INANNA++
A Highly Object-Oriented C++ Library for Artificial Neural Networks

Marko A. Grönroos
Computer Science
Department of Mathematical Sciences
University of Turku, 2000

Contents

Chapter 1

Introduction

> "Aiming for universality in an [initial] design is a prescription for a project that will never be completed."
> — Bjarne Stroustrup, The C++ Programming Language, 3rd edition

INANNA is a library for neural network computation and learning. It uses the MAGICLIB toolkit as its foundation class library.

Despite of the Stroustrup's warnings above, we aimed for generality and openness in the design of this library. Being able to reimplement any central parts of the tools is important in scientific research, where we want to test new ideas. In object-oriented programming, adding new parts is done by inheriting from abstract base classes. This is the approach which we use heavily in the libraries.

1.1 Sources of documentation

This documentation describes the major modules and classes of all the five libraries. We use an example-based approach in our presentation; more accurate descriptions of the classes can be found from the source files. Especially the header files are quite well documented. The source files are currently available from URL:

http://www.iki.fi/magi/ohjemointi/

This documentation provides just a brief introduction and some tutorial material to the five libraries. More documentation can be found from various simple test projects done with the library. See Appendices for more information.

1.2 Hands-on intro

We start by creating a simple neural network:

```
FreeNetwork myNet;
myNet.setBuilder (new LayeredTopology ("2-2-1"));
myNet.createNeurons ();
myNet.connectFull ();
myNet.init (0.5);
```

This network has two input units, a hidden layer with two units, and one output unit. Inanna uses the concept of *topology builders* to build and maintain network architectures. The *connectFull()* consults the builder to connect the layers. The *LayeredTopology* builder is the default for *FreeNetwork*, so we can do the above a bit easier with:

```
FreeNetwork myNet ("2-2-1");
```

In this case each layer is automatically connected totally to the successive layer. After the network has been created, it can be trained. We use here the RProp (resilient backpropagation) learning algorithm with parameters $\Delta_0 = 0.1$ (learning speed at the beginning), $\Delta_{max} = 50.0$ (maximum learning speed) and $\alpha = 4$ (weight decay exponent). In the following example we first set these parameters, then load the training data from a file, and finally train the network for a 1000 training cycles:

```
ANNTrainer* myTrainer = new TrainerLib(‘‘RProp’’);
myTrainer->addFeature (new WeightDecay (0.9999));
myTrainer->setParam (‘‘eta-’’, 0.1); /* Initial learning speed */
myTrainer->setParam (‘‘eta+’’, 50);  /* Maximum learning speed */
myNet.setTrainer (myTrainer);
PatternSet trainset ("mydata.pat");  /* Read in SNNS format (.pat) */
myNet.train (trainset, 1000);        /* Train for 1000 cycles */
```

The length of the parameter vector depends on the training algorithm.

After the neural network has been trained, it can be used to calculate output values from input vectors. In the following example we have the expected output values also in the testing pattern set, which we use to calculate mean squared error (MSE) in relation to the test set:

```
PatternSet testset ("testset.pat");

// Iterate through the test set
float sum=0; /* Calculate the sum of squared errors */
for (int p=0; p<testset.patterns; p++) {
    // Let the network compute outputs for the test set pattern p
    Vector outputs = myNet.test (testset, p);

    for (int i=0; i<outputs.size; i++) {
        sout.printf ("%f ", outputs[i]); /* Print an output value */

        // Add the squared error to the sum
        sum = sum + sqrt (outputs[i] - testset.output(p,i));
    }
}

// Calculate the mean of squared errors (MSE)
float MSE = sum / (testset.patterns * testset.outputs);
```

The *ANNetwork* class provides also a direct method for calculating the MSE in relation to a given pattern set:

```
float MSE = myNet.test (testset); /* Test and calculate MSE */
```

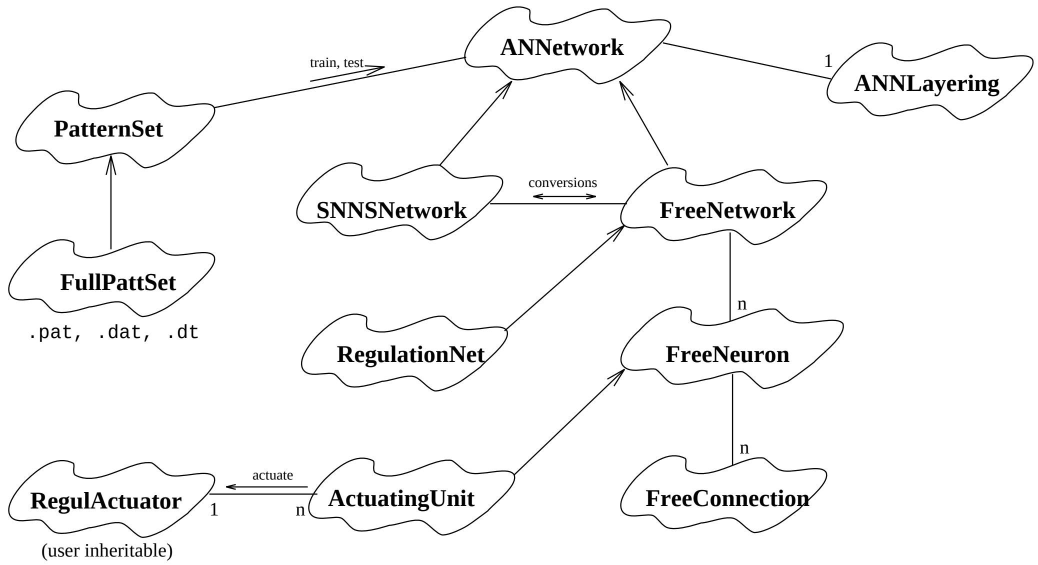

Figure 1.1: ANNlib top-level class hierarchy

1.3 Learning features

Currently there are no algorithms. The initial version will include:

- Vanilla Backpropagation
- Backpropagation with momentum
- Resilient Backpropagation

The algorithms of the library have been valitated by comparing the results with those reported by Prechelt [?] with his examples with the PROBEN1 datasets. Our results were within acceptable limits from the results reported by Prechelt. There were minor differences in the results that were probably caused by implementation-specific undocumented differences in the training algorithms.

1.4 Neural network classes

ANNetwork is the base class of all kinds of neural networks. It has interfaces for training and testing neural networks, which the inheritors should implement. The class hierarchy is illustrated in Figure 1.1.

Two major neural network classes are currently implemented in the library: *FreeNetwork* and *SNNSNetwork*. The former is “free” in the sense that it has very open object-oriented structure; most of its functionalities can easily be replaced with inheritance, etc. The latter of the classes is an interface to the SNNS. Both of the classes inherit the abstract *ANNetwork* baseclass.

1.4.1 Network topology builder

The *ANNLayering* class controls and provides information about the possible layering structure of an *ANNetwork*. Its main purpose is to handle network descriptions of format "`33-10-1`". Pure feedforward networks do not need it. The layering structure of an *ANNetwork* can be obtained with the *getLayering()* method.

1.5 Design notes

Inanna has been developed with heavily object-oriented approach, with total openness in mind. This is important for reuse of the library algorithms.

Chapter 2

Network objects

A "free and generic " network class. It provides a nice object-oriented hierarchy for different network objects. It does not currently contain any training algorithms. It also provides methods for cleaning up useless units and connections, determining visual positioning for the units, drawing an EPS image of the network, etc.

2.1 Training

2.2 Testing

2.3 Redefining networks

Sometimes, although hopefully rarely, the standard *FreeNetwork* class is not enough. One such case is the *RegulationNet*, which we will use as an example of redefining the baseclass. Its purpose is to act as a control device, where the control logic is implemented as a neural network. The network topology is inherently recurrent, and the network is executed until the its state reaches one of many possible criteria. Let us begin by defining the header:

```
class RegulationNet : public FreeNetwork {
    RegulationEnv* mRegulationEnv;                    /* Regulation environment */
    bool mStopState; /* Should the execution stop immediately? */
  public:
    RegulationNet (int n);

    // The copy conctructor uses mostly the baseclass.
    RegulationNet (const RegulationNet& orig) : FreeNetwork (orig) {
        mRegulationEnv = orig.mRegulationEnv; /* No deep-copying */
        mStopState = false; /* Assume not to be in stop state after copy */
    }

    // Creates a network with given number of nodes
    void make (int n);

    // Sets the regulation environment interface for the
```

```
        // network. RegulationNet doesn't destroy this interface when it
        // is destroyed
        void setEnv (RegulationEnv* env) {mRegulationEnv = env;}

        // Runs the network for number of cycles or until the endstate
        // attains given threshold (not used if -1 is given). Returns the
        // number of iterations actually performed
        int run (int cycles, int endstate=-1, double es_threshold=-1.0);

        // Tells the network to stop executing as soon as possible
        void stop () {mStopState = true;}
};
```

Further details are omitted here. The *RegulationEnv* is an interface to get external input to the network during the execution. It is defined simply as:

```
class RegulationEnv {
  public:
    virtual void getInput (RegulationNet& net)=0;
};
```

When called, the implementor of the interface can put any new input values to the network. A reference to the network is passed with the interface method, although the environment typically knows the network already. Output signals are sent back to the environment through the *ActuatingUnit*s, described below.

class *RegulationNet*

An agent-oriented control network that sends messages to different actuators (defined with *RegulActuator* objects) whenever the network state falls into a state subspace defined by the *ActuatingUnit* thresholds. As the name may imply, this class was originally made for genetic regulation networks, but it can be used for any control purposes.

class *ActuatingUnit*

A unit that sends a message to the user-inherited *RegulActuator* class whenever the activation of the unit exceeds a defined threshold.

class *RegulActuator*

User should redefine the *actuate* method of this class to receive messages from *ActuatingUnit*s.

2.3.1 Redefining neurons

The standard neuron used by the *FreeNetwork* is ***FreeNeuron***. Its basic attributes are an activation value, a bias value, and a transfer function. It can also be given three-dimensional coordinates, and it can be disabled with the *enable(false)* method. Disabled neurons are not updated and optionally not drawn in graphical representations of the network. Disabled neurons are also purged with the *FreeNetwork::cleanup (true)}* . The following example sets the output layer to use a linear transfer function.

```
FreeNetwork myNet;
myNet.makeUnits ("5-10-2");

ANNLayering& myLayering = myNet.getLayering();          /* Get the layering object */
int outputs = myLayering[-1];                           /* Size of the output layer */
int first = myLayering.getLayerIndex(-1);               /* Output layer start index */

for (int i=first; i<first+outputs; i++)                 /* For each output unit */
    myNet[i].setTfunc (FreeNeuron::ELLIOTT_TF); /* Set transfer function */
```

If we want to have really different neurons, we must inherit the basic neuron. There are some methods that we are required to implement, for example the default and copy constructors, and a cloning operation. In the following example, we define the *ActuatingUnit*. It is a unit that is given a pointer to an actuator object. When the activation of the unit exceeds a given threshold, it calls the *actuate* method of the actuator object. The purpose is to use recurrent neural networks as control devices.

```
class ActuatingUnit : public FreeNeuron {
  private:
    RegulActuator* mActuator;                               /* The callback-object */
  public:
    /* Default constructor. The (optional) parameter specifies the
       actuating callback object */
    ActuatingUnit (RegulActuator* act=NULL) : FreeNeuron(), mActuator(act) {}

    /* Copy constructor must be defined to allow cloning */
    ActuatingUnit (const ActuatingUnit& o) : FreeNeuron (o) {
        mActuator=o.mActuator;
    }

    /* Cloning operation is required to allow use of unit templates */
    ActuatingUnit* clone () const {return new ActuatingUnit (*this);}

    /* Sets the actuator callback-object for the unit */
    void setActuator (RegulActuator* act) {mActuator=act;}

    /* We reimplelement the transfer function to add new functionality */
    void transfer (FreeNetwork& net) {
        FreeNeuron::transfer (net); /* Transfer normally */

        if (mEnabled) {                                     /* Honor the enabledness */
            RegulationNet& regnet = dynamic_cast<RegulationNet&>(net);

            if (mActuator && mActivation > mActuator→threshold())
                /* The threshold was exceeded -> actuate */
                if (mActuator→actuate (regnet))
                    /* The actuate returned true -> order the regulation
```

```
                    network to stop running */
                regnet.stop ();
            }
        }
    };
```

We can now use the new neuron type to create a network using a neuron template:

```
RegulationNet::RegulationNet (int n) {
    ActuatingUnit* templ = new ActuatingUnit ();      /* Instantiate the template */
    templ→setActivation (0.0);                      /* Set initial activation value */
    templ→setBias (0.0);                            /* Set initial bias value */
    templ→setTFunc (ELLIOTT_TF);                    /* Set transfer function */
    templ→enable (true);                            /* Enabled by default */

    setUnitTemplate (templ); /* Set it as our template */
    make (n); /* Create n such units */

    // Perform other duties...
    mRegulationEnv = NULL; /* This has to be told to us later */
    mStopState = false; /* Not at stop state initially */
}
```

The connections between neurons are also objects, typically of class *FreeConnection*. That class can also be inherited in a similar manner to what we did above with the neurons.

2.3.2 An example

Suppose we want to create a network with standard sigmoid transfer function, three layers with sizes 33, 10 and 1, of which the last one has linear outputs (no sigmoidal squashing). Consider the following example:

```
#include <object.h>
#include <snns.h>

Main () {
    SNNSNetwork myNet;
    myNet.setDefActFunc ("Logistic");               /* Not really needed */
    myNet.make ("F33-10-1l");                       /* Set network topology */
    myNet.setTerminator ("GL5");                    /* Set early stopping method */
```

The standard logistic activation function is the default so it's not really necessary to set it here. The "F" in front of the layer description string causes every layer to be fully connected with shortcuts to all successive layers, not just to next one. In the final step we set the learning termination method to GL_5 [?].

Now we need to define pattern sets for training and testing. In our example, we split our training data to training and validation sets in 80:20 ratio:

```
FullPattSet trainingData ("data/mydata.pat");           /* In SNNS format */
FullPattSet testSet ("data/mytest.dat", 100, 33, 1);        /* Raw format */

/* Split the training data into training set and validation set */
FullPattSet trainingSet, validationSet;
trainingData.split (trainingSet, validationSet, 0.8);
```

Before we can train the network, training parameters must be specified. We'll use Rprop algorithm with parameters $\Delta_0 = 0.1$, $\Delta_{\max} = 50$, and $\alpha = 4$ (α is weight decay exponent).

```
myNet.setLearningFunc ("Rprop");         /* Use resilient backpropagation */
myNet.params.make (3); /* RProp requires three parameters: */
myNet.params[0] = 0.1; /* Lower eta */
myNet.params[1] = 50; /* Upper eta */
myNet.params[2] = 4; /* Exponent of the weight decay */
```

We are now ready to train and test the network.

```
/* Initialize weights to [-0.5...0.5] */
myNet.init (0.5);

/* Train for a maximum of 3000 cycles, with validation
   interval of 10 training cycles */
myNet.train (trainingSet, validationSet, 3000, 10);

/* Compute the MSE for the test set */
float MSE = myNet.test (testSet);

/* Count how many classification errors */
int classifErrors = myNet.testClassify (testSet);

/* Calculate error % */
float errorPerC = 100 * float(classifErrors) / testSet.patterns;
```

The number of classes is deduced directly from the number of outputs. The *testClassify* method is of course applicable only to classification problems.

The *PatternSet* and *ANNetwork* classes provide many other pattern-handling methods as well as nice informative methods for collecting statistics.

Chapter 3

Data handling

Different types of training and testing data are abstracted behind the *PatternSource* superclass. This abstraction makes it possible to have, for example, a dynamic data source, such as a physical device, attached to the neural network. Another common use would be automatic "*bootstrapping*" of patterns by adding "*jitter*", i.e., noise with an appropriate distribution (note that this can also be implemented by redefining the *FreeNeuron*).

3.1 Pattern sources (Class *PatternSource*)

Interface for all generic pattern handling, such as shuffling, copying, joining, selecting certain inputs, etc.

3.1.1 Implementing new pattern set source types

3.2 Pattern sets (Class *PatternSet*)

This is the most common type of pattern sets: a simple array of patterns which consist of respective input and output vectors.

FullPattSet contains also methods for equalization of pattern sets (see below).

class *ArrayPattSet*

Useful for teaching time series; pattern is given as a value vector, and the access operations map the vector into a standard pattern set.

3.3 Loading and storing patterns

PatternSet has an opaque interface for loading and saving different pattern file types, using the methods *load()* and *save()*.

```
PatternSet mySet;
mySet.load (“myset.pat”);
myset.save (“myset.pat”);
```

The file format is determined by file suffix. The toolkit currently supports the following file types:

- `.pat` – SNNS file format
- `.dt` – PROBEN1 file format
- Other suffixes (such as `.dat`) – raw data: number of inputs, outputs and patterns must be specified

3.3.1 Implementing new file formats

3.3.2 Design notes

Design patterns involved STRATEGY: data format are different strategies for storing data. ABSTRACT FACTORY: creation of *DataFormat* instances is centralized in *DataFormatLib*. FACTORY METHOD: each file format has a factory method to instantiate *DataFormat* objects. SINGLETON: There is only one instance of *DataFormatLib* and the *DataFormat* objects.

3.4 Equalization of data

PatternSet has two different methods for equalizing the patterns: *histogram equalization* and *gaussian equalization* (see below). These are defined in `equalization.h`.

The patterns of a pattern set can be equalized with equalization parameters acquired from the pattern set itself, or with parameters determined for another set. The validation and testing sets are typically equalized with the parameters acquired from the training sets. For example:

```
PatternSet trainset  ("trainset.pat");
PatternSet testset   ("testset.pat");

HistogramEqualizer equalizer (10000)
equalizer.analyze  (trainset); /* Analyze the data. */
equalizer.equalize (trainset); /* Apply to training set. */
equalizer.equalize (testset);  /* Apply to test set. */
```

3.4.1 Histogram equalization

The histogram equalization makes a histogram for each component vector $\vec{s}$ of the training pattern set. A histogram $\vec{h}$ is formed by counting the number of the vector elements in $\vec{s}$ that fall in each slots with domain function

$$d(x, \vec{h}) = \left\langle |\vec{h}| \cdot \frac{x - \min(\vec{s})}{\max(\vec{s}) - \min(\vec{s})} \right\rangle \tag{3.1}$$

where $\langle\rangle$ denotes taking the integer part of a real value. A typical histogram resolution is $|\vec{h}| = 100000$. The histogram is then transformed into a cumulative histogram with

$$c_i = \sum_{j=1}^{i} h_i. \tag{3.2}$$

The cumulative histogram can then be used to equalize values with

$$q(x, \vec{c}) = \min(\vec{s}) + (\max(\vec{s}) - \min(\vec{s})) \cdot \frac{c_{d(x,\vec{c})}}{|\vec{s}|}, \tag{3.3}$$

and the original component vector is equalized with

$$s'_i = q(s_i, \vec{c}). \tag{3.4}$$

This is done for all the pattern vector components. Other patterns can be equalized using the same equalization vectors.

We could imagine that the histogram equalization removes any clustering inside any one vector component, but also makes small variations within the clusters more easily detectable and possibly enhances clusterings within dependent component hypercubes.

3.4.2 Gaussian equalization

Mean $\bar{X}$ and standard deviation σ are calculated for each pattern set vector component set. Values are equalized with

$$y_i = \frac{x_i - \bar{X}}{\sigma} \tag{3.5}$$

so that the mean will be 0 and the standard deviation 1.0. This linear mapping does not lose any inherent clustering inside a single pattern component as the histogram equalization does.

3.4.3 Minmax equalization

Scales the data into an explisit range $[min, max]$. For example:

```
MinmaxEqualizer eq (0.0, 1.0);
```

3.4.4 Implementing new equalization methods

3.4.5 Design notes

Design patterns involved STRATEGY: equalization methods are strategies for equalizing data.

3.5 Drawing networks

The Miller and Kitano encoding methods do not provide two-dimensional coordinates for the neurons, so a drawing algorithm must be used. We introduce below a simple algorithm suitable for drawing feedforward networks:

1. Split the hidden units into layers using a simple rule:

$$layer(i+1) = \begin{cases} layer(i)+1 & \text{if}\, connected(i, i+1), \\ layer(i) & \text{if}\, \neg connected(i, i+1) \end{cases}$$

2. Position input and output units according to their index

3. Iterate from the first to the last hidden layer:

    (a) For each unit in the layer, calculate the "optimal" position in the sense that the unit would be positioned at the average y-position of source unit of all connections

    (b) Order the units in the layer according to their "optimal" y-positions

4. Scale and center and all layers to fit the print area

We implemented this algorithm in the ANN library, and used it in producing the network figures for this work. A major problem with the algorithm can be observed when the network is, for example, connected so that each hidden neuron is connected to the neuron with the successive index value. The hidden neurons are then ordered one neuron per layer, at the center of the layer, and the connections thus overlap each other.

3.6 Design notes on pattern handling

Chapter 4

Network creation

The network creation is

4.1 Topology builders

In the above hands-on example, we used a description string to define the structure of the MLP network. In the simplest case, we just list the sizes of the layers in the string, including the input layer. Each neuron on a specific layer will be connected to every neuron on the next layer. The *SNNSNetwork* has two additions in the description string. The letter "F" in the beginning of the string (for example, "`F2-2-1`") will cause every layer to be connected to *every* successive layer. The connections that jump over the next layer are often called *shortcut connections*. The second addition is that if the size of a layer is followed by letter "`l`" (for example, "`33-4-2l`"), linear output function will be used in the neurons of that layer. This is often used with output layer, when we want to have the output values in $\Re$.

The description string gives a fast way to create neural networks, but it is not the only way with the *FreeNetwork*. Next, we create a fully connected recurrent *FreeNetwork* network:

```
FreeNetwork myNet; /* Create network */
myNet.make ();  /* Create 100 neurons */
for (int i=0; i<myNet.size(); i++) /* Every neuron */
        for (int j=0; j<myNet.size(); j++) /* to every neuron */
                myNet.connect (i, j);
```

This can also be done with the method *connectFull*().

4.2 Loading and saving network to a file

FreeNetwork class does not have such file operations currently. The conversion operations between these two classes are described above for *SNNSNetwork*.

Chapter 5

Training algorithms

5.1 Algorithm library (*TrainerLib*)

5.2 Implementing new algorithms

5.3 Design notes on training algorithms

Chapter 6

Early stopping methods

ANNetwork handles the *early stopping* (termination) regularization during training.

Early stopping [?, ?, ?] is a way of controlling the model complexity by terminating the training when the model begins to overfit to the data. It is also a way for making the training faster, which is important with evolutionary neural networks, where we have to train numerous network topologies during the search to find the best one.

The available training patterns are divided into two parts: a *training set* and a *validation set*. The training set is used for the actual training of the network. Then, at certain intervals, the network is evaluated with the validation set. The purpose of this is to measure the generalization ability of the network. When the validation error begins to rise significantly, it is seen as a sign of overtraining and the training is terminated. The weight state of the network is stored after each validation test. After the training is terminated. the network state with the lowest validation error is restored.

6.1 Class *Terminator*

The base class for early stopping methods. Terminators can be constructed easily by description string using the *TerminatorLib::create()* function. For example,

```
Terminator* arnold = TerminatorLib::create ("GL5", validationSet, 10);
```

creates a *GLTerminator* (see below) with termination parameter GL_5.

One of the services provided by this base class is calculating the generalization loss (GL)[?]. GL is defined as

$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right) \tag{6.1}$$

where $E_{va}(t)$ is the validation error at training epoch t and $E_{opt}(t) = \min_{t' \leq t}(E_{va}(t'))$ is the minimum validation error so far.

6.2 Class *CycleTerminator*$\rightarrow$ *Terminator*

CycleTerminator is the most commonly used method for terminating learning.

6.3 Class *ErrorTerminator→ Terminator*

6.4 Class *CompoundTerminator→ Terminator*

A terminator consisting of multiple termination criteria.

6.5 Class *TerminatorT800→ Terminator*

A very fast terminator. It is otherwise equal to the UPTerminator, except that it does not restore the best encountered network state.

6.5.1 Class *SavingTerminator→ Terminator*

This is an abstract base class for terminators that restore the best reached network state after termination.

6.5.2 Class *GLTerminator→SavingTerminator*

This is the GL, or generalization loss termination method, described by Prechelt [?]. Notion GL_α means that the training is terminated after $GL(t) > \alpha$. Prechelt has used parameters around GL_2 and GL_5.

6.5.3 Class *PRTerminator→SavingTerminator*

This is a termination method used in the examples by Prechelt [?, p. 27]. Four parameters are needed. First is the GL parameter (typically 2 to 5), second is a training progress minimum threshold parameter (typically 0.1), third is a number of strips (typically 8), and fourth is a GL/P ratio threshold parameter (typically 3). Example: "`PR5,0.1,8,3`"

6.5.4 Class *PQTerminator→SavingTerminator*

Progress quotient (PQ) termination by Prechelt. It has one parameter, termination threshold, ranging typically from 2 to 5. Example: "`PQ2`".

6.5.5 Class *UPTerminator→SavingTerminator*

UP termination by Prechelt. It stops the learning when the generalization error increases in n successive strips. Very fast with parameter value 2. Example: "`UP2`".

6.5.6 Design notes

Design patterns involved Strategy: different termination methods are implemented as strategies. Abstract Factory: terminators are constructed by the *TerminatorLib::createTerminator()*

method. Factory Method: the terminator factory is implemented as factories for each termination method, which are stored in *TerminatorLib*. Memento: the *SavingTerminator* clones the network under training to store its state.

Chapter 7

Additional tools

We have written some additional tool classes for handling the problem-specific pattern sets and for collecting some statistics. These are located in the `neural/proj/comparison` directory.

Class	Description
ProblemStats	Collects statistics for regression (approximation) problems. Computes minimum, maximum and average MSEs and variances of MSEs for training, validation and different test sets. Also computes statistics of training times. Produces reports.
ClassificationStats	Provides similar statistics for classification problems.
LearningProblem	Superclass for handling different problem datasets. Provides interfaces for loading training, validation and test pattern sets and for iterating trough different variants. This way all problem datasets can be evaluated from single interface.
ArtificialProblem	Artificial datasets `XOR`, `letters`, `encoder` and `artif32`. No variants are available and training, validation and testing sets are equal.
Proben1Problem	PROBEN1 problems. Each dataset comes in three different permutations and six different partitionings.
BankruptcyProblem	The bankruptcy data has three training sets, of sizes 100, 200 and 400. Its test data has been divided into five different partitionings.

Chapter 8

MagiClib Common library

MagiClib is a object toolbox that has evolved over some years to fit personal programming needs. Therefore, it should not be considered as a complete object library. Its core classes have been tested only in minor projects by a single programmer, and some features may produce unexpected behaviour.

The documentation given below only scratches the surface of the library; the class documentations and the header files provide much better documentation for anyone intending to use the library.

8.1 Fundamental classes

We give here a description of the most important classes. The library contains also modules for

- runtime class information (`class.h`, `object.h`)
- basic mathematics (`Math.h`)
- matrix calculations (`Matrix.h`)
- regular expressions (`regexp.h`)
- streams (`Stream.h`)
- dynamically attributed objects (`attributed.h`)
- coordinate calculations (`coord.h`)
- date and time manipulation (`datetime.h`)
- file databases (`filedb.h`)
- HTTP-CGI processing (`http.h`)
- Mini-SQL interface (`msqlpp.h`)
- interprocess communication (pipes, sockets), and

- graphical operations.

And many others.

The graphics operations are built in an abstract window and window object hierarchy with graphics devices such as GIF, SuperVGA, LaTeX, and Encapsulated Postscript (EPS). For example, printing Postscript can be done using the EPS graphics driver.

8.1.1 *Object,* the base abstraction

Object is the abstract mother of (almost) all other classes. The most notable of its services is the dynamic object creation and runtime class information handling. Yes, runtime type information (RTTI) is provided by the modern C++ standard, but it is not yet powerful enough. Dynamic classes are defined as follows:

```
class MyClass : public Object { // In .h file
    decl_dynamic (MyClass);
};

impl_dynamic (MyClass, {Object}); // In .cc file
```

Such objects can be created dynamically with

```
Object* newobject = dyncreate ("MyClass"); // Create it
MyClass* myobject = dynamic_cast<MyClass*> (newobject); // Convert it
```

Object also provides a framework for outputting objects into a stream and reading them from a stream. For example, a *String* (see below) can be output to a stream with

```
OStream myFile ("filename.ext");
myFile.print ("Some text\n");
myFile.printf ("Formatted text:  %d\n", i);
String myString = "Hello, World!\n";
myFile << myString;
```

The standard output is defined with name *sout*, and standard error with name *serr*.

The dynamic object creation *requires* that all classes implement the empty constructor. This causes quite a lot of headache when the classes contain references to objects that would normally be given as parameters for the constructor. When this unfortunate situation occurs, the references must be initialized with fake null objects that are cast to the types of the references. Such constructors should have the `FORBIDDEN` macro as their definition.

8.1.2 Class *String* → *Comparable*

The string class of the library is rather powerful string implementation. It is rather “fat” though in the sense that it has quite many rarely used operations defined in the class.

Strings are *Comparable*. This means, for example, that you can put them into an ***Array<String>*** and issue the *quicksort*() command to sort the array.

Strings have also a built-in hash function. Additionally, the (8-bit) hash key of the string is cached in the string structure, so comparing the equality of two string is extremely fast, which is essential for fast *Map<String,String>* implementation (see below).

8.1.3 Linear containers

Class *Array<class>*

Array is the basic container of the class hierarchy. It contains references to indexed objects. The objects can be any structures, even native types. The *Array<>* "owns" the contained objects so that they can be destroyed by certain destructive class methods. All contained objects are destructed when the array is destructed. For non-destructive arrays, see *RefArray<>* below.

An array can be created and filled like this:

```
Array<String> myArr;
myArr.add (new String ("Foo"));
```

or like this:

```
Array<String> animals (3);
animals[0] = "Cat";
animals[1] = "Dog";
animals[2] = "Dragon";
```

The objects in the array can be sorted if they are inherited from *Comparable*, and they have implemented the *compare*-method.

To enable copying of arrays, you must implement *copy*() operator for your class.

RefArray<class> This is otherwise similar to the above container, except that the contained objects are not "owned" and are thus not destroyed by any of the class methods. The functionality of this array is somewhat different from *Array<>*.

PackArray<type> *PackArray* is a "packed array": instead of containing references to objects, it contains a single memory block. Using it is faster and can take considerably less memory than using regular arrays. The downside is that the size of the array is static.

Note that mathematical vectors are defined in *Vector* (in `Math.h`), which inherits *PackArray<double>*.

Table<type> *Table<>* is a simple two-dimensional container, similar to *Array<>*.

PackTable<type> *PackTable* is to *Table* what *PackArray* is to *Array*: it doesn't contain references to objects, but objects directly. Using it is faster and can take considerably less memory than using regular arrays. The downside is that the size of the array is static.

Note that mathematical matrices are defined as *Matrix* (in `Matrix.h`), which inherits *PackTable<double>*.

8.1.4 Class *Map<Object,Object>* (*StringMap*)

An association map is perhaps one of the most useful and flexible of the common data structures. Its enormous number of powerful uses is simply too huge to be considered here.

The very commonly used map of string keys and values has an abbreviated macro name *StringMap*.

8.1.5 Class *OStream*

The stream module of MagiClib is too complex to be previewed here more than just briefly. The most important class of the stream module is *OStream*, which is an output stream. It can be opened to a file or to a memory buffer.

Note also that the stream module is currently under construction and not yet very functional! The module does currently support abstract reading and writing of objects into binary files. This allows saving and loading complex objects very easily. The most central core classes such as *String* and *Array* support this currently, but not all other fundamental classes. The support for this behaviour must be coded for each new class by overloading the *operator<<(IStream&)* and *operator>>(OStream&)*. The stream objects also contain flags which control the formatting of the stream; by setting the appropriate flag you can save the objects either in binary or in text format. This feature makes it easy to "dump" object to standard output (especially with *DumpContext* stream).

8.2 Writing applications

Writing and debugging applications is easy with the *Application* class framework. It provides services such as command-line parsing, exception handling, and memory debugging. Since the application framework may not always be needed, it is implemented as a separate library. It is linked with `-lapp` flag for the compiler or linker. It defines the standard C *main*() function, so it should not be included unless necessary.

8.2.1 Main program

The "official" way of writing applications is inheriting the *Application* class as follows

```
class MyApp : public Application {
    decl_dynamic (MyApp); /* Macro adds runtime class information */
  public:
    MyApp () {...}                          /* The default constructor is required */

    // Then define the main () method
    void main (const Array<String>& params) {
        ...
    }
};

// This macro is required to create a runtime class object instance
```

```
impl_dynamic (MyApp, {Application});
```

and by implementing the *main*() method. However, it is usually easier to write programs using the *Main* macro. A hello-world program is done as follows

```
#include <object.h>                         /* The basic object hierarchy */
#include <applic.h>           /* The Application class and the Main macro */

Main (const Array<String>& params) {
    sout.print ("Hello, World!\n");
}
```

The parameters are given in the *Array<String>* passed for the macro.

8.2.2 Configuration files

The command-line parameters provide an easy way of defining simple configuration files for applications. The following program reads a file, puts it into a string map, and adds any parameters given in command line

```
Main () {
    /* Load config file if available */
    String configs;
    loadString (configs, "annalee.cfg");

    /* Parse the file into a string map */
    StringMap params;
    splitpairs (params, configs, '=', 0x0d);

    /* Add command-line parameters on top of those */
    params += app.paramMap();
    ...
```

.

The keys in the configuration file can be written either in long form as *<section>.<key>=<value>* or the section can be specified enclosed in brackets [*<section>*], and the keys belonging to that section after that.

Values can be used with simply calling: *parameters*[*"<section>.<key>"*]. The macro *getOrDefault(map,"key",default)* can be used to check if the value is in the parameter map, and if it is not, to use the given default value.

An example of a configuration file, for ANNALEE:

```
trainset=1400
testset=170
artifsize=50
logdir=.
datapath=/home/magi/c/annalee/proj/comparison
generations=100

[Population]
```

```
autoAdapt=1
boolRate=0.01
intRate=0.01
floatRate=0.1
floatVariance=0.1

[SimplePopulation]
size=50

# Miscellaneous parameters
[]
MutationRate.lowBound=0.01
Gentainer.recombFreq=0.5
BitFloatGene.graycoding=1
BitIntGene.graycoding=1

[EAStrategy]
elites=1
silent=0
minSimilarity=0.1

[Selection]
micro=8
q=3
eta+=1.2
adaptMicro=0
adaptQ=0
adaptEta+=0

[LearningGAEnv]
evals=1
noise=0.0
encoding=miller
maxTrainCycles=30
permutate=0
stripLen=10
terminator=none
termPart=0
optimizeParams=0

[ANNEncoding]
maxHidden=21
prunePassthroughs=0

[MillerEncoding]
pruneInputs=1
pcVariance=0.3
pcAverage=0.3

[KitanoEncoding]
rewrites=3
rules=64
nonTerminals=26

[NolfiEncoding]
neurons=32
types=16
tipRadius=auto
axonScale=1.0
xSize=9
ySize=22
```

Configuration files can also be included with the INCLUDE: directive. For example:

```
INCLUDE:../generic.cfg

[LearningEAEnv]
encoding=kitano
optimizeParams=0

[KitanoEncoding]
rewrites=3
rules=64
nonTerminals=26
```

This makes it easy to use general configuration files for all projects, and specializing files for specific projects.

Parameters can also be given from command line, as was done in the program example above. For example:

```
$ myprogram 4 2 --LearningEAEnv.encoding=kitano --SimplePopulation.size=50 “some other parameter”
```

The parameters 4, 2 and string “some other parameter” are passed to the *Main()* program in *mParams* array.

8.2.3 Exception handling

Any uncaught exceptions are finally caught by the application framework. The default action is to output the exception comments to the standard error output and then exit.

Assertions

The primary source for exceptions are probably various assertions that every decent programmer cultivates in his or her programs. These are often fatal.

8.2.4 Memory debugging

The application framework supports memory debugging, which is enabled with "*#define DEBUGNEW new*" line in `common/config.h`. The debugging is enabled on default on Solaris, and disabled on Linux (it doesn't currently work under Linux).

The memory debugging adds relevant information such as the location (file, function, line) of memory reservation to every reserved memory block. This feature understandably causes quite a lot of overhead, so it should be disabled in speed- or memory-critical applications. The memory debugging also initializes all reserved memory blocks with character '`\0xcc`' and all deleted memory blocks with '`\0xdd`'.

8.3 Using the library

The library is included and linked to your programs with the following compiler flags:

> `> gcc myprog.cc -I` *path-to-include-directory* `-L` *path-to-library* `-lmagic`

where we first give the include directory path for the library, and then the path to the directory where the file `libmagic.a` resides.

Be warned though: the library uses some very uncommon include files of the standard C++ library, such as exception handling. Such include files may not be in your standard include path, which may cause some problems.

Chapter 9

Notes

Portability issues

The libraries have been compiled using egcs release 1.0.3-1.1.3. Some of the parts of the libraries do not work with gcc (at least version 2.7.2.3 and earlier), mostly because of bugs and limitations in gcc, especially with templates and exceptions. The libraries have been compiled under Solaris 2.5 and Linux-2.1.36, using glibc-2.0.5 and up.

History

The INANNA library was constructed primarily in order to conduct the experiments for my master's thesis [?], as a part of the ANNALEE library that implements evolutionary learning of artificial neural networks. As such, the library is extremely experimental and tested mostly just for the purposes required by the thesis. It does, however, contain many developmental extensions and functionalities that surpass the requirements of the thesis. INANNA has been used for experiments in the Countess (Computational Intelligence for Business) project at Åbo Akademi. Some parts of INANNA were coded in 1993 as an assignment of a course in neural networks. The MAGICLIB has been slowly built since about 1989 as a personal programming toolbox, mostly during commercial software development. None of the libraries have been used by other persons than myself, so they should be considered to be very alpha.

Other

The name INANNA might refer to "Indigenous Artificial Neural Network A-something" (Architecture?), but actually the name comes from an ancient sumerian mother-goddess, whose name just happened to contain the critical letters ANN. Unfortunately, she has nothing to do with neural networks nor learning.

As the library has not yet reached a release status, the name should be considered as a work name.