+ − Insane

Artificial
Neural
Environment

Insane

Insane for Java user guide

Version 0.0.7

Contents

1 Introduction

The brain - that's my second most favourite organ!
– Woody Allen

Supported versions
This guide applies to `0.0.x` versions of Insane for Java library.

Requirements
Insane for Java library uses latest Java specificities, such as iterators. It is recommended to use Java version `1.6.0_18` or later.

Insane ("instrumented artificial neural environment") is designed to manipulate artificial neural networks. The following characteristics are seeked:

- Easiness-to-use
- Highly configurable solution
- Multiple initialization and training methods
- Efficient training and evaluation
- Stable memory consumption
- Training and evaluation decorrelation
- Lightweight solution executable on mobile devices (at least the evaluation part)

About this guide
This document is a step-by-step introduction to Insane's characteristics using the Java programming language.
Insane's main library and examples source code can be dowloaded from Insane's official website at `http://www.insane-network.org`.

2 Practical example

2.1 Description

This infamous example demonstrates how to create a simple multi-layered neural network. This network learns from the exclusive "or" table described by table 1, were "true" and "false" boolean values are respectively transposed to 1.0 and 0.0 values.

2.2 Source code

The source code is provided hereafter:

OpA	OpB	Result
false	*false*	*false*
false	*true*	*true*
true	*false*	*true*
true	*true*	*false*

Table 1: The exclusive "or" logical table

```
public final class ExclusiveOrBackPropagation {

    private static void println(double... values) {
        int max = values.length - 1;
        for (int i=0; i<max; i++) {
            System.out.print(values[i]);
            System.out.print(' ');
        }

        System.out.println(values[max]);
    }

    public static void main(String[] args) {
        try {
            TrainingInformation[] info = new TrainingInformation[] {
                new TrainingInformation(new double[]{0.0, 0.0}, new
                    double[]{0.0}),
                new TrainingInformation(new double[]{0.0, 1.0}, new
                    double[]{1.0}),
                new TrainingInformation(new double[]{1.0, 0.0}, new
                    double[]{1.0}),
                new TrainingInformation(new double[]{1.0, 1.0}, new
                    double[]{0.0})
            };

            ActivationFunction activation = new Sigmoid();

            // Set layers properties
            NetworkLayerProperties[] props = {
                new NetworkLayerProperties(2, activation),
                new NetworkLayerProperties(1, activation)
            };

            NeuralNetwork nnet = new NeuralNetwork(2, props);

            // Configure back-propagation training using default
            // configuration
            // The configuration parameters (ie learning rate and
            // momentum) may vary depending on the nature of the
// source training data
Random rand = new Random();
BackPropagationConfiguration configuration = new
    BackPropagationConfiguration();
TrainingMethod trainingMethod = new BackPropagation(rand,
     configuration);

// Set other training constraints
TrainingConstraints constraints = new TrainingConstraints
    ();
// Indicate the expected MSE
constraints.setMaxError(1E-4);
// Make sure the training process terminates
constraints.setMaxEpochs(1500);

// Train the network with uniformly chosen values
NetworkInitializer.initialize(new UniformDistribution(
    rand), nnet);
TrainingResults results = trainingMethod.train(
    constraints, TrainingMethod.ALL_ITEMS, nnet, new
    NoPruning(), info);

// It may happen that the training gets stuck in a local
// minimum. In this case, the required maximum error
// could not be reached after running all epochs, even
// if back propagation uses a momentum...
if (!results.valid(constraints)) {
    System.out.println("MSE is " + results.
        getMeanSquareError());
    System.out.println("Caution: the expected MSE is not 
        reached");
    System.out.println("The evaluated values may not be 
        accurate");
    System.out.println("Please run again");
    System.out.println();
    return;
}

System.out.println("MSE:              " + results.
    getMeanSquareError());
System.out.println("Best epoch:       " + results.
    getBestEpoch());

double[] outputs;
for (TrainingInformation ti : info) {
    System.out.println();
    System.out.print("Input values:     ");
    println(ti.getInputValues());
    System.out.print("Expected output:  ");
    println(ti.getExpectedOutputValues());
```

```
                System.out.print("Evaluated output: ");

                // Use the network to produce output values
                outputs = nnet.evaluate(ti.getInputValues());
                println(outputs);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The maximum error indicates the maximum deviation between theorical (provided during training) and evaluated outputs of the neural network. To make sure the training process does not loop endlessly, a maximum number or epochs[1] is also defined. These two constraints can be modified to find the best compromise between the required precision and the maximum delay necessary to train the network.

2.3 Results

An execution of this source code is performed in less than 1 second. The printed results show that the neural network successfully learnt from the exclusive "or" logical table training information as each evaluation is very close to the expected output:

```
MSE:               9.992529126095958E-5
Best epoch:        754

Input values:      1.0 1.0
Expected output:   0.0
Evaluated output:  0.006710131419360526

Input values:      0.0 1.0
Expected output:   1.0
Evaluated output:  0.9956579576799316

Input values:      0.0 0.0
Expected output:   0.0
Evaluated output:  0.003955773067477172

Input values:      1.0 0.0
Expected output:   1.0
Evaluated output:  0.995627804669113
```

[1]An epoch refers to the period in which the whole set of training information is presented to the training process.

Input values permutations
The training process modifies the order of the training information before starting an epoch. Indeed, the order of the results may vary. This order has no impact on the accuracy of the training operation.

2.4 Unsuccessful training

As discussed in section 4.7, it may happen that the training process does not succeed due to local minima of the error function. The maximum error (i.e. maximum deviation between theorical and evaluated outputs) allowed could not be reached. In this case, the following error message is displayed:

```
MSE is 0.8556225884761905
Caution: the expected MSE is not reached
The evaluated values may not be accurate
Please run again
```

This may come (among other parameters) from inappropriate initial weights distribution heuristics[2] and configuration settings. Re-running the training process is then required.

3 Creating networks structures

3.1 Definition

A multi-layered neural network (called "perceptron") is made of one or (usually) more layers. Each layer is composed of a list of neurons (as described by figure 1). A neuron responds to an input signal (i.e. an array of values) to produce an activation level (a single value). Consequently, the number of neurons of the last layer corresponds to the number of outputs of the neural network.

As illustrated by figure 2, each neuron produces a result in two steps. The input values are first mixed together (along with a bias) using a combination function c which produces a single value. The most commonly used combination function is called "linear combination function". This function is defined as follows:

$$c\left(x_1, \ldots, x_k\right) = -b + \sum_p w_p \times x_p$$

The resulting value of the combination function is then given to an activation function a to determine the activation degree (i.e. final result) of this particular neuron.

[2]This particular point will be discussed in section 4.3.

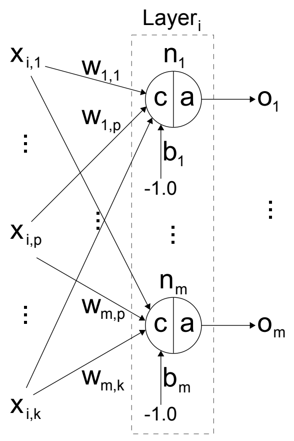

Figure 1: Feedforward neural network layer

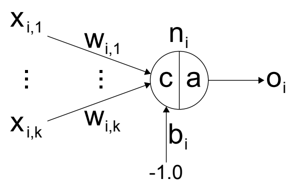

Figure 2: Neuron components

3.2 Source code

Creating a neural network requires to indicate the number of layers and the composition of each layer (number of neurons and behaviour of each neuron), as shown by the following (partial) source code:

```
// Define the network properties:
//   - 2 layers
//   - First layer contains 2 neurons with no initial bias
//   - Second layer contains a single neuron with an initial
//        bias of 0.5
ActivationFunction activationFunction = new Sigmoid();
NetworkLayerProperties[] props = {
    new NetworkLayerProperties(2, activationFunction),
    new NetworkLayerProperties(1, 0.5, activationFunction)
};

// Create a network which expects 2 inputs (i.e. an array of
// 2 doubles is required to ask the network for an evaluation)
// The evaluation result is an array which contains a single value
// (last layer contains only one neuron)
NeuralNetwork nnet = new NeuralNetwork(2, props);
```

4 Training networks

Training a neural network consists in performing synaptic weights and biases assignments to match a mathematical function defined by a set of samples.

The training process is the most costly phase when using a neural network. The training time and accuracy depend on many parameters:

- The training method
- The number of different training information
- The network structure (i.e. the number of layers, the number of neurons per layer, the combination and activation functions)
- The initial weights and biases values
- The maximum number of epochs
- The maximum error required

The "Mean Square Error" (MSE), also called quadratic error, corresponds to the sum of square deviations between evaluated (practical) results and theorical outputs. This error is usually determined during the training phase. The aim of the training is to update the network weights so that this error is minimized.

4.1 Training methods classification

Insane supports both stochastic and sequential training methods.

Stochastic training method
A stochastic (or "incremental") training method modifies the weights and biases values of the neural network being trained each time a training information item is presented. This means that these values are modified several times during a given training epoch.

Sequential training method
Contrary to a stochastic approach, a sequential (or "batch") training method computes the global error function for all training information items before modifying the current weights and biases values. Hence these values are modified once per training epoch.

In the particular case of sequential methods, Insane provides the ability to use training information from a source file[3]

Pros and cons of batch training from a file
This process is recommended when a large amount of training information is used, as it guarantees small memory consumption. The main drawback is the loss of good performances: file opening and closing operations are time and processor consuming.

Compared with the use of a list of pre-loaded training information items, the training time is extended in an average order of magnitude of 10[4].

4.2 Overall process

The actual training of a neural network requires to select an appropriate training method (i.e. algorithm). The whole training process is composed of the following steps:

1. Create a neural network with default biases values
2. Initialize the weights of this network, either from a previously saved network or generating new values using a heuristic
3. Select and configure an appropriate training method (back-propagation for example)
4. Set training constraints, such as the maximum error and the maximum number of running epochs. The training process will fail when none of these constraints is provided (to avoid endless looping)
5. Get training information

[3]Please refer to section 7.2 for further explanations and source code.
[4]Please refer to benchmarks for further details and comparisons.

6. Use the training method to train the network based on the training information. This training phase may integrate a pruning of the network

Although the expected MSE is normally reached, robust implementations may require to analyze the training results, such as in the provided example of section 2.

4.3 Setting initial weights and biases

Without any initialization, each weight and bias of a newly created neural network is set to 0.0. These internal weights and biases must be initialized with non-zero values in order to successfully execute a converging training of the network.

4.3.1 Setting default biases

An initial bias value in range $[0.0, 1.0]$ can be assigned to each layer of a neural network. This bias is set in the network layer properties when creating the network stucture. This bias value may impact the training accuracy.

4.3.2 Generating initial weights

The initial weights and biases values attributed to the neural network before training must be chosen with care. Combined the biases, they condition the whole training process. A good distribution of initial values can avoid local minima and may reduce the required number of epochs to reach the global minimum error up to $53,9\%$[5].

There is no good or bad heuristic to initialize weights with. The optimal strategy depends on the training algorithm, the structure of the network and the set of training information.

Insane suggests different distribution strategies for generating initial weights values. Each strategy is applied by the common weights initializer, which makes sure extreme values (i.e. -1.0 and 1.0, depending on the distribution strategy) and 0.0 are never used.

About a fixed value strategy
Empirical tests demonstrate that a strategy which uses a pre-defined constant value (each weight has the same value) is not applicable as training methods never reach the global minimum of the error function.

[5]Source: Y. F. Yam, Tommy W. S. Chow and C. T. Leung. *A new method in determining initial weights of feedforward neural networks for training enhancement*. Neurocomputing, vol. 16, issue 1, p. $23 - -32$, 1996.

4.3.3 Uniform distribution

The uniform distribution (see figure 3) is historically the first distribution implemented in Insane. It provides average training results.

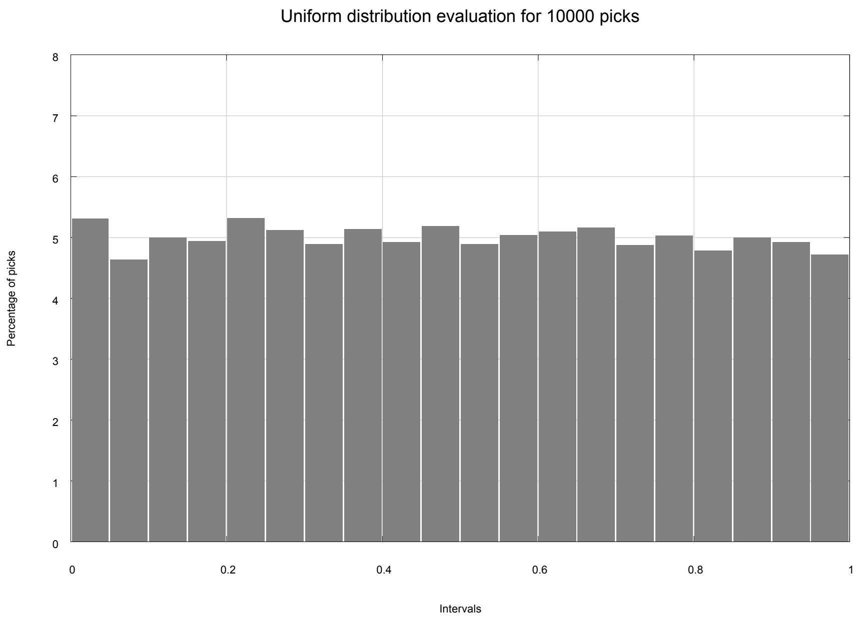

Figure 3: Initial weights picks in range $[0.0, 1.0]$

4.3.4 Symmetrical uniform distribution

The symmetrical uniform distribution (see figure 4) picks values in range $[-1.0, 1.0]$ instead of $[0.0, 1.0]$.

4.3.5 Normal distribution

The normal distribution picks values around 0.5, following a gaussian distribution illustrated by figure 5. Empirical results show that such distribution gives better results than the default uniform distribution.

4.3.6 Symmetrical normal distribution

Based on the simple normal distribution presented in section 4.3.5, the symmetrical normal distribution picks values around -0.5 and 0.5, following a double gaussian

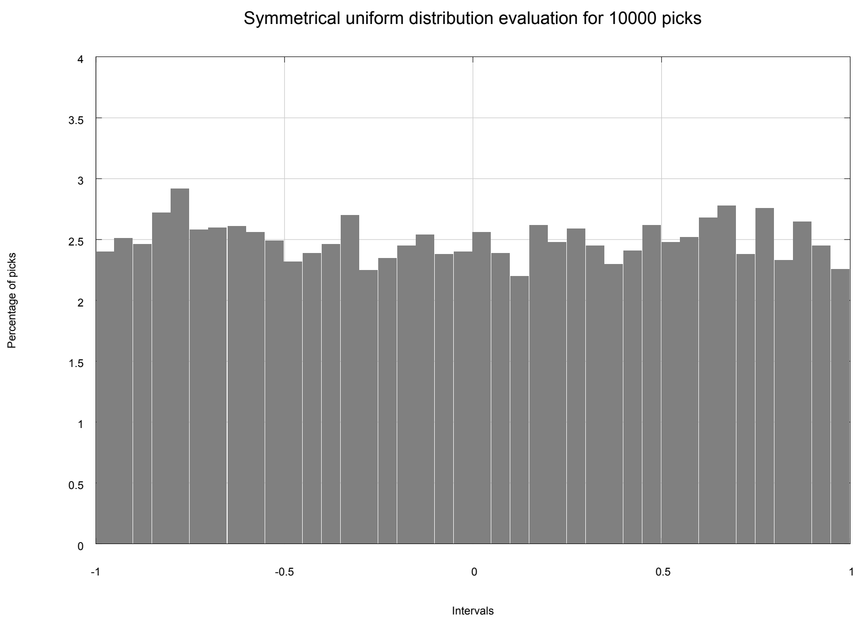

Figure 4: Initial weights picks in range $[-1.0, 1.0]$

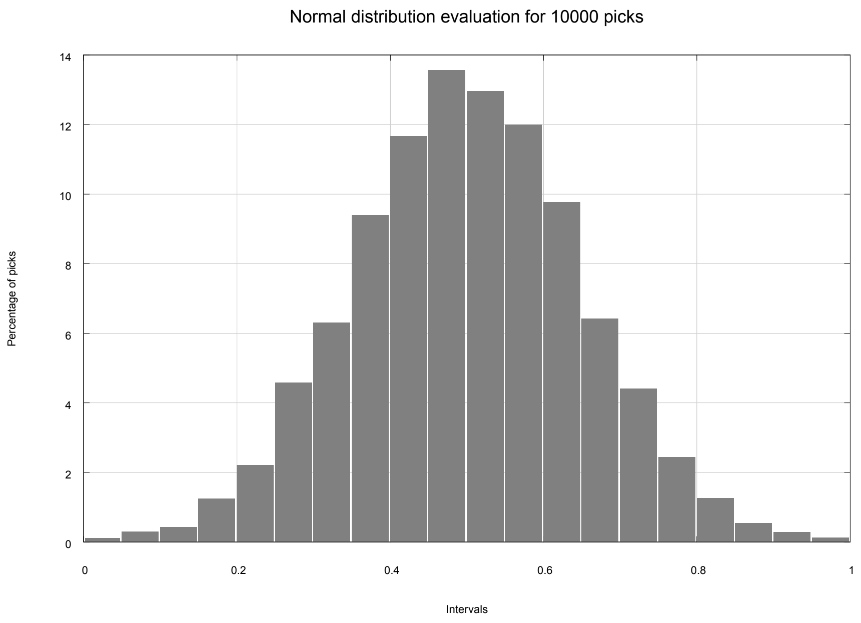

Figure 5: Initial weights picks in range $[0.0, 1.0]$

distribution illustrated by figure 6. Empirical results show that such distribution gives better results than the default uniform distribution.

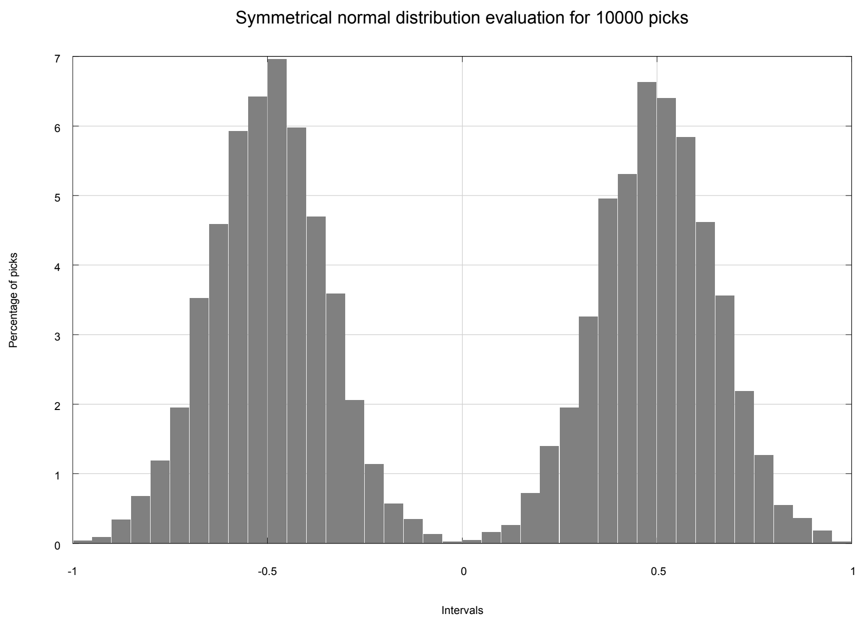

Figure 6: Initial weights picks in range $[-1.0, 1.0]$

4.4 Obtaining training information

The example of section 2 partially indicates how to train a network using coded training information (i.e. the training information list is part of the source code of this example).

In most cases, training information consists of a large amount of data which must be loaded from external sources.

Insane provides default functionalities for loading (resp. saving) training information either from source (resp. destination) files and streams.

4.4.1 Source code

Here is a simple example of training information storage and retrieval using files (manipulating input and output streams is similar):

```
public final class SaveLoadTrainingInformationTest {

    public static void main(String[] args) {
        try {
            // Change with your own paths
            File infoFile1 = new File("info.dat");
            File infoFile2 = new File("info_copy.dat");

            // Define training information
            TrainingInformation[] info = new TrainingInformation[] {
                new TrainingInformation(new double[]{0.0, 0.0}, new
                    double[]{0.0}),
                new TrainingInformation(new double[]{0.0, 1.0}, new
                    double[]{1.0}),
                new TrainingInformation(new double[]{1.0, 0.0}, new
                    double[]{1.0}),
                new TrainingInformation(new double[]{1.0, 1.0}, new
                    double[]{0.0})
            };

            // Save to a file
            TrainingInformationSaver saver = new
                DefaultTrainingInformationSaver();
            saver.save(info, infoFile1);

            // Load and store again to compare files
            TrainingInformationLoader loader = new
                DefaultTrainingInformationLoader();
            List<TrainingInformation> infoList = loader.load(
                infoFile1);
            info = TrainingInformationManager.toArray(infoList);

            saver.save(info, infoFile2);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Make sure to modify the files paths before running this example.

The default loader allows to retrieve training information stored using the default saver. The training information is first retrieved in a list to allow easy manipulations (adding or removing training information).

Insane provides complementary operations, such as the ability to load each training information item separately.

4.4.2 Commenting saved training information

The default training information loader allows to manually add empty lines and add comments using '%' in a similar way the network loader does.

4.5 Executing a supervised training method

The aim of a supervised training method is to make the neural network learn from samples (a set of training information). Two major parameters are seeked:

- Fast convergence
- Local minima avoidance

Insane currently supports the training methods presented hereafter.

4.5.1 Back-propagation

Insane provides a stochastic implementation of the common back-propagation ("BP") method. This method asks for two configuration parameters: a learning rate and a momentum. Both parameters must be in range $[0, 1]$. Default values should get good results for most cases, however it is recommended to manually modify these parameters depending on the nature of the training information.

About a batch version
This method was primarily proposed in batch mode, where the network weights and biases are modified once per epoch, after the whole set of samples has been presented. Empirical evaluations demonstrated that the incremental version was faster and provided more accurate results.

4.5.2 Resilient propagation

The resilient propagation ("RPROP") runs in sequential mode. It appears to be a very fast method which gives minimal errors, although it tends to frequently diverge using default parameters values.

For example, when the RPROP algorithm is used in placed of the back-propagation training method (setting a positive momentum of 1.5) and initial weights are correclty chosen, the following output is produced:

```
MSE:                8.14394420217272E-5
Best epoch:         54

Input values:       0.0 0.0
Expected output:    0.0
Evaluated output:   1.1981904535834107E-10
```

```
Input values:     0.0 1.0
Expected output:  1.0
Evaluated output: 0.9966610690534616

Input values:     1.0 0.0
Expected output:  1.0
Evaluated output: 0.9999999999999902

Input values:     1.0 1.0
Expected output:  0.0
Evaluated output: 1.1952585154914498E−10
```

To obtain comparable mean square errors (and better evaluation results), the optimal number of training epochs is reduced to 54, compared with 754 epochs for the traditional back-propagation (please refer to section 2.3 to compare results). . .

Local minima issue
This is the default RPROP method. While this method can be extremely fast in converging to a solution, it still suffers from the local minima problem. This means that, depending on the initial weights of the neural network, this method may not converge to the global optimum. Using initial uniformly distributed weights, this method usually gets stuck around a local minimum of 0.5, from which it cannot escape, as shown in next execution results.

```
MSE is 0.5005144807007855
Caution: the expected MSE is not reached
The evaluated values may not be accurate
Please run again
```

4.5.3 Mixed training method

This training method is built on top of all other training methods. It allows to call other training methods sequentially. Each training method is invoked until its associated constraints are met (i.e. the expected MSE is obtained or the maximum number of epochs is reached).

Note that the expected MSE can be reach although all participating training methods have not been invoked.

4.6 Pruning the network

Pruning
Pruning techniques aim to reduce the number of active connection between neurons, thus improving the network ability to extrapolate from unkown inputs.

The pruning process is executed each time an epoch terminates. The pruning method is specified when calling the main training operation of a training method.

Insane currently defines the pruning techniques described hereafter.

4.6.1 No pruning method

The default pruning technique implemented by Insane, called `NoPruning`, consists in leaving the network "as is" (i.e. without modifying any weight or bias). Alternatively, a `null` parameter can be set to the training method in order to indicate that no pruning applies.

4.6.2 Threshold pruning method

This pruning method disables all weights or biases values below a predefined threshold by setting their values to zero.

4.7 Analyzing training results

As previously said, it may happen that the training process fails as it gets stuck inside a local minimum. In this case, the resulting error is greater than the expected error. Among others, this may occur due to an unlucky randomization of the initial weights or a non optimal configuration of the training method.

Insane provides complete information to determine whether the training process succeeded or failed. A training result is composed of:

- The best MSE obtained during the training (or the first MSE below the required MSE)
- The epoch count of this MSE (best epoch)

Adding the information of the best epoch is useful when one want to determine the most appropriate maximum number of training epochs: it is pointless to allow a maximum of 100000 epochs when the MSE is usually reached after 500 epochs...

5 Evaluating inputs

Insane offers three operations to perform inputs evaluation.

5.1 Common evaluation

The `evaluate(double[])` operation is usually used to perform an evaluation. In this case, the provided argument is internally duplicated to the first layer's inputs array before starting the evaluation.

5.2 Constrained evaluation

It may happen that the running environment requires to save time and memory space. In this case, the `getInputs()` and `evaluate()` operations can be used:

1. Use `getInputs()` to set the normalized input values within the inputs array
2. Use `evaluate()` to perform the evaluation of the inputs and get the evaluation result in return

A typical example is provided hereafter:

```java
public final class ConstrainedEvaluation {

    public static void main(String[] args) {
        try {
            // 1. Create a neural network
            int nInputs = ...;
            NetworkLayerProperties[] props = ...;
            NeuralNetwork nnet = new NeuralNetwork(nInputs, props);

            // 2. Train the network
            ...

            // 3. Perform a constrained evaluation
            double[] inputs = nnet.getInputs();

            // 31. Modify the input values
            inputs[0] = ...;
            ...

            // 32. Evaluate the new input values
            double[] outputs = nnet.evaluate();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Note that Insane also provides the `getOutputs()` operation to retrieve the evaluation results.

6 Saving and restoring networks

Saving networks is a very important feature to avoid a training phase each time a network is created. It offers the ability to save the network structure as well as its weights and biases to a destination stream. This stream can be loaded to re-build the network at a later time.

6.1 Description

The following example demonstrates how to save and load a network using the default saver and loader. Finally, the restored network is saved to another file to verify it was correctly loaded. To make it simple, the saved network is not trained.

6.2 Source code

! Make sure to modify the files paths before running this example.

```java
public final class SaveLoadNetwork {

    public static void main(String[] args) {
        try {
            // Change with your own paths
            File netFile1 = new File("network.net");
            File netFile2 = new File("network_copy.net");

            // Create a logistic neural network
            ActivationFunction activation = new Sigmoid();
            NetworkLayerProperties[] props = {
                new NetworkLayerProperties(4, 0.5, activation),
                new NetworkLayerProperties(2, activation)
            };

            NeuralNetwork nnet = new NeuralNetwork(5, props);

            // Assign randomly distributed weights
            NetworkInitializer.initialize(new UniformDistribution(new
                 Random()), nnet);

            NeuralNetworkSaver saver = new DefaultNeuralNetworkSaver
                ();
            saver.save(nnet, netFile1);

            NeuralNetworkLoader loader = new
                DefaultNeuralNetworkLoader();
            nnet = loader.load(netFile1);
            // Save a copy of this network
            saver.save(nnet, netFile2);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

6.3 Commenting a saved network

The default neural network loader allows to manually add empty lines and add comments using '%'. A comment can follow some information on a given line (which means that only the right part of the line is commented).

For example, the default network loader is able to parse this commented network:

```
% This is a commented example of a saved
% neural network using the default saver

% Header to make sure that the following
% information describes an Insane neural network
Insane network description

% The network expects 3 entries and is made of
% 2 layers (1 hidden layer and 1 output layer)
3 2

% The first layer has the following characteristics:
%   - 3 neurons
%   - bias value of 0.2
%   - sigmoid activation function
3 0.2 sigmoid

% The second layer is defined as follows:
2 0.5 sigmoid % Bias equals 0.5

% Weights of the first layer (a single line per neuron)
0.63468314562631 0.39658242179123 0.28137518464209 0.59493772158695
0.12796423914789 0.50434285634176 0.85797599570718 0.72043025541313
0.66234759805424 0.04564729968697 0.09234072935261 0.24969324045428

% Weights of the second layer (a single line per neuron)
0.85421334552185 0.57034562635145 0.25964506887161 0.46856497405905
0.53224535189452 0.99319286370061 0.20919801652207 0.04530912567847
```

7 Miscellaneous

7.1 Multi-layered neural networks design

According to Kevin Swingler[6], the best way to design the appropriate multi-layered network consists in verifying the estimated error using both the same set of training information and a set of extra (test) information. He presents a simple set of heuristics for arriving at a correct neural network model:

- If the training error is low and the test error is high, there are too many weights
- If both training and test error are high, there are too few weights
- If the weights are all very large, there are too few weights
- Adding weights is not a panacea: do not add too many weights
- The initial weights of an untrained network must be (randomly) set over a small range of values (say in range $[-1.0, 1.0]$

Extra heuristics
One could also add that a random weights initialization is not the most suitable solution to guarantee the convergence of the training methods. Moreover, initial weights set to 0.0 must be avoided as this absorbing element would prevent from multiplicative modifications of these weights.

7.2 Running under constrained environments

Insane is designed so that evaluation tasks are decorrelated from training procedures.

When only evaluation from an initialized neural network is required (on a mobile device for example), the `insane.training` package and its sub-packages can be removed from the Java library (`.jar`) file, as well as all unused combination and activation functions. It is still possible to load and save neural networks and perform evaluations.

Moreover, in the particular case of sequential training methods, Insane offers the ability to save memory by directly training from a source training information file. This file is opened and closed at the beginning and end of each training epoch. During an epoch, training information items are loaded one after another. The following source code gives some guidelines to use this functionnality[7]:

```
// Create and initialize a neural network
NeuralNetwork nnet = ...
```

[6]K. Swingler. *Applying neural networks: a practical guide*. Academic Press, 1996.

[7]Please refer to the `SequentialTrainingMethod` class documentation for further information.

```
DistributionStrategy distribution = ...
NetworkInitializer.initialize(distribution, nnet);

// Use a sequential training method
SequentialTrainingMethod trainingMethod = ...

// Set your own training information source file
File srcFile = ...

// Select a loader which supports single information loading
TrainingInformationLoader infoLoader = new
    DefaultTrainingInformationLoader();

// Perform training directly from source file
TrainingConstraints constraints = new TrainingConstraints();
constraints.setMaxError(...);
constraints.setMaxEpochs(...);
TrainingResults results = trainingMethod.train(srcFile,
    TrainingMethod.ALL_ITEMS, TrainingMethod.ALL_ITEMS, infoLoader,
    constraints, nnet);
```

7.3 Multi-threaded environments

Insane's neural networks are not thread-safe. However, a `clone()` operation is proposed so that a copy of a network can be used in each thread. This can be useful when trying different training methods or evaluating inputs.

Muti-threaded evaluations
For performance reasons, the current implementation uses temporary arrays to store evaluated data between layers as well as the final results produced by a given neural network. This implies that each evaluation stores its results into a single array (i.e. the new results replace the previous ones). Indeed, a neural network must perform a single evaluation at a time.

8 More functionalities

Complementary to Insane core functionalities, an extra set of packages may be used when other functionalities are needed.

8.1 Networks storage and retrieval

The default procedures for storing and restoring neural networks are based on a pure textual format. A complementary package also provides an XML-based format to represent neural networks.

9 Conclusion

Insane is still in its early development stage. Although the current back-propagation and resilient propagation training algorithms proved to be efficient compared with other available libraries[8], they can still be improved in various ways (adaptive parameters) to minimize the number of required epochs to reach the most accurate solution (i.e. with a minimal error).

Appart from implementing new training techniques, a critical issue consists in developping efficient weights initialization schemes which guarantee the convergence of the training methods.

Finally, Insane's libraries and their associated documention (including the Javadoc files) can be improved with your reviews and comments.

For more information, please visit `http://www.insane-network.org`.

Thank you for your help in contributing to make Insane the most efficient and powerful open-source artificial neural network environment on the market.

– Nathanaël COTTIN

[8]Please refer to benchmarks for more information.