t n(0); n < numTrainExamples_; ++n)
        {
            updateInput();
            cout << "Training example " << dataRow_ << " of " << numTrainExamples_ << "... ";
            feedforwardNet();

            error = errorNode->error();

            for (size_t i(0); i < numOutputNeurons_; ++i)
            {
                fout_   << (*outputLayer)[i]->output() * reciprocalScaleFactor << " ";
            }
            fout_   << error * recipSquareScaleFactor << endl;

            iteration = 0;
            while (error > scaledErrorThreshold)
            {
                ++iteration;
                updateNetWeights();
                feedforwardNet();
                error = errorNode->error();
                if (iteration >= divergenceThreshold_) throw "Convergence failure. Try
reducing learning rate.";
            }
            cout << "done in " << iteration << " steps.\n";
        }

        while (updateInput())   //
        {
            for (size_t i(0); i < numOutputNeurons_; ++i)
            {
```

```
                fout_   << (*outputLayer)[i]->output() * reciprocalScaleFactor << " ";
            }
            fout_   << error * recipSquareScaleFactor << endl;

            feedforwardNet();
            error = errorNode->error();
        }
    }

    // accessor for layers* in net
    NeuronLayer* NeuralNet::operator[](const int layerIndex) { return layers_[layerIndex]; }

    ostream & neuralnet::operator<<(ostream &os, const NeuralNet &rhs)
    {
        for (size_t i(0); i < rhs.layers_.size(); ++i) {
            os << *rhs.layers_[i];
        }
        return os;
    }

    ////////////////////////////////////////////////

//NeuronLayer

    NeuronLayer::~NeuronLayer()
    {
        for (vector<Neuron*>::iterator vecIt = members_.begin(); vecIt < members_.end();
++vecIt) {
            delete (*vecIt);
        }
        members_.clear();
    }

    // parameterised constructor
    NeuronLayer::NeuronLayer(NeuralNet* network, const int index, const LayerType layerType,
const int numMembers)
                  : network_(network), index_(index), layerType_(layerType)
    {
        switch (layerType_)
        {

            case INPUT: // create an input layer of N input neurons + 1 bias neuron
                for (int i(0); i < numMembers; ++i) {
                    members_.push_back(new NInput(this, network->getInputPointer(i) ));//,
data[i]));
                }
                members_.push_back(new NBias(this));
                break;

            case HIDDEN: // create a hidden layer of N hidden neurons + 1 bias neuron
                for (int i(0); i < numMembers; ++i) {
                    members_.push_back(new NHidden(this));
                }
                members_.push_back(new NBias(this));
                break;

            case OUTPUT: // create an output layer of N output neurons only
                for (int i(0); i < numMembers; ++i) {
                    members_.push_back(new NOutput(this, network->getOutputPointer(i)));
                }
                break;

            case ERROR:
                for (int i(0); i < numMembers; ++i) {
                    members_.push_back(new NError(this));
                }
```

```
                break;

            default:
                throw "Bad node type.";
                break;
        }

    }

    int         NeuronLayer::getIndex()   const { return index_;          }
    size_t      NeuronLayer::getSize()    const { return members_.size(); }
    LayerType   NeuronLayer::getType()    const { return layerType_;      }
    NeuralNet*  NeuronLayer::getNetwork() const { return network_;        }

    Neuron*     NeuronLayer::operator[](const int neuronIndex) const
    {
        return members_[neuronIndex];
    }

    bool NeuronLayer::initializeLayerLinkage() const
    {
        if (members_.empty()) return false;
        for (size_t i(0); i < members_.size(); ++i) {
            members_[i]->generateLinks(i);
        }
        return true;
    }

    void NeuronLayer::feedforwardLayer()
    {
        for (size_t i(0); i < members_.size(); ++i) {
            members_[i]->feedforward();

        }
    }
    bool NeuronLayer::updateLayerWeights(const double learningRate, const double momentum)
const
    {
        if (members_.empty()) return false;
        for (size_t i(0); i < members_.size(); ++i) {
            members_[i]->updateWeights(learningRate, momentum);
        }
        return true;
    }

    ostream & neuralnet::operator<<(ostream &os, const NeuronLayer &rhs)
    {
        os << setprecision(3) << fixed;
        os << "\nNeuron layer " << rhs.index_ << ", size " << rhs.members_.size();

        os << "\nMembers:\n";
        for (size_t i(0); i < rhs.members_.size(); ++i) {
            cout << i << " ";
            os << *rhs.members_[i];
        }
        return os;
    }

    ////////////////////////////////////////////////

//Neuron ABC

    Neuron::Neuron() {}
    Neuron::Neuron(NeuronLayer* layer) : ID_(generateID()), index_(0), layer_(layer),
                                         activation_(0), output_(0), error_(0)
    {}
```

```
    int Neuron::generateID() // DEBUGGING ONLY
    {
        static int staticID;
        return staticID++;
    }

    int     Neuron::ID()         const { return ID_; } // DEBUGGING ONLY
    int     Neuron::layerIndex() const { return layer_->getIndex(); } // ?

    double  Neuron::activation() const { return activation_; }
    double  Neuron::output()     const { return output_;     }
    double  Neuron::error()      const { return error_;      }

    bool    Neuron::updateWeights(const double learningRate, const double momentum) {return
true;}

    // DEBUGGING ONLY
    ostream & neuralnet::operator<<(ostream &os, const Neuron &rhs)
    {
        os << "[" << rhs.ID() << "] "    << rhs.type() << " (" << rhs.layerIndex() << ")"
            << "; Activation: "          << rhs.activation()
            << "; Output: "              << rhs.output()
            << "; Error: "               << rhs.error()
            << endl;
            return os;
    }

//NHidden

    NHidden::NHidden() {}
    NHidden::NHidden(NeuronLayer* layer) : Neuron(layer)
    {
    }

    void NHidden::generateLinks(const int n) // populate vectors with pointers to anterior &
posterior nodes
    {
        index_ = n;
        NeuralNet*   network     = (*layer_).getNetwork();        // get a pointer to the
neural network object

        NeuronLayer* sourceLayer = (*network)[layerIndex() - 1];
        sources_.resize(sourceLayer->getSize());

        for (size_t i(0); i < sources_.size(); ++i)
        {
            sources_[i] = (*sourceLayer)[i];            // assign link to source nodes
        }

        NeuronLayer* sinkLayer   = (*network)[layerIndex() + 1];
        sinks_.resize(sinkLayer->getSize());

        for (size_t i(0); i < sinks_.size(); ++i)
        {
            sinks_[i] = (*sinkLayer)[i];                // assign link to sink node
        }

        randomizeWeights();
    }

    void NHidden::randomizeWeights()
    {
        weights_.resize(sources_.size());
        deltas_.resize(sources_.size());

        for (size_t i(0); i < sources_.size(); ++i) {
```

```
            weights_[i] = 2 * ( (double)rand() / ( (double)RAND_MAX + 1.0) ) - 1.0; // random
number between -1 and 1
            deltas_[i] = 0;
        }
    }

    string NHidden::type()      const { return "Hidden"; }

    void NHidden::feedforward()
    {
        updateActivation();
        updateOutput();
    }

    void NHidden::updateActivation()
    {
        double weightedSum(0);
        for (size_t i(0); i < sources_.size(); ++i) {
            weightedSum += weights_[i] * sources_[i]->output(); // Sum over w(i)*x(i)
        }
        activation_ = weightedSum;
    }

    void NHidden::updateOutput()
    {
        output_ = sigmoid(activation(), 1);
    }

    void NHidden::updateError()
    {
        double error(0);
        for (size_t i(0); i < sinks_.size(); ++i)
        {
            error += sinks_[i]->getDelta(index_) * sinks_[i]->getWeight(index_);
        }
        error_ = error;
    }

    double NHidden::getWeight(const int index) { return weights_[index]; }
    double NHidden::getDelta(const int index) { return deltas_[index]; }

    bool NHidden::updateWeights(const double learningRate, const double momentum)
    {
        updateError();
        for (size_t i(0); i < sources_.size(); ++i)
        {
            double previousDelta = deltas_[i];
            deltas_[i] = learningRate * error() * sigmoidDerivative(activation(), 1) *
sources_[i]->output();
            weights_[i] += deltas_[i] + momentum * previousDelta;
        }
        return true;
    }

//NOutput

    NOutput::NOutput(NeuronLayer* layer, double* const data)
            : Neuron(layer), data_(data)
    {
    }

    string NOutput::type()   const { return "Output"; }

    void NOutput::feedforward()
    {
        updateActivation();
        updateOutput();
```

```
        updateError();
    }

    void NOutput::updateError()   // error function, (t - o)
    {
        error_ = (*data_ - output_);
    }

    // Linear transfer functions
    void NOutput::updateOutput()
    {
        output_ = activation();
    }

    bool NOutput::updateWeights(const double learningRate, const double momentum)
    {
        updateError();
        for (size_t i(0); i < sources_.size(); ++i)
        {
            double previousDelta = deltas_[i];
            deltas_[i] = learningRate * error() * sources_[i]->output();
            weights_[i] += deltas_[i] + momentum * previousDelta;
        }
        return true;
    }

//NError

    void NError::generateLinks(const int n)
    {
        index_ = n;

        NeuralNet*   network     = (*layer_).getNetwork();        // get a pointer to the
neural network object
        NeuronLayer* sourceLayer = (*network)[layerIndex() - 1];  // get the address of the
previous

        sources_.resize(sourceLayer->getSize());
        for (size_t i(0); i < sources_.size(); ++i) {
            sources_[i] = (*sourceLayer)[i];
        }

        randomizeWeights();
    }

    string NError::type()   const { return "Error"; }

    void NError::feedforward()
    {
        updateError();
    }

    void NError::updateError()
    {
        double error(0);
        double totalError(0);

        for (size_t i(0); i < NHidden::sources_.size(); ++i)
        {
            error = sources_[i]->error();
            totalError += error * error / 2; // Sum[ (t - o)^2 /2 ]
        }
        error_ = totalError;
    }

//NInput
```

```
    NInput::NInput(neuralnet::NeuronLayer* layer, double* const data) : Neuron(layer),
data_(data)
    {}

    string NInput::type()   const { return "Input"; }
    double NInput::output() const { return *data_; }

//NBias

    string NBias::type()    const { return "Bias";  }
    double NBias::output()  const { return -1;      }

    ////////////////////////////////////////////////

double neuralnet::sigmoid(const double t, const double response)
{
    return  1 / (1 + exp(-t / response) );
}

double neuralnet::sigmoidDerivative(const double t, const double response)
{
    double exponent = exp(t/response);
    return exponent/(response * (1 + exponent) * (1 + exponent));
}
```

main.cpp

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <string>
#include <cctype>
#include <ctime>

#include "config.h"
#include "neuralnet.h"

using namespace std;
using namespace neuralnet;
using namespace config;

int main()
{
    srand((unsigned)time(0));   // time seed for random number generator

    cout << "Persephone v0.1 2012: An artificial neural network, by Theo Evans" << endl;
    cout << "Enter configuration file name:" << endl << ">";
    char configFilename[64];
    cin >> configFilename;

    try
    {
        Config config(configFilename);

        string selection;
        bool finished(false);

        while(!finished)
        {
            selection.clear();
            cout << "\n[C]reate neural network\n[P]rint configuration\n[Q]uit\n>";
            cin >> selection;

            switch ( toupper(selection[0]) ) // get the first letter of input
            {
                case 'C':
                {
                    NeuralNet myNetwork(config);
                    cout << "\nNetwork created, press any key to begin training.\n";
                    cin.get();
                    cin.ignore();

                    myNetwork.trainNetwork();

                    finished = true;
                    break;
                }

                case 'P':
                {
                    cout << config;
                    break;
                }

                case 'Q':
                {
                    finished = true;
                    break;
                }
```

```
                default:
                {
                    cout << "Invalid selection." << endl;
                    break;
                }
            }
        }

        cout << "Press any key to exit.\n";
        cin.get();
    }
    catch(const string &err)
    {
        cout << "\nException: " << err << endl;
    }
    catch(exception &ex)
    {
        cout << "\nException: " << ex.what() << endl;
    }

    return 0;
}
```