A Deep Learning Approach to Thyroid Disease Classification

Endri Kastrati
Computer Science Department
Monash University
Wellington Road
Victoria 3800, Australia
email: ekas6@student.monash.edu

Abstract

The diagnosis of thyroid diseases is a very difficult and complicated process that depends on a variety of medical factors and empirical tests. This report presents an application of artificial neural networks for assisting a more precise identification of diagnostics regarding thyroid diseases. An introduction to the project objectives, requirements and constraints is initially outlined. A voluminous introduction to the theoretical and mathematical aspects of ANN is given and a detailed walk through of physical implementation of the ANN library is provided. The discussion then focuses on the correctness of the application and it's computational properties in terms of space and time complexity. The report continues with the presentation of the thyroid disease data set that is used to train the neural network and the configuration parameters that are selected to obtain an accurate classifier. A minimalist web application is introduced that abstracts away all of the complicated details and provides a user-friendly interface for interacting with the trained thyroid disease classifier. Finally, it is concluded that the submitted project can be used to enhance the diagnostic skills of medical experts.

Keywords: Artificial neural networks, deep learning, thyroid disease classification, back-propagation, gnu scientific library, the C programming language, gradient descent, momentum optimisation, curve fitting.

1 Introduction

The accumulation of massive quantities of medical data that are rich in features and content has led to the deployment of supervised learning algorithms that are able to extract patterns and generalisations from data-sets and provide useful insights that would have been otherwise impossible to reveal with just the sheer computational power of the human brain. Though there are many supervised learning algorithms, this report specifically concentrates on a particular kind of a supervised learning algorithm known as multi-layer perceptron network, and provides a thorough and detailed walk-through of the process of implementing such an abstract concept into a fully functional computer program. Before dwelling into the theoretical properties and technical aspects it is important to provide a proper introduction to multi-layer perceptrons and their operating principle and provide a comprehensive understanding of how such a system obtains patterns or generalisations from data-sets. Once a thorough understanding has been obtained, the consequent step is the translation of the abstract notion of a neural network to a computer program implementation. Such a program must be able to receive a data-set, apply the necessary transformations and fetch it into the neural network. The neural network in return, must be able to run the back-propagation algorithm in order to extract some form of pattern or generalizer and save and load the trained model on request. All of these procedures have to be optimal in performance, the library itself must be flexible and portable as a standalone package and meet certain memory constraints. All of these matters will be properly elaborated and articulated in the following sections of the report.

2 Artificial Neural Networks

An artificial neural network [1] is an abstract concept of a computational model that can extract patterns from a given data-set or figures out some kind of encoded generalizer. Such a model can be broken down into three main components [2]. The first component, namely the hidden layer, is responsible for receiving input signals from the external environment. The second component, namely the hidden layers consist of a set of neurons which are responsible for the pattern extraction or encoded generalisation. The third component, namely the output layer also consists of a set of neurons which generate the corresponding output signals associated with the initial given input stimulus. The components in Illustration 1, show an neural network configuration comprised of an input layer that receives three input signals, two hidden layers with four neurons per layer and an output layer with a single neuron. There are many different types of ANN that come in different forms and shapes, but the most frequent one is the multi-layer perceptron network.

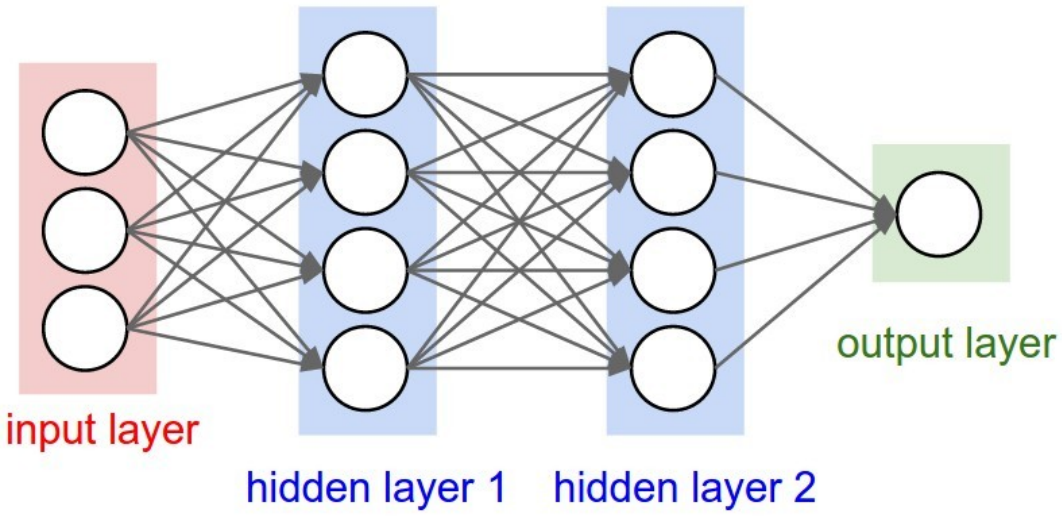


Illustration 1: A 3-layered feed-forward neural network (2 hidden + output)
Taken from http://cs231n.github.io/neural-networks-1/

2.1 Operating principle of the ANN

Assuming for convenience that the training process has been completed, the process of fetching input signals and retrieving the corresponding output signals from the neural network is called forward propagation. Each given input signal is forward propagated layer by layer towards the output layer. In greater depth, the output signals of the neurons from the previous layer will be fetched as inputs to the neurons of the next layer until the output layer has been reached and the desired output signals have been obtained. By observing Illustration 2, we can see that propagation always flows in a single direction during the forward propagation process, that is from the input layer towards the output layer. While the input/output layers are used for fetching and retrieving output signals, the hidden layers are responsible for extracting patterns and encoding generalisations regarding the system behaviour.

Though for all sets of functions that meet certain properties [3], there exists an optimal topological configuration, no method or algorithm has been discovered that calculates the optimal configuration topology for all such functions. A common practice is to start with a single hidden layer that contains a single neuron and keep adding neurons and layers until the optimality of the model has converged to a particular value. Unfortunately, due to time constraints no such technique has been incorporated into final project. So far, it was assumed that the synaptic weights of the neural network had been properly adjusted in order to demonstrate the functionality of the forward propagation process. As it was previously mentioned, the adjustment of the synaptic weights of the neural network is achieved by a supervised training process known as back-propagation algorithm.

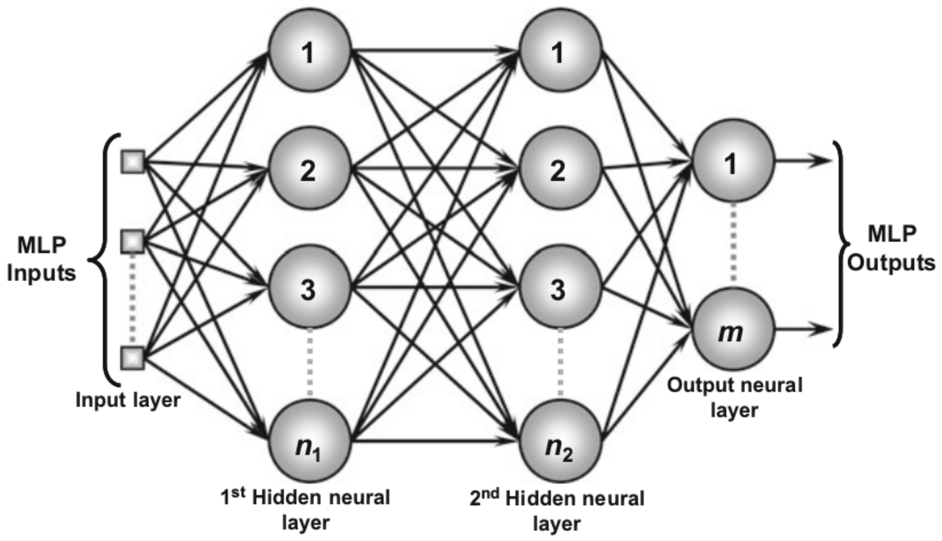


Illustration 2: Forward propagate process.
Taken from the book *Artificial Neural Networks: A Practical Course*

2.2 Training process of the ANN

There are two specific stages that take place during the back-propagation algorithm. The first one, the forward propagation procedure was described in the previous section. The second stage, back-propagation, adjusts the synaptic weights of the output layer by comparing the deviation of the generated output signals and the desired output signals. The error is then propagated backwards to the previous layer which adjusts it's synaptic weights based on the values of the synaptic weights of it's posterior layer and so on until we reach the initial input layer of the neural network [4]. This repetitive forward and backward propagation process is the very essence of the back-propagation algorithm, and it is repeated until the mean square error reaches a global minimum or a user predefined convergence value. A single forward-backward propagation iteration is called a training epoch and it is used to measure the duration of the training process. Illustration 3, provides a graphical visualisation of one training epoch where bold arrows represent the forward propagation stage and the dotted arrows represent the backward propagation stage.

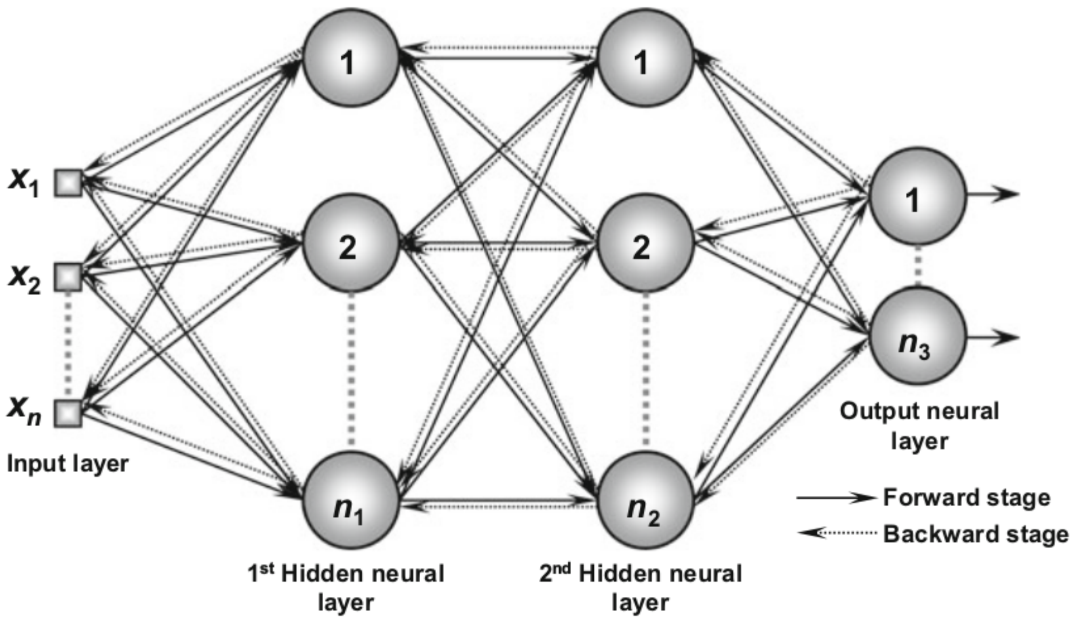


Illustration 3: The two stages of training.
Taken from the book *Artificial Neural Networks: A Practical Course*

2.3 Back-propagation algorithm

It is of prime importance to establish the mathematical representation of the two stages of the training process before the back-propagation algorithm is defined. For this purpose, it is important to break down the abstract concept of a neural network into smaller abstract concepts and those in turn into mathematical objects. Based on Illustration 5 and Illustration 4, the following notations are defined:

- $W_{ji}^{(L)}$, represents the synaptic weights matrix, where each cell connects the jth neuron of the current layer L to the ith neuron of the layer (L-1).
- $I_{j}^{(L)}$, represents the linear aggregator matrix, where each cell represents the weighted inputs related to the jth neuron of current layer L. It is defined as follows:
    - $I_j^{(L)} = \sum_{i=0}^{n} W_{ji}^{(L)} \cdot x_i$, when we are at the input layer L=1.
    - $I_j^{(L)} = \sum_{i=0}^{n} W_{ji}^{(L)} \cdot Y_i^{(L-1)}$, when we are at hidden layers or output layer.
- $Y_j^{(L)}$, represents the activation output matrix, where each cell stores the output signal for produced by the jth neuron of the current layer L. It is defined as follows:
    - $Y_j^{(L)} = g(I_j^{(L)})$, where g is the activation function.
- $\delta_j^{(L)}$, represents the gradient matrix, where each cell represents the local gradient associated with the jth neuron of the current layer L. It is defined as follows:
    - $\delta_j^{(L)} = (d_j - Y_j^{(L)}) \cdot g'(I_j^{(L)})$, when we are at the output layer
    - $\delta_j^{(L)} = -(\sum_{k=1}^{n} \delta_k^{(L+1)} \cdot W_{kj}^{(L+1)}) \cdot g'(I_j^{(L)})$, when we are at hidden layers or input layer.
    - η , represents the learning rate of the backpropagation process.

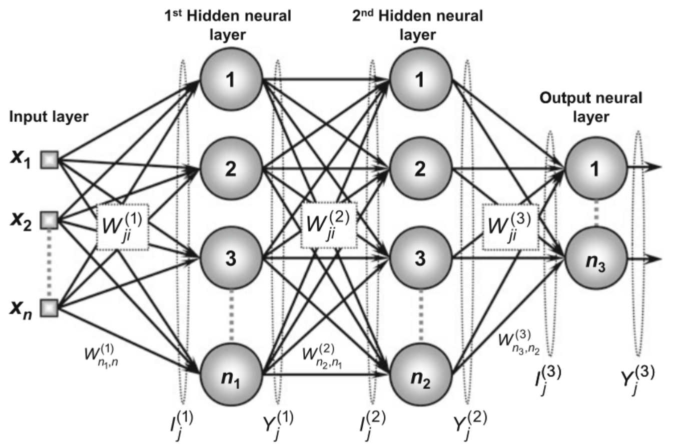


Illustration 5: Abstract components of a neural network.Taken from the book Artificial Neural Networks: A Practical Course

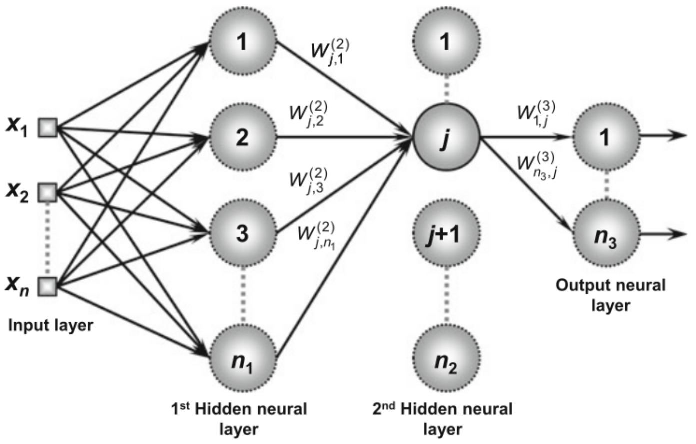


Illustration 4: Adjustment of hidden neural layers based on posterior layers.Taken from the book Artificial Neural Networks: A Practical Course

For the complete mathematical derivation of the back-propagation algorithm consult [2] and [5].
The adjustment of the synaptic weights matrices is given in algorithmic notation as follows:

1) $W_{ji}^{(L)}(t+1) = W_{ji}^{(L)}(t) + \eta \cdot \delta_j^{(L)} \cdot Y_i^{(L-1)}$, for all hidden and output layers.

2) $W_{ji}^{(L)}(t+1) = W_{ji}^{(L)}(t) + \eta \cdot \delta_j^{(L)} \cdot x_i$, for the input layer.

The general accuracy of neural network is obtained through the mean square error function. The absolute error could be used as well, but because of the nice mathematical properties that the square error has, namely it is differentiable in it's entire domain, it is more convenient to use the square than the absolute. The mean square error is given as follows:

square error: $E(k)=\frac{1}{2} \cdot \sum_{j=1}^{n} (d_j(k) - Y_j^{(L_{output})}(k))^2$, For the kth training sample.

mean square error: $E_M=\frac{1}{p} \cdot \sum_{k=1}^{p} E(k)$, Where p is the total number of training samples.

The quintessential of the back-propagation algorithm is to minimise the mean square error function by adjusting the synaptic weights of the neural network accordingly. After a certain number of iterations the algorithm should reach or get close enough to a global minimum at which point the training process is completed and the synaptic weights have encoded a generaliser for the given data-set. Having provided all the necessary mathematical representations for the abstract components of both stages of the training process it is time to provide a proper definition of the back-propagation algorithm in pseudo-code notation:

procedure train:

```
1   Initialise the synaptic weights of the neural network
    randomly between values zero and one.
2   Initialise the learning rate and the convergence constant.
3   Set the initial number of epochs to zero.
4   Repeat:
     4.1  calculate the mean square error
     4.2  for every sample in the training set:
           4.2.1  apply the forward propagate procedure.
           4.2.2  Apply the backward propagate procedure.
     4.3  Calculate the new mean square error.
     4.4  Increment the number of epochs by one.
5   Until the absolute different of the current mse and the
    previous mse is smaller than the convergence constant.
```

procedure forward propagate:

```
1   For all layers in the neural network starting from input layer toward the output layer:
     1.1  Obtain $I_j^{(L)}$ and $Y_j^{(L)}$ as described.
```

procedure backward propagate:

```
1   For all layers in the neural network starting from output layer towards the input layer:
     1.1  Obtain $\delta_j^{(L)}$ and adjust $W_{ij}^{(L)}$
```

At the end of the training process, the neural network is considered trained and the synaptic weights matrices have extracted all the necessary patterns from the given training data-set. At this point the model can be directly used on unseen data or stored into memory for future deployment.

2.4 Momentum parameter optimisation

Although many optimisation techniques have been proposed, the momentum parameter [6] is by far the simplest and the one specifically selected for this project. The algorithmic notation is given as follows:

$$W_{ji}^{(L)}(t+1) = W_{ji}^{(L)}(t) + \alpha \cdot (W_{ji}^{(L)}(t) - W_{ji}^{(L)}(t-1)) + \eta \cdot \delta_{j}^{(L)} \cdot Y_{i}^{(L-1)}$$

where α is the momentum rate that ranges between values [0,1]. The proper selection of a momentum constant and the initialisation of the synaptic weights matrices also plays an important role in the optimisation process [7]. When the variation of the mean square error between two training epochs is notable, the optimisation part of the above equation conducts a bigger incremental step towards the global minimum. On the other hand, when the variation of the mean square error between epochs is negligible, the optimisation part approaches the value of zero, making it's contribution insignificant. This implies that when the gradient descent is far from the global minimum, the optimisation part is more active, while when the gradient descent is closing in on a global minimum, the optimisation part approaches the value of zero transforming the above equation to the original back-propagation equation. Illustration 6, provides a geometric interpretation of the momentum parameter optimisation technique.

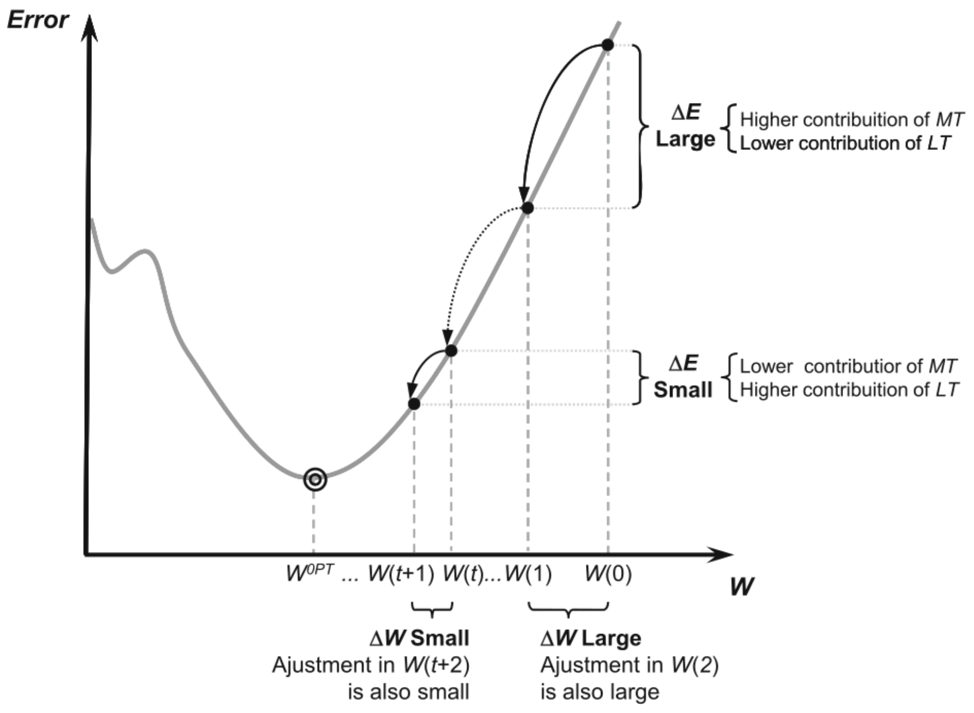

Error
ΔE Large
Higher contribuition of MT
Lower contribution of LT
ΔE Small
Lower contributior of MT
Higher contribuition of LT
W^OPT ... W(t+1) W(t)...W(1) W(0)
W
ΔW Small
Ajustment in W(t+2) is also small
ΔW Large
Ajustment in W(2) is also large

Illustration 6: Training process using momentum parameter optimisation. Taken from the book Artificial Neural Networks: A Practical Course

3.0 ANN library

In the previous sections, we provided an introduction to the concept of artificial neural networks and provided the necessary mathematical representations of the abstract components of a typical feed-forward neural network. Having properly understood the fundamental properties and operating principles of ANN, it is time to provide a thorough and detailed walk-through of the ANN library. Before dwelling into the architecture and the data structures, a proper introduction to the tools and scientific libraries used, will be initially provided.

3.1 The C programming language

The ANN library is written in the C programming language [8]. Despite being old, C still remains the most widely used programming language of all time, mostly because of it’s minimalist design that provides constructs that map efficiently to machine instructions. Pointers [9] and manual memory management are the bread and butter of the language as they provide direct access to hardware and the ability to manage memory productively and efficiently. These properties make the language the prime candidate for the implementation of a neural network library.

3.2 The GNU scientific library

The GNU scientific library [10] is a collection of modules and functions for numerical and scientific computing, written in C from scratch. The library provides an application interface for many modules and purposes. The ANN library makes extensive use of the vector and matrix libraries. Both matrix and vector libraries provide a view operation, which is basically a pointer to a subsection of a particular matrix or vector data structure. Views are temporary objects allocated on the stack and are used to operate on a subset of elements without the need to copy them into a new block cells. This ability of altering the state will play an important role during the performance analysis of the library.

3.3 Algorithms and data structures

The ANN library consists of five modules, neural layer, neural network, data-set and neural utilities. Each module has it’s header file defined in the include directory and it’s corresponding source file in the src directory. The neural layer module is the physical representation of the abstract concept of a neural layer in a neural network. It is defined as neural_layer_t and has as fields five gsl_matrix data structures. The components W, I and Y where defined in section 2.3, while D represents to the local gradient matrix (also defined in section 2.3) and O represents the synaptic weights matrix from the previous training epoch (used primarily for the optimisation process). The module contains a constructor and destructor as well as a set of getters for retrieving the memory addresses of it’s components. The neural net module contains the physical implementation of the neural network and a neural configuration data structures. The neural_config_t data structure is a simple struct that stores information and configuration options regarding the creation and training process for the neural network. The neural_net data structure is the physical implementation of the abstract concept of a neural network. The neural_net_t has as fields an array of neural_layer_t data structures and a neural_config_t which is given during the instantiation process. The module also defines,besides a constructor and destructor, a set of operations to train and predict and also dump and load the adjusted synaptic weights as binary into and from the file system. The neural_net_t data type depends explicitly on the neural_layer_t and neural_config_t types. These three data types are given as UML diagrams in Illustration 8, Illustration 7 and Illustration 9. These data structures are independent of the training process and have been built in such a way as to allow flexibility and the application of different training algorithms and configurations. This is achieved via the usage of function pointer data types that specify an interface for the activation function and it’s derivative and the training function.

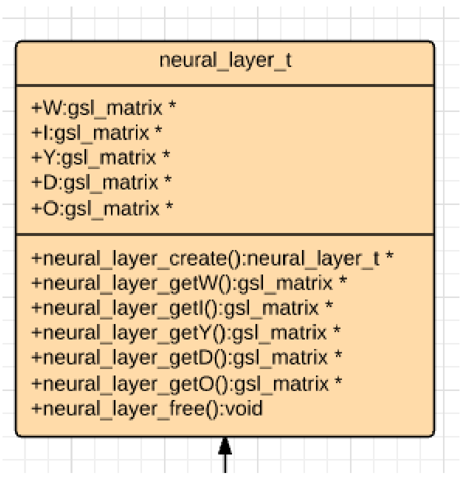

neural_layer_t
+W:gsl_matrix *
+I:gsl_matrix *
+Y:gsl_matrix *
+D:gsl_matrix *
+O:gsl_matrix *
+neural_layer_create():neural_layer_t *
+neural_layer_getW():gsl_matrix *
+neural_layer_getI():gsl_matrix *
+neural_layer_getY():gsl_matrix *
+neural_layer_getD():gsl_matrix *
+neural_layer_getO():gsl_matrix *
+neural_layer_free():void

Illustration 8: neural layer UML diagram

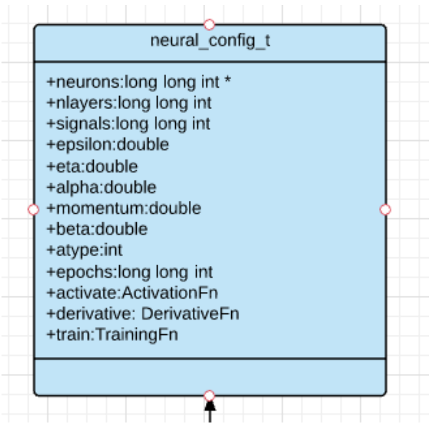


Illustration 7: neural configuration UML diagram

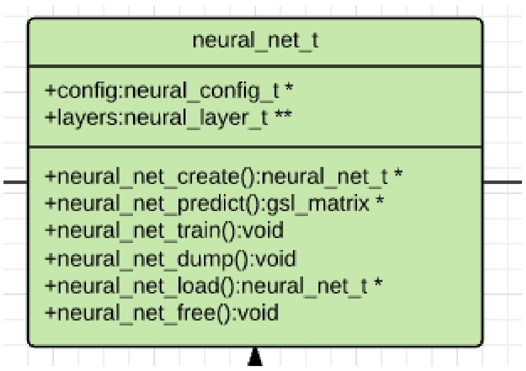

neural_net_t
+config:neural_config_t *
+layers:neural_layer_t **
+neural_net_create():neural_net_t *
+neural_net_predict():gsl_matrix *
+neural_net_train():void
+neural_net_dump():void
+neural_net_load():neural_net_t *
+neural_net_free():void

Illustration 9: neural network UML diagram

The neural utilities module contains the user-defined configuration options for the neural network data structure. It contains the definition of three activation functions and their corresponding derivatives, namely the logistic function, the hyperbolic tangent function and the linear function. It also contains the definition of the mean square error function and the back-propagation function that employs the momentum parameter optimisation technique. The data-set module contains the definition of a dataset_t data-type that represents the concept of a numeric data-set with rows and columns. It has as fields two gsl_vector data types that store the minimum and maximum values for each column in the data-set and additional information regarding the type of data being read and it's dimensions. Besides the definition of a constructor and a destructor, it defines functions that scale and re-scale the given data-set and dump and load the min max vectors in and from the file system.

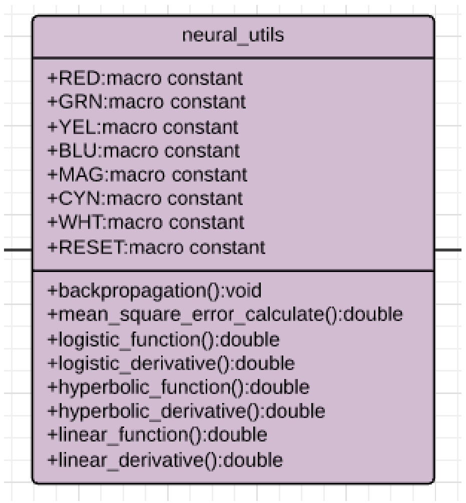

neural_utils
+RED:macro constant
+GRN:macro constant
+YEL:macro constant
+BLU:macro constant
+MAG:macro constant
+CYN:macro constant
+WHT:macro constant
+RESET:macro constant
+backpropagation():void
+mean_square_error_calculate():double
+logistic_function():double
+logistic_derivative():double
+hyperbolic_function():double
+hyperbolic_derivative():double
+linear_function():double
+linear_derivative():double

Illustration 11: neural utilities UML diagram

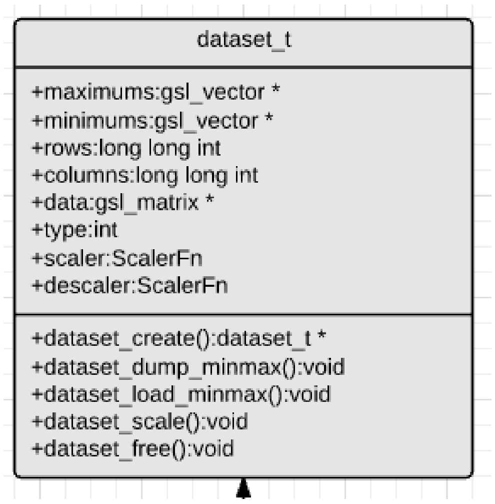


Illustration 10: dataset UML diagram

Finally, the last component that glues together all of the above modules is the main program defined in the main source file. It contains procedures that read command line arguments from the user and perform the corresponding actions, whether it is training a new model or deploying an already trained one for prediction or classification purposes. For detailed explanation look at the source code, as it has been thoroughly documented. Compiling all of the source files with the main file generates a command line program that can be used to train a neural network, dump the adjusted synaptic weights and the neural net configurations into a binary folder and load from the same directory to perform pattern classification or curve-fitting operations.

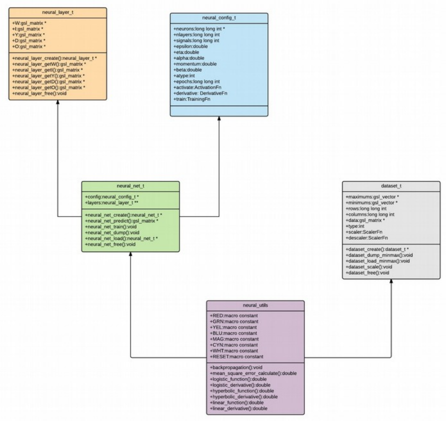


Illustration 12: UML diagrams of the architecture of the ANN library

3.4 Command line interface

Once the compilation process has been completed, a executable file is generated named ‘neuralnet’. This executable is the command line program. This program receives a set of configuration flags and based on them performs the corresponding action(s). The first flag ‘--train’ or ‘--predict’ refers to the execution mode. When the training flag is set the neural network creates a new model using the back-propagation algorithm with momentum parameter optimisation. When the prediction flag is set the neural network loads an already trained model from the file-system for deployment purposes. The second flag, ‘--curve-fitting’ or ‘pattern-classification’ refers to the execution type. When the curve fitting type is set, the neural network will train or load a model for curve-fitting purposes. On the other hand, when the pattern-classification type is set the neural network will train or load a model for pattern classification purposes. The third flag, ‘--normalization=<yes|no>’ takes a yes or no value depending on whether the user wants to normalise the data-set or not. At this point, the configuration flags begin to differ for the different execution modes.

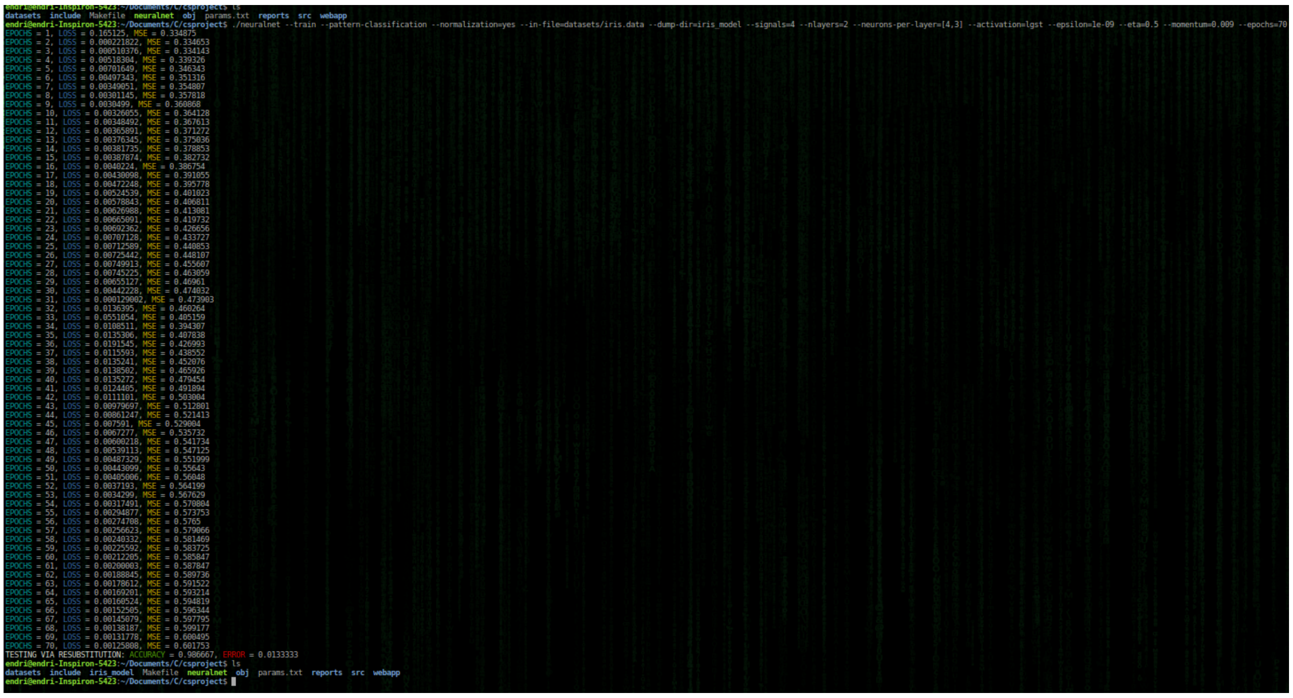
For the **training mode**, the user has to specify the path to the data-set file using the ‘--in-file=<filepath>’ flag. The user has to also specify the name of the directory where the adjusted synaptic weights will be stored using the ‘--dump-dir=<filepath>’ flag. Having specified the corresponding paths to the training data-set and the directory in which the model will be stored, it is time to configure the neural network. The user has to specify the number of input signals or features that the training data-set contains using the flag ‘--signals=<number>’. The user must specify the desired number of layers the neural network should have using the flag ‘--nlayers=<number>’. Once the number of neural layers has been defined, the number of neurons per layer has to also be defined by the user using the flag ‘--neurons-per-layer=[number, ..]>’. The neurons at each layer must have an activation function that reacts to stimuli, that function is set using the flag ‘activation=<lnr|lgst|htan>’. These are the compulsory configuration options that the user must provide for the training process. The following flags are optional and the user may choose not to provide. The user can specify the convergence rate using the flag ‘--epsilon=<number>’, the learning rate using the flag ‘--eta=<number>’, the momentum rate using the flag ‘--momentum=<number>’ and the number of epochs using the flag ‘--epochs=<number>’. The user can also specify the coefficient values of the activation function using ‘--alpha=<number>’ and ‘--beta=<number>’. To get a better understanding of how to properly use the command line tool, consider the following example for building an iris flower classifier using the iris flower data-set.

The iris flower data-set has 150 number of instances, 4 features (input signals) and 3 classes. To build a classifier we have to properly encode the classes as follows:

1.	Iris Setosa	= 1 0 0
2.	Iris Versicolour	= 0 1 0
3.	Iris Virginica	= 0 0 1

From this observation, we can conclude that the output layer of the neural network will have three neurons. The iris data-set with the encoded classes can be found at the datasets directory under the name iris.data. The first and second lines of the file must always contain the number of rows and number of columns of the data-set. Having properly formatted the data-set it is time to build a classifier for the iris data-set. We are going to call the directory where we save the model 'iris_model'. Here is the terminal command for the training process:

```
./neuralnet --train –pattern-classification –normalization=yes –in-file=datasets/iris.data –dump-dir=iris_model –signals=4 –nlayers=2 –neurons-per-layer=[4,3] –activation=lgst –epsilon=1e-09 –eta=0.5 –momentum=0.009 –epochs=70
```

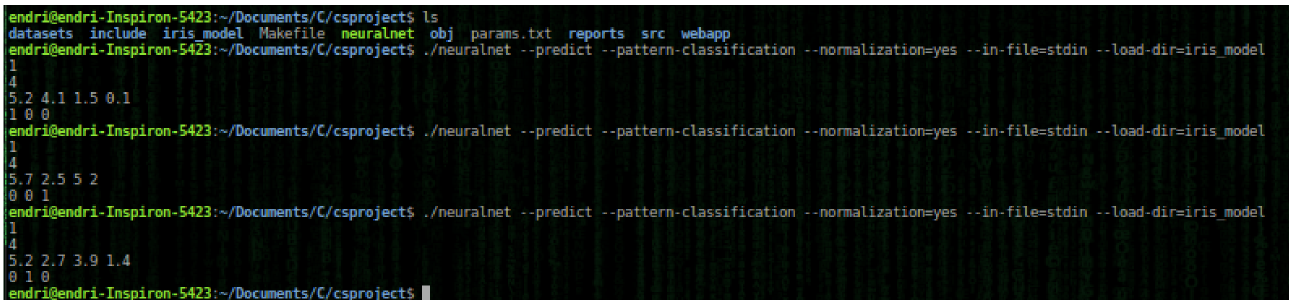


```
endri@endri-Inspiron-5423:~/Documents/C/csproject$ ls
datasets  include  Makefile  neuralnet  obj  params.txt  reports  src  webapp
endri@endri-Inspiron-5423:~/Documents/C/csproject$ ./neuralnet --train --pattern-classification --normalization=yes --in-file=datasets/iris.data --dump-dir=iris_model --signals=4 --nlayers=2 --neurons-per-layer=[4,3] --activation=lgst --epsilon=1e-09 --eta=0.5 --momentum=0.009 --epochs=70
EPOCHS = 1, LOSS = 0.165125, MSE = 0.334875
EPOCHS = 2, LOSS = 0.000221822, MSE = 0.334653
EPOCHS = 3, LOSS = 0.000510376, MSE = 0.334143
EPOCHS = 4, LOSS = 0.00518304, MSE = 0.339326
EPOCHS = 5, LOSS = 0.00701649, MSE = 0.346343
EPOCHS = 6, LOSS = 0.00497343, MSE = 0.351316
EPOCHS = 7, LOSS = 0.00349051, MSE = 0.354807
EPOCHS = 8, LOSS = 0.00301145, MSE = 0.357818
EPOCHS = 9, LOSS = 0.0030499, MSE = 0.360868
EPOCHS = 10, LOSS = 0.00326055, MSE = 0.364128
EPOCHS = 11, LOSS = 0.00348492, MSE = 0.367613
EPOCHS = 12, LOSS = 0.00365891, MSE = 0.371272
EPOCHS = 13, LOSS = 0.00376345, MSE = 0.375036
EPOCHS = 14, LOSS = 0.00381735, MSE = 0.378853
EPOCHS = 15, LOSS = 0.00387874, MSE = 0.382732
EPOCHS = 16, LOSS = 0.0040224, MSE = 0.386754
EPOCHS = 17, LOSS = 0.00430098, MSE = 0.391055
EPOCHS = 18, LOSS = 0.00472248, MSE = 0.395778
EPOCHS = 19, LOSS = 0.00524539, MSE = 0.401023
EPOCHS = 20, LOSS = 0.00578843, MSE = 0.406811
EPOCHS = 21, LOSS = 0.00626988, MSE = 0.413081
EPOCHS = 22, LOSS = 0.00665091, MSE = 0.419732
EPOCHS = 23, LOSS = 0.00692362, MSE = 0.426656
EPOCHS = 24, LOSS = 0.00707128, MSE = 0.433727
EPOCHS = 25, LOSS = 0.00712589, MSE = 0.440853
EPOCHS = 26, LOSS = 0.00725442, MSE = 0.448107
EPOCHS = 27, LOSS = 0.00749913, MSE = 0.455607
EPOCHS = 28, LOSS = 0.00745225, MSE = 0.463059
EPOCHS = 29, LOSS = 0.00655127, MSE = 0.46961
EPOCHS = 30, LOSS = 0.00442228, MSE = 0.474032
EPOCHS = 31, LOSS = 0.000129002, MSE = 0.473903
EPOCHS = 32, LOSS = 0.0136395, MSE = 0.460264
EPOCHS = 33, LOSS = 0.0551054, MSE = 0.405159
EPOCHS = 34, LOSS = 0.0108511, MSE = 0.394307
EPOCHS = 35, LOSS = 0.0135306, MSE = 0.407838
EPOCHS = 36, LOSS = 0.0191545, MSE = 0.426993
EPOCHS = 37, LOSS = 0.0115593, MSE = 0.438552
EPOCHS = 38, LOSS = 0.0135241, MSE = 0.452076
EPOCHS = 39, LOSS = 0.0138502, MSE = 0.465926
EPOCHS = 40, LOSS = 0.0135272, MSE = 0.479454
EPOCHS = 41, LOSS = 0.0124405, MSE = 0.491894
EPOCHS = 42, LOSS = 0.0111101, MSE = 0.503004
EPOCHS = 43, LOSS = 0.00979697, MSE = 0.512801
EPOCHS = 44, LOSS = 0.00861247, MSE = 0.521413
EPOCHS = 45, LOSS = 0.007591, MSE = 0.529004
EPOCHS = 46, LOSS = 0.0067277, MSE = 0.535732
EPOCHS = 47, LOSS = 0.00600218, MSE = 0.541734
EPOCHS = 48, LOSS = 0.00539113, MSE = 0.547125
EPOCHS = 49, LOSS = 0.00487329, MSE = 0.551999
EPOCHS = 50, LOSS = 0.00443099, MSE = 0.55643
EPOCHS = 51, LOSS = 0.00405006, MSE = 0.56048
EPOCHS = 52, LOSS = 0.0037193, MSE = 0.564199
EPOCHS = 53, LOSS = 0.0034299, MSE = 0.567629
EPOCHS = 54, LOSS = 0.00317491, MSE = 0.570804
EPOCHS = 55, LOSS = 0.00294877, MSE = 0.573753
EPOCHS = 56, LOSS = 0.00274708, MSE = 0.5765
EPOCHS = 57, LOSS = 0.00256623, MSE = 0.579066
EPOCHS = 58, LOSS = 0.00240332, MSE = 0.581469
EPOCHS = 59, LOSS = 0.00225592, MSE = 0.583725
EPOCHS = 60, LOSS = 0.00212205, MSE = 0.585847
EPOCHS = 61, LOSS = 0.00200003, MSE = 0.587847
EPOCHS = 62, LOSS = 0.00188845, MSE = 0.589736
EPOCHS = 63, LOSS = 0.00178612, MSE = 0.591522
EPOCHS = 64, LOSS = 0.00169201, MSE = 0.593214
EPOCHS = 65, LOSS = 0.00160524, MSE = 0.594819
EPOCHS = 66, LOSS = 0.00152505, MSE = 0.596344
EPOCHS = 67, LOSS = 0.00145079, MSE = 0.597795
EPOCHS = 68, LOSS = 0.00138187, MSE = 0.599177
EPOCHS = 69, LOSS = 0.00131778, MSE = 0.600495
EPOCHS = 70, LOSS = 0.00125808, MSE = 0.601753
TESTING VIA RESUBSTITUTION: ACCURACY = 0.986667, ERROR = 0.0133333
endri@endri-Inspiron-5423:~/Documents/C/csproject$ ls
datasets  include  iris_model  Makefile  neuralnet  obj  params.txt  reports  src  webapp
endri@endri-Inspiron-5423:~/Documents/C/csproject$ 
```

Illustration 13: training a neural network using the iris flower data-set. It can be seen that after the training process a new directory has been created called 'iris_model' that contains the obtained iris flower classifier.

For the **prediction mode**, the user has to provide the path to the unseen data-set and the path to the saved model using the flags '--in-file=<filepath>' and –load-dir=<filepath>. The user has to remember whether he normalised the training data or not, and act accordingly to the given unseen data. For the demonstration purposes, we are going to use as input file the standard input stream. It is important to remember that the first line and second line must contain the number of rows and the number of columns. To deploy the trained iris flower classifier we have to execute the following terminal command:

```
./neural –predict –pattern-classification –normalization=yes –in-file=stdin –load-dir=iris_model
(input)$ 1
(input)$ 4
(input)$ 5.2 4.1 1.5 0.1
```



```
endri@endri-Inspiron-5423:~/Documents/C/csproject$ ls
datasets  include  iris_model  Makefile  neuralnet  obj  params.txt  reports  src  webapp
endri@endri-Inspiron-5423:~/Documents/C/csproject$ ./neuralnet --predict --pattern-classification --normalization=yes --in-file=stdin --load-dir=iris_model
1
4
5.2 4.1 1.5 0.1
1 0 0
endri@endri-Inspiron-5423:~/Documents/C/csproject$ ./neuralnet --predict --pattern-classification --normalization=yes --in-file=stdin --load-dir=iris_model
1
4
5.7 2.5 5 2
0 0 1
endri@endri-Inspiron-5423:~/Documents/C/csproject$ ./neuralnet --predict --pattern-classification --normalization=yes --in-file=stdin --load-dir=iris_model
1
4
5.2 2.7 3.9 1.4
0 1 0
endri@endri-Inspiron-5423:~/Documents/C/csproject$ 
```

Illustration 14: deployment of trained iris classifier. It can be seen how the neural network classifies different input signals based on the patterns it has extracted from the training set.

The program can equally be used for curve-fitting applications with a few minor modifications to the training configuration options. We are going to demonstrate the curve-fitting capabilities in the next section where we elaborate on the theoretical correctness of the neural network library and provide empirical evidence to support it.

4.0 Correctness

Due to the nature of ANN, it is really difficult to prove the correctness of the back-propagation algorithm. As we have repetitively mentioned in the previous sections, a generaliser is encoded within the synaptic weights of the neural network. This encoding prevents as from gaining any insights into the inner workings of the neural network. For this specific reason ANN are also refereed to as black-box models [11]. The correctness of the ANN library can be proven implicitly by examining the loss between consecutive training epochs and the mean square error value. Also empirical evidence will be provided demonstrating the universal curve-fitting approximation theorem [12].

4.1 Convergence of mean square error

A proper way to validate the functionality of the ANN library is to observe the difference between the synaptic weights of consecutive training epochs. It can be observed that indeed, the loss decreases with each training epoch and that the mean square error value convergences to a global or local minimum. This implicitly implies that the synaptic weights are constantly being adjusted with each epoch such that the mean square error value is minimised. From this aspect, the back-propagation algorithm seems to work correctly.

4.2 Universal function approximators

A more formal way of proving the correctness of the ANN library is to gather empirical evidence regarding curve-fitting applications and demonstrate that it holds the properties of a universal function approximator [3]. Illustration 16, Illustration 17 and Illustration 15 depict the approximation of the semicircle function, the quadratic function and the sin function using artificial neural networks. Based on the experimental evidence it can be conclude that the ANN library is correct and holds the above mentioned properties. To repeat the curve fitting experiments, the user has to navigate into the datasets directory and execute the following octave scripts circlef.m, quadratic.m and sinf.m. For further documentation consult the source code.

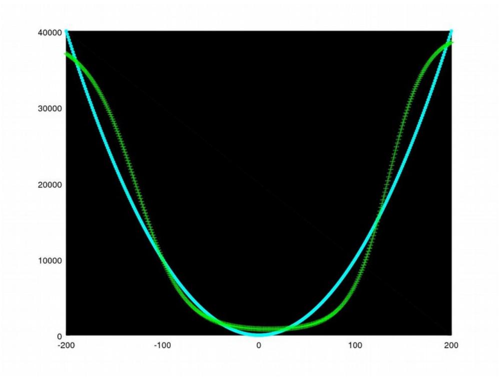


Illustration 15: blue - quadratic function, green -quadratic approximator

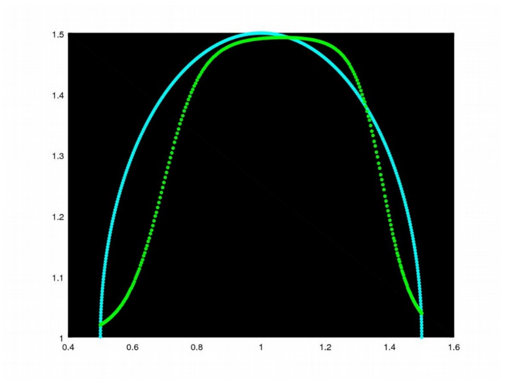


Illustration 16: blue - semicircle function, green - semicircle approximator

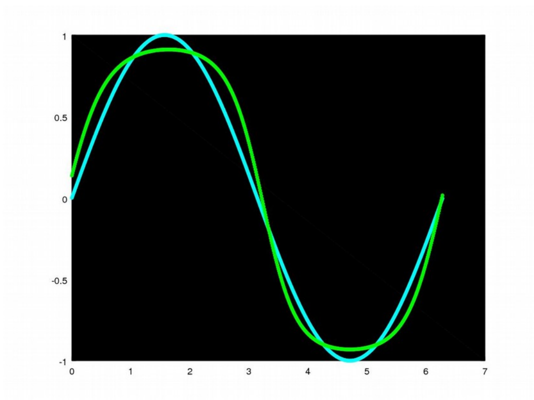


Illustration 17: blue - sin function, green - sin approximator

4.3 Pattern classification

For multi-class pattern classification applications the output targets must be properly encoded. Given n-classes the neural network will contain in it’s output layer n neurons. The classes are encoded as binary strings of length n. Consider the classes A,B and C of some data-set, the equivalent encoded classes are 100,010 and 001. Each character in the string is represented as a separate column in the data-set. Therefore a data-set with n classes is going to have (n-1) additional columns. The correctness of the pattern-classification is implied by sections 4.1 and 4.2.

4.4 Testing via resubstitution

Due to time restraints, the only performance evaluation method that was implemented was the testing via resubstitution. Once the model has been trained, it is given as testing input the same data-set. This is not a proper evaluation technique but it suffices for the scope of the subject.

5.0 Computational properties

One of the main requirements for the project was to develop an extremely minimalist and efficient ANN library. Selecting the proper tools played an important role in the performance evaluation and resource management capabilities of the application. Though no comparison benchmarks were created in relation to other similar libraries, the performance capabilities are beyond doubt satisfactory.

5.1 Time complexity

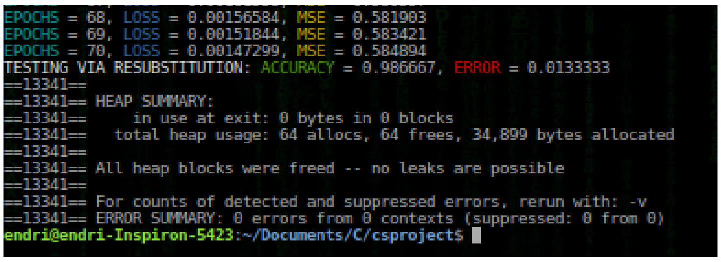
It is really hard to argue about the time complexity of the back-propagation algorithm. For different data-sets with different features the algorithm might take longer to converge to a global minimum. This uncertainty is what makes the time complexity properties of the algorithm difficult to reason about. More specifically, the number of training epochs varies and depends upon the training data-set and the configuration options. Nevertheless, the nested loops per training epoch never exceed the upper bound time complexity of cubic power O(n^3). Also given the fact that, the matrices used for the training process are allocated only once and constantly overwritten with the passage of training epochs thus boosting the performance of the algorithm significantly. For further detailed, consult the source code.

5.2 Space complexity

As previously mentioned, the number of matrices and manual memory allocations remains constant throughout the entire duration of the training process. This is achieved using the view capabilities provided by the gnu scientific library for matrix operations. The size of the memory allocated depends entirely on the number of rows and columns in the data-set yielding an upper bound space complexity of quadratic power O(n^2). Given the highly efficient space complexity, the library can be deployed in embedded system applications that require some form of machine learning capabilities.

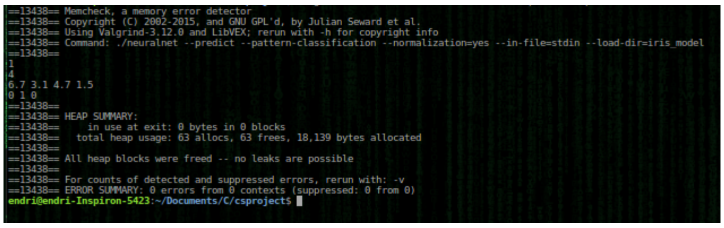
5.3 Memory management

It is of prime importance that the application manages memory allocated on heap correctly and efficiently. All data structures in the modules have their corresponding constructors and destructors properly defined. By using the valgrind memory checker tool [13] we can observer that all memory is freed and that there are no corruptions or buffer overflows during the execution of the training and deployment stages.



```
EPOCHS = 68, LOSS = 0.00156584, MSE = 0.581903
EPOCHS = 69, LOSS = 0.00151844, MSE = 0.583421
EPOCHS = 70, LOSS = 0.00147299, MSE = 0.584894
TESTING VIA RESUBSTITUTION: ACCURACY = 0.986667, ERROR = 0.0133333
==13341==
==13341== HEAP SUMMARY:
==13341==     in use at exit: 0 bytes in 0 blocks
==13341==   total heap usage: 64 allocs, 64 frees, 34,899 bytes allocated
==13341==
==13341== All heap blocks were freed -- no leaks are possible
==13341==
==13341== For counts of detected and suppressed errors, rerun with: -v
==13341== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
endri@endri-Inspiron-5423:~/Documents/C/csproject$
```

Illustration 18: training stage, using valgrind memchecker



```
==13438== Memcheck, a memory error detector
==13438== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==13438== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==13438== Command: ./neuralnet --predict --pattern-classification --normalization=yes --in-file=stdin --load-dir=iris_model
==13438==
1
4
6.7 3.1 4.7 1.5
0 1 0
==13438==
==13438== HEAP SUMMARY:
==13438==     in use at exit: 0 bytes in 0 blocks
==13438==   total heap usage: 63 allocs, 63 frees, 18,139 bytes allocated
==13438==
==13438== All heap blocks were freed -- no leaks are possible
==13438==
==13438== For counts of detected and suppressed errors, rerun with: -v
==13438== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
endri@endri-Inspiron-5423:~/Documents/C/csproject$
```

Illustration 19: deployment stage using valgrind memchecker

6.0 Thyroid disease data-set

The objective of the project is to build a mathematical model that can accurately classify thyroid diseases based on a given set of inputs. The inputs are laboratory tests that represent different hormonal levels of a patient. The selected data-set is rich enough in instances and features and provides detailed documentation regarding its origin and purpose.

6.1 Origin and purpose

The thyroid disease data-set was originally established by Danny Coomans, of the department of mathematics and statistics at James Cook University and was donated in 1992 to the University of California’s machine learning data-set repository. The data-set has been used for comparison of multivariate discriminant techniques [14] and in medical decision making applications [15]. Five laboratory tests are used to predict whether the patient’s thyroid functioning belongs to the euthyroidism, hypothyroidism and hyperthyroidism classes.

6.2 Context and features

There are 215 number of instances in the data-set and each instance contains 5 attributes. This translates to 215 rows of sample data, each 6 (including target output) columns wide. The features are continuous and numeric in nature and there are no specific rows or columns with missing attributes. The features are the following:

1. T3-resin update test. (As percentage)
2. Total serum thyroxin as measured by the isotopic displacement method.
3. Total serum triiodothyronine as measured by radioimmuno assay.
4. Basal thyroid-stimulating hormone (TSH) as measured by radioimmuno assay.
5. Maximal absolute difference of TSH value after injection of 200 micro grams of thyrotropin-releasing hormone as compared to the basal value.

There are three classes (output targets) in the data-set. The value 1 represents euthyroidism, value 2 represents hyperthyroidism and value 3 represents hyperthyroidism. The class distribution in terms of instances is given as follows:

1. class 1: (euthyroidism)	150
2. class 2: (hyperthyroidism)	35
3. class 3: (hypothyroidism)	30

6.3 Target encoding

The classes of the data-set have to be properly encoded using binary notation. The value 1, which represents euthyroidism is encoded as 1 0 0, the value of 2, which represents hyperthyroidism is encoded as 0 1 0 and the value of 3, which represents hypothyroidism is encoded as 0 0 1. Each character in the string represents a separate column in the data-set. Therefore the final formatted data-sets contains 8 columns in total, 5 for the features and 3 for the encoded output targets.

6.4 Training configuration

Through repetitive experimentation, the following configurations give an optimal thyroid disease classifier:

Neural layers:	2
Neurons per layer:	20, 3
Activation function:	Logistic function
Convergence rate:	1e-12
Learning rate:	0.5
Momentum rate:	0.009
Epochs:	700

The equivalent shell command is given as follows:

```
./neuralnet --train --pattern-classification --normalization=yes --in-file=datasets/thyroid-train.data --dump-dir=thyroidologist --signals=5 --nlayers=2 --neurons-per-layer=[20,3] --activation=lgst --epsilon=1e-12 --eta=0.5 --momentum=0.009 –epochs=700
```

6.5 Obtaining a classifier

The above command builds an optimal thyroid disease classifier based on the given data-set and stores the adjusted synaptic weights matrices into the thyroidologist directory. It is recommended to not modify any of the files in that directory. There are two ways to deploy the obtained classifier. The first one is the usage of the command line program which is achieved as follows:

```
$ ./neuralnet –predict –pattern-classification –normalization=yes –in-file=stdin –load-dir=thyroidologist
(input)$ 1
(input)$ 5
(input)$ f1 f2 f3 f4 f5
```

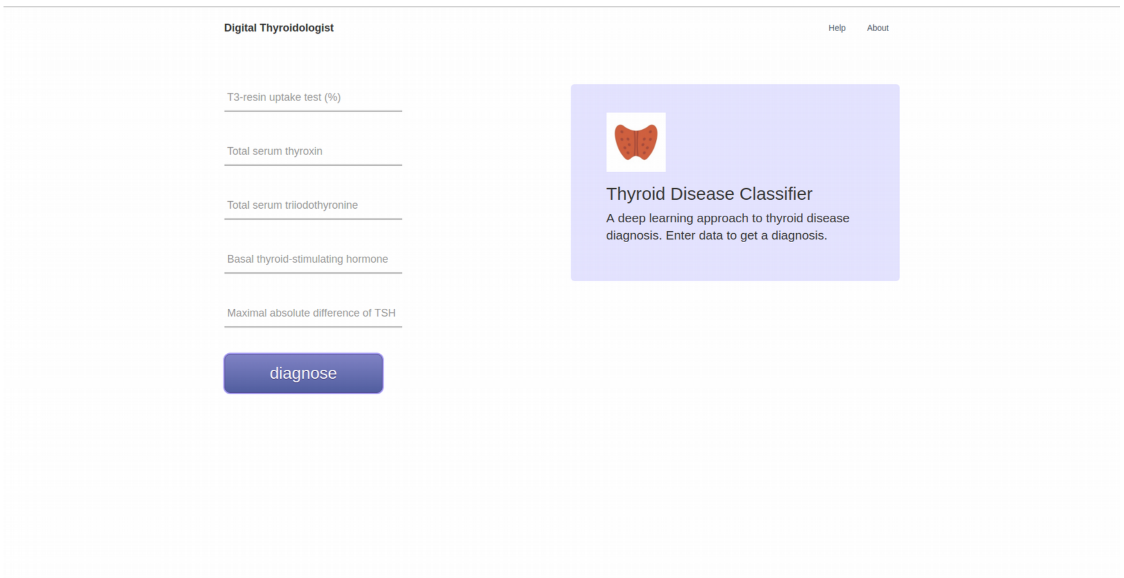
The f1, f2, f3, f4 and f5 are the corresponding input signals (features). The program will output the result in binary, that is 1 0 0, 0 1 0 or 0 0 1. of course this approach is not user-friendly to the medical expert. In the next section a web application that abstracts away technical and unnecessary details will be presented.

7.0 Web application

A minimalist web application has been developed that provides an interface for interacting with the trained thyroid disease classifier. The application was built using the nodejs environment and the socket.io library for real-time communication and uses processes to communicate with the neuralnet program.

7.1 Minimalist design

The application interface is very simple and minimal as to demonstrate the applicability of the neuralnet program and how it can interact with different environments and processes to provide real time feedback to medical experts. To serve the web application the user has to navigate into the web app directory and execute the command ‘node index.js’ which will open up a port at http://localhost:3000 that listens for http requests from users and servers the corresponding HTML documents. The application interface can be seen at Illustration 20.The user enters the corresponding values at the input fields and clicks the diagnose button. The client-side script validates the inputs and if they are correct and within the corresponding ranges it sends the data as json to the server via web sockets. The server receives the json data, instantiates a new process and fetches it into the neuralnet program which in turns classifies the data accordingly and prints the results. Once the results have been received by the server it sends them back to the client which in turns prints it to the user. Because of the real-time communication between server and client the performance of the web interface is lightning fast. Of course the provided interface is just a simple demonstration of how different components or environments can interact with the built ANN command line program.


Digital Thyroidologist
Help
About
T3-resin uptake test (%)
Total serum thyroxin
Total serum triiodothyronine
Basal thyroid-stimulating hormone
Maximal absolute difference of TSH
diagnose

Thyroid Disease Classifier

A deep learning approach to thyroid disease diagnosis. Enter data to get a diagnosis.

Illustration 20: Application interface for interacting with the trained thyroid disease classifier.

7.2 Final product

There are three project deliverables that have been presented in this report. The first one was a highly abstract and flexible artificial neural network library written in the C programming language using the GNU scientific library. The ANN library has been thorough designed in a way that it allows the implementation of different kinds of training algorithms or the design of other neural networks besides multi-layer perceptrons. The second deliverable, was the neuralnet command line program that can train models for both pattern classification and curve-fitting applications and deploy them for further usage later on. Finally, the third deliverable was a optimally trained, thyroid disease classifier with which a medical expert can interact via a web application interface.

8.0 Conclusion

To sum up, an introduction to the proposed project, the requirements and the constrains was initially presented. A thorough and detailed description of the artificial neural networks was outlined, including the operating principle, the mathematical properties and the optimisation techniques. A rigorous explanation of the design of the ANN library and it’s correctness was given and it’s computational properties were stated. A short introduction to the thyroid disease data-set and it’s properties was provided and a optimal configuration for the training of the thyroid disease classifier was proposed. Based on the final product and the properties demonstrated throughout the report it is safe to assume that indeed the neuralnet program and especially the web application interface can be used for the precise identification of diagnostics regarding thyroid diseases.

References

[1] J. E. Dayhoff and J. M. DeLeo, “Artificial neural networks,” *Cancer*, vol. 91, no. S8, pp. 1615–1635, Apr. 2001.

[2] *Artificial Neural Networks - A Practical Course | Ivan Nunes da Silva | Springer*. .

[3] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Netw.*, vol. 2, no. 5, pp. 359–366, Jan. 1989.

[4] J. Li, J. Cheng, J. Shi, and F. Huang, “Brief Introduction of Back Propagation (BP) Neural Network Algorithm and Its Improvement,” in *Advances in Computer Science and Information Engineering*, Springer, Berlin, Heidelberg, 2012, pp. 553–558.

[5] *Neural Networks and Statistical Learning | Ke-Lin Du | Springer*. .

[6] E. Barnard, “Optimization for training neural nets,” *IEEE Trans. Neural Netw.*, vol. 3, no. 2, pp. 232–240, Mar. 1992.

[7] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *PMLR*, 2013, pp. 1139–1147.

[8] B. W. Kernighan, *The C programming language*, 2nd ed. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[9] R. M. Reese, *Understanding and Using C Pointers*. .

[10] M. Galassi, *GNU scientific library: reference manual*, 3rd ed., For GSL version 1.12. Bristol: Network Theory, 2009.

[11] J. E. Dayhoff and J. M. Deleo, “Artificial neural networks: opening the black box,” *Cancer*, vol. 91, no. 8 Suppl, pp. 1615–35, 2001.

[12] B. Kosko, “Fuzzy systems as universal approximators,” *Comput. IEEE Trans. On*, vol. 43, no. 11, pp. 1329–1333, 1994.

[13] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.

[14] D. Coomans, I. Broeckaert, M. Jonckheer, and D. L. Massart, “Comparison of multivariate discrimination techniques for clinical data--application to the thyroid functional state.,” *Methods Arch.*, vol. 22, pp. 93–101, 1983.

[15] N. Bratchell, “Potential pattern recognition in chemical and medical decision making D. Coomans and I. Broeckaert, Research Studies Press Ltd., Letchworth, 1986. (Chemometrics Series). No. of pages: xiii + 256. Price: £29.50. ISBN: 0 86380 041 6,” *J. Chemom.*, vol. 1, no. 4, pp. 247–248, Oct. 1987.